INTERNATIONAL TELECOMMUNICATION UNION

CCITT

THE INTERNATIONAL
TELEGRAPH AND TELEPHONE
CONSULTATIVE COMMITTEE

Z.100 Annex D

(11/1988)

ANNEX D TO RECOMMENDATION Z.100
SDL USER GUIDELINES

SDL USER GUIDELINES

Reedition of CCITT Recommendation Z.100 Annex D
published in the Blue Book, Fascicle X.2 (1988)

NOTES

1 CCITT Recommendation Z.100, Annex D was published in Fascicle X.2 of the *Blue Book*. This file is an extract from the *Blue Book*. While the presentation and layout of the text might be slightly different from the *Blue Book* version, the contents of the file are identical to the *Blue Book* version and copyright conditions remain unchanged (see below).

2 In this Recommendation, the expression "Administration" is used for conciseness to indicate both a telecommunication administration and a recognized operating agency.

© ITU 1988, 2008

All rights reserved. No part of this publication may be reproduced, by any means whatsoever, without the prior written permission of ITU.

ANNEX D

(to Recommendation Z.100)

SDL User Guidelines

TABLE OF CONTENTS

Page

D.1 Preface 4

D.2 Introduction 5

D.2.1 General overview of SDL 5
D.2.2 Syntax forms of SDL 6
D.2.3 Applicability of SDL 7

D.3 Basic concepts of SDL 9

D.3.1 System 9
D.3.2 Blocks 10
D.3.3 Channels 11
D.3.4 Signals 13
D.3.5 Signal routes 13
D.3.6 System and block diagrams 14
D.3.7 Comments and text extension 17
D.3.8 Processes 20

D.3.8.1 Process creation 23
D.3.8.2 States 25
D.3.8.3 Inputs 33
D.3.8.4 Saves 41
D.3.8.5 Enabling conditions and continuos signals 45
D.3.8.6 Outputs 49
D.3.8.7 Tasks 51
D.3.8.8 Decisions 52
D.3.8.9 Joins and connectors 55

D.3.9 Procedures 57

D.3.10 Data handling 60

D.3.10.1 Variable declarations 60
D.3.10.2 Revealed/Viewed variables 62
D.3.10.3 Exported/Imported variables 63
D.3.10.4 Expressions 64

D.3.11 Expressing time in SDL 65

D.3.12 Qualifiers usage 66

D.3.13 Syntax of names 67

D.4 Structuring and refining SDL systems 70

D.4.1 General 70
D.4.2 Criteria for partitioning 70
D.4.3 Block partitioning 70
D.4.4 Block tree diagram 74
D.4.5 Channel partitioning 75
D.4.6 System representation in case of partitioning 77
D.4.7 Refinement 79

Page

D.5 Additional concepts 83

D.5.1 Macros 83
D.5.2 Generic systems 85
D.5.3 Services 86

D.5.3.1 General 86
D.5.3.2 Priority signals 90
D.5.3.3 Transformation 93

D.5.4 Guidelines for state-oriented representation and pictorial elements 97

D.5.4.1 General comments on state-oriented representation 98
D.5.4.2 State picture and pictorial elements 98

D.5.5 Auxiliary diagrams 103

D.5.5.1 State overview diagram 103
D.5.5.2 State/signal matrix 104
D.5.5.3 Sequence chart 105

D.6 Definition of SDL data 106

D.6.1 Guidelines on data in SDL 106

D.6.1.1 General introduction 106
D.6.1.2 Sorts 107
D.6.1.3 Operators, literals and terms 107
D.6.1.4 Equations and axioms 109
D.6.1.5 More on equations and axioms 111

D.6.2 Generators and inheritance 113

D.6.2.1 Generators 113
D.6.2.2 Inheritance 115

D.6.3 Views on equations 117

D.6.3.1 General requirements 117
D.6.3.2 Function application on constructors 118
D.6.3.3 Test-set specification 119

D.6.4 Features 119

D.6.4.1 Hidden operators 120
D.6.4.2 Ordering 120
D.6.4.3 Sorts with fields 121
D.6.4.4 Indexed sorts 122
D.6.4.5 Default value of variables 122
D.6.4.6 Active operators 122

D.7 Additional guidelines for drawing and writing 123

D.7.1 SDL/GR guidelines 123

D.7.1.1 General 123
D.7.1.2 Inlets and outlets 123
D.7.1.3 Symbols 124
D.7.1.4 Template 124

D.7.2 SDL/PR guidelines 126

D.8 Documentation 126

D.8.1 Introduction 126
D.8.2 Types of system representation 126
D.8.3 Document structure 131
D.8.4 Referencing mechanism 132
D.8.5 Classification of documents 132
D.8.6 Mixing of SDL/GR and SDL/PR 134

Page

D.9 Mappings 136

D.9.1 Mapping between SDL and CHILL 136
D.9.2 Mapping between SDL/GR and SDL/PR 139

D.10 Application examples 139

D.10.1 Introduction 139
D.10.2 The service concept 140

D.11 Tools for SDL 155

D.11.1 Introduction 155
D.11.2 Classes of tools 155
D.11.3 Document input 155
D.11.4 Document verification 156
D.11.5 Document reproduction 156
D.11.6 Document generation 156
D.11.7 System modelling and analysis 157
D.11.8 Code generation 157
D.11.9 Training 157

D.1 *Preface*

The CCITT Specification and Description Language, known as SDL, was first defined by Recommendations Z.101 to Z.103 in 1976 (the Orange Book, Volume VI.4), later extended in Recommendations Z.101 to Z.104 in 1980 (the Yellow Book), further extended and reorganized into Recommendations Z.100 to Z.104 in 1984 (the Red Book). In the study period 1985-1988 the language has been further extended and harmonized, the existing Recommendations have been merged into a single one and the mathematical definition has been provided.

User guidelines are needed to facilitate the use of the SDL in application to a wide range of telecommunications systems. The aim of the user guidelines is to assist users in the understanding of the SDL Recommendation and in their application to various areas.

SDL is being widely used by the CCITT and its member organizations, and the range of applications of SDL continues to increase. These user guidelines are intended to assist people who are considering or starting to use SDL, by supplementing SDL Recommendations with useful advice and helpful examples. It is realized that there will be some overlap between the user guidelines and the Recommendations; this is believed to be desirable, to make the user guidelines self-sufficient and readable. Nevertheless, it is the Recommendation which is the leading document.

D.2 *Introduction*

D.2.1 *General overview of SDL*

SDL can be used both to specify the required behaviour of a system and to describe the actual behaviour of a system. SDL has been designed having in mind the specification of behaviour of telecommunications switching systems, but can also be used to model other applications. As a matter of fact, SDL is well-suited to all systems whose behaviour can be effectively modelled by extended finite-state machines (§ D.2.1.1) and where the focus is to be placed especially on interaction aspects.

SDL also can be the base for a methodology of documentation to represent completely a system specification or description. In this context the meaning of specification and description is related to their use in a system life cycle. Both define the functional properties of a system in an abstract way. The description will usually include some design dependent aspects (e.g. error handling) and will be more complete with respect to functional details. Both should be related to the concrete system design in a consistent manner and serve as documentation afterwards.

SDL may be used to represent, at various levels of detail, the functional properties of a system, or a function or facility. The functional properties consist of some structural properties (block interaction diagrams) as well as behaviour. By "behaviour" is meant the way of reacting upon received signals (inputs), i.e. doing actions, e.g. sending signals (outputs), asking questions (decisions) and doing tasks.

Specifications may be very broad and general when an Administration wishes to explore the possibilities of updating a system with new features, new services, new technology, etc., while allowing the supplier to offer a wide range of design solutions. This type of specification is often not very detailed. The other extreme is a specification in which an Administration is requesting a replacement or addition to an existing exchange. This specification would probably have a greater level of detail, because of the very detailed specification of interfaces necessary.

A specification and a description might be identical. In any case it is preferable in new development that the design is derived from the specification so that compliance is ensured.

In general, descriptions are written by suppliers in response to a specification (although they can be written to describe systems that the supplier wishes to sell). A description will usually have more levels of detail than the specification because of the need to describe the detailed behaviour of the system. For simplicity sake the SDL representation will be referred to just as specification in the following sections.

It is to be noted also that SDL provides means of describing a system with various degrees of formality.

First, it is possible to describe a system using the SDL constructs with associated natural language. The resulting specification conveys information only to a reader with knowledge of the context but not to a machine. Only very limited checks can be performed automatically.

Second, it is possible to associate with the SDL constructs formal statements consisting of elements of defined types and operators on these elements. The properties of these elements are not specified: an example is "Connect A-B" where A and B are of type subscriber and connect is an operation allowed for that type. The resulting specification conveys information to readers who know the meaning of the operators used. A machine can understand the specification to a certain level and can perform checks on it, but it cannot perform complete checks nor "implement" the system because the properties of the operators are not fully specified.

Third, it is possible to also provide all the properties of all the operators. In this latter case the specification is completely formal and a machine can perform all the checks and conceptually implement the systems described.

Depending on the goal, the specifications can be tailored to the user needs using these different levels of formalism. Of course, the more formal the specification the more difficult it is to read by a human being.

In the following the term specification will be used both for the required and for the actual behaviour representation.

D.2.1.1 *SDL is based on an Extended Finite-State Machine model*

In the application of SDL, the system to be specified is represented by a number of interconnected abstract machines. A complete specification requires:

1) the specification of the structure of the system in terms of the machines and their interconnections;

2) the dynamic behaviour of each machine in terms of its interactions with the other machines and the environment; and

3) the operations on the data associated with the interactions.

The dynamic behaviour is described with the aid of models which define the mechanisms for the operation of the abstract machines and the communication between machines. The abstract machine used in SDL is an extension of the deterministic Finite-State Machine (FSM). The FSM has a finite internal state memory and operates with a discrete and finite set of inputs and outputs. For each combination of input and state, the memory defines an output and the next state. Transitions from one state to another are usually regarded as taking zero time.

A limitation of the FSM is that all information which needs to be stored must be represented as explicit states. Although it is possible to represent most systems in this way, it will not always be practical. There may be many values to store which are significant for the future behaviour but do not contribute greatly to the overall understanding of the system. This information should not be a part of the explicit state space as it will clutter the presentation. For such applications, the FSM may be extended with auxiliary storage and auxiliary operations on that storage. Address information and sequence numbers are examples of information suitable for storage in auxiliary memory.

The SDL Recommendations define two auxiliary operations which may be included in transitions of the Extended Finite-State Machine (EFSM), i.e., decisions and tasks. "Decisions" inspect parameters associated with inputs and information in auxiliary memory when such information is important for the sequencing of the main machine. "Tasks" perform functions such as counting, operating on auxiliary memory, and manipulating input and output parameters.

In SDL, the interactions between the machines are represented by signals, i.e., the EFSM's receive signals as input and generate signals as output. The signals are composed of a unique signal identifier and optionally a set of parameters. SDL allows for the possibility of non-zero transition time, and defines a first-in first-out conceptual queueing mechanism for signals which arrive at a machine while it is executing a transition. Signals are considered one at a time in order of their arrival.

D.2.2 *Syntax forms of SDL*

SDL is a language which has two different forms, both based on the same semantic model. One is called SDL/GR (SDL Graphical Representation) and is based on a set of standardized graphical symbols. The other is called SDL/PR (SDL textual Phrase Representation) and is based on program-like statements. Both represent the same SDL concepts.

A graphical language has the advantage that it clearly shows the structure of a system and allows the control flow to be easily visualized for human beings. The textual phrase representation is best suited for machine use.

As a design tool SDL should be in a form which allows the user to clearly and concisely express his ideas. SDL/GR allows this and is more in line with the traditional representation of extended finite state machines.

SDL/GR is the original form of SDL. It was devised during 1973 to 1976 and first appeared in the 1976 version of the Z.100-series of Recommendations.

SDL/GR was derived from the graphical languages developed by different organizations for their own use.

The textual phrase representation of SDL, SDL/PR, was devised during the 1977-1980 study period but some refinement was needed before it should be recommended. This refinement was accomplished in the following study period and from 1984 SDL/PR has been one of the recommended concrete syntaxes of SDL.

At first, SDL/PR was intended to be used as an easy way to input SDL documents into a machine, since the GR is more difficult to input. (Graphical peripherals are needed to handle it.) For this reason, emphasis was on a one-to-one mapping between PR and GR. The evolution in graphics terminals (increase of capabilities and decrease of cost) has since led to the acceptability of the GR as suitable for input into a machine. This does not diminish the importance and the use of the PR because some users find it more to their liking, especially those working with programming languages.

This evolution led to a looser correlation between GR and PR, however we can still (easily) map one onto the other, but each form has its own peculiarities. At first glance, the PR strongly resembles a programming language (see Figure D-2.2.1).

```
                –
                –
STATE  aw_off_hook;
INPUT  off_hook;
TASK  'activate charging';
TASK  'connect';
OUTPUT  reset_timer;
NEXTSTATE  conversation;
                –
                –
```

FIGURE D-2.2.1

Example of SDL/PR

In fact, it all depends on what characterizes a text as a programming language.

If we assume a program is defined as "information interpretable by a machine", then not only PR but also GR are "programs".

There are however some differences between an SDL specification and a real program. First it is not essential that an SDL specification should be executable by a machine (though it is not forbidden); what is essential is its capability of conveying precise information from one human being to another human being.

If we look at an SDL specification as a program, what may be considered as "wrong SDL specification" (because of incomplete informal text), might be perfectly valid SDL if considered as a representation of the functional requirements of a system.

A further difference is in the "style" of a SDL specification, compared with the usual representation of a program.

Since SDL is intended to aid communication between human beings, care has been taken to allow different layouts so that the SDL layout can be used to guide the reader to focus on certain aspects which are considered more important than others. This is, of course, unimportant to a program which is supposed to be interpreted by a machine. The machine does not focus on any particular aspect, but has to consider the whole equally, nor is the machine trying to "understand" the program.

For its similarity to a program, the PR is preferred by some programmers who will probably use CHILL to implement the requirements. There is therefore a strong temptation to find a one-to-one mapping from PR to CHILL, so that the requirements expressed in PR can *be* automatically transformed into CHILL code. The reverse is also of interest because it would allow the derivation of a PR specification from a CHILL program.

In § D.9, possible ways of mapping SDL to CHILL are illustrated.

D.2.3 *Applicability of SDL*

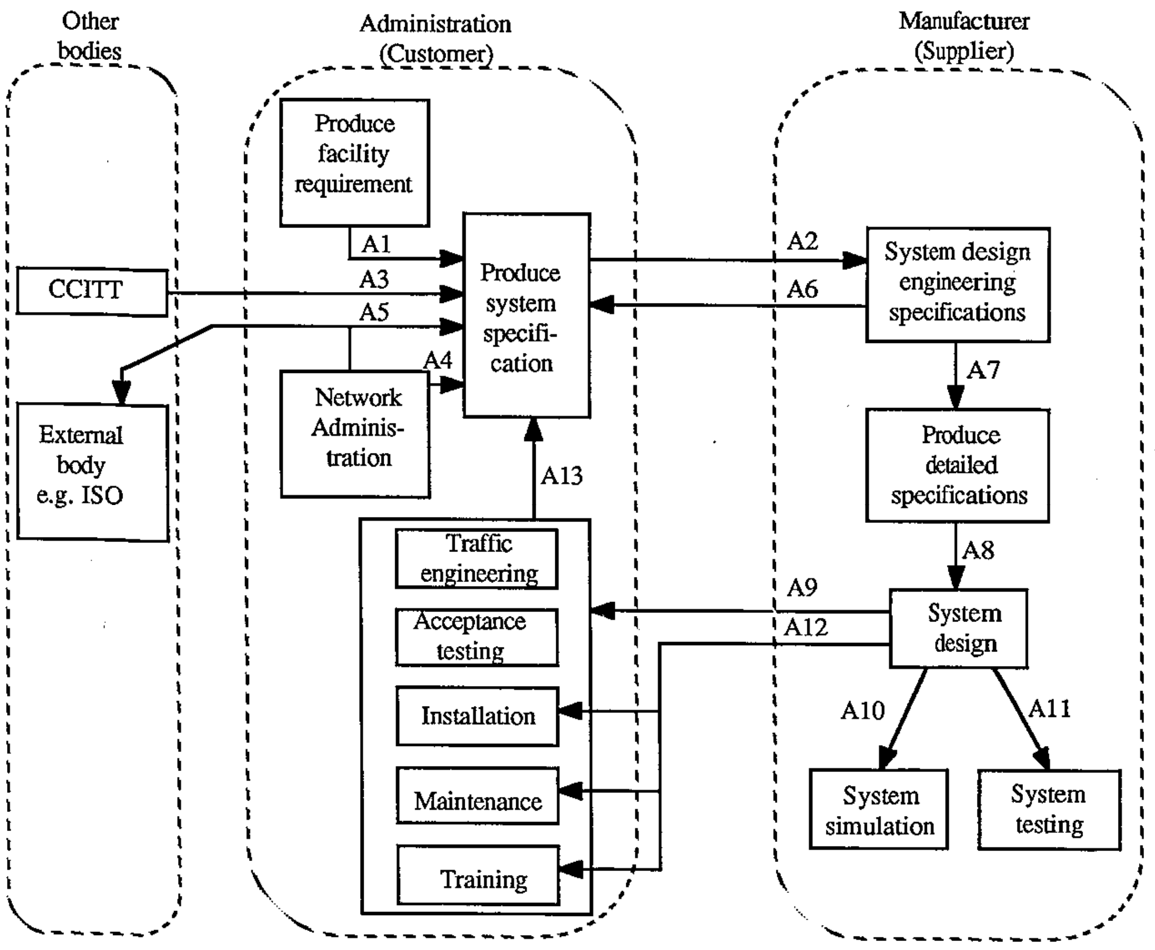
Figure D-2.3.1. shows a range of possible uses of SDL, in the context of the purchase and supply of telecommunications switching systems.

In this figure, the rectangles illustrate typical functional groups, whose precise names may vary from organization to organization, but whose activities would be typical of many Administrations and manufacturers. Each of the directed lines (flow lines) represents a set of documents passing from one functional group to another; SDL can be used as part of each of these sets of documents. The figure is intended to be merely illustrative and is neither definitive nor exhaustive.

The areas of applications are the ones effectively modelled by communicating Extended Finite-State Machines, e.g. functions for telephone, telex, data switching, signalling systems (e.g. Signalling System No. 7), interworking of signalling systems and data protocols, user interfaces (MML).

When particularly considering SPC switching systems, examples of functions which can be documented using SDL are: call processing (e.g. call handling, routing, signalling, metering, etc.), maintenance and fault treatment (e.g. alarms, automatic fault clearing, system configuration, routine tests, etc.), system control (e.g. overload control) and man-machine interfaces.
Application examples of SDL can be found in § D.10.

Specification of protocols using SDL is dealt with in CCITT X-series Recommendations.



A1 An implementation-independent and network-independent specification of a facility or feature
A2 An implementation-independent but network-dependent system specification, including a description of the system environment
A3 CCITT Recommendation and guidelines
A4 Contributions to the system specification, showing the network administration and operational requirements
A5 Other Recommendations that are relevant
A6 Description of an implementation proposal
A7 A project specification
A8 A detailed design specification
A9 A complete system description
A10 Appropriate system and environment description documentation for system simulation
A11 Appropriate system and environment description documentation for system testing
A12 Installation and operation manuals
A13 Contributions to the system specification from specialized functional groups within the Administration
Note 1 - Iteration is possible at all levels.
Note 2 - In some circumstances, SDL documentation that is here shown as being internal to one organization, e.g. A1,A7,A8, could be supplied to another organization.

FIGURE D-2.3.1

General scenario for the use of SDL

D.3 *Basic concepts of SDL*

D.3.1 *System*

As has been stated earlier, SDL models systems. Thus the system is what an SDL specification is defining. As such, an SDL system may model a part of a telephone system (or exchange) or a complete network of telephone systems or parts of a multiplicity of telephone exchanges (e.g. the trunk controllers at both ends of a trunk). The key thing is that from an SDL point of view, the SDL system contains everything the specification is trying to define. Implementation features such as duplicated systems or multiple equivalent nodes cannot be explicitly modelled by SDL. The environment is outside the specification and is not defined in SDL.

The system interfaces with the environment through channels. In theory, only a single bidirectional channel is required to interface with the environment. In practise, channels are usually defined for each logical interface to the environment.

Each system is composed of a number of blocks connected together by channels. Each block in the system is independent from every other block. Each block may contain one or more processes which describe the behaviour of the block. The only means of communication between processes in two different blocks is by sending signals that are transported by the channels. The criteria leading to a certain division of the system into blocks may be to define parts of a manageable size, to create a correspondance with actual software/hardware division, to follow natural functional subdivisions, to minimize interactions, and others.

For large SDL systems there are some SDL constructs that allow the specification of substructures of the parts of a system, so that starting from a broad overview of the system, more and more details can be provided. In this case we say that the system is represented at different levels of details. Those constructs are explained in § D.4.

At the first level of detail the SDL specification of a system describes the structure of the system and includes the following items that will be explained in the following paragraphs:

- System name.
- Signal definitions: the specification of the types of signals interchanged between the blocks of the system or between the blocks and the environment. It includes the specification of the types of values conveyed by the signals (sort list).
- Signal list definitions: the specification of identifiers grouping several signals and/or other signal lists together. Such identifiers can be used to save space and to allow a clearer specification.
- Channel definitions: the specification of the channels connecting the blocks of the system to one another and to the environment. A channel definition includes the specification of the identifiers of signals transported by that channel.
- Data definitions: the specification of user defined newtypes, syntypes and generators visible in all the blocks.
- Block definitions: the specification of the blocks into which the system is divided.
- Macro definitions: guidelines on the use of macros are given in § D.S.I.

According to the SDL Reccomendation, there exist predefined data types that are available to every system. They do not need a definition and can be used by means of their predefined names, i.e.: INTEGER, REAL, CHARACTER, STRING, CHARSTRING, BOOLEAN, PID, TIME, DURATION. The predefined data types are visible at any level in the system definition; they can be considered implicitly defined in a system library accessible at any point in the specification.

Sort names used in signal definitions at system level, must be introduced by partial type definitions visible at system level, i.e., predefined data types or user defined newtypes or syntypes defined at this level.

Further explanations on the use of data types can be found in § D.3.10 and § D.6.

The SDL/PR for a system definition consists of a set of statements, each terminated by a ";" (semicolon). The definition of a system structure begins with the statement "SYSTEM name;" and ends with the statement "ENDSYSTEM name;". The name in the closing statement is optional but, if given, it must be the same name given after the keyword SYSTEM. It is suggested to put the name in the closing statement always, as it increases the readability of the document.

The schema of the SDL/PR of a system structure definition is shown in Figure D-3.1.1.

```
SYSTEM . . . ;
      . . . signal definitions . . .
      . . . signal list definitions . . .
      . . . channel definitions . . .
      . . . data definitions . . .
      . . . block definitions . . .
      . . . macro definitions . . .
ENDSYSTEM . . . ;
```

FIGURE D-3.1.1

Schema of the SDL/PR of the system structure definition

In order to have a clearer and simpler representation of the system structure or also to allow a top-down system specification, a general referencing mechanism is provided in SDL. At this level the referencing mechanism can be applied to block definitions. This feature of the language allows the user to specify just the block name within the system structure definition; the actual block definition can be given separately. (See Figure D-3.1.2).

```
SYSTEM  s;
      . . .
      BLOCK  b1  REFERENCED;
      BLOCK  b2  REFERENCED;
      . . .
ENDSYSTEM  s;
```

FIGURE D-3.1.2

Example of block definition referencing

The referencing mechanism is especially used in SDL/GR as most of the diagrams have to be contained in a single page and often there is not room enough for nested graphical specifications.

Examples of the SDL/GR for a system definition can be found in § D.3.6. D.3.2 *Blocks*

D.3.2 *Blocks*

Within a block, processes can communicate with one another either by signals or shared values. Thus the block provides not only a convenient mechanism for grouping processes, but also, a boundary for the visibility of data. For this reason, care should be taken when defining blocks to ensure that the grouping of processes within a block is a reasonable functional grouping. In most cases it is useful to break the system (or block) into functional units first and then define the processes that go into the block.

Within a block it is possible (optionally) to define communication paths between the processes or between the processes and the environment of the block (i.e. the block boundary). Such communication paths are called signal routes.

For large SDL system it is possible to describe the substructure of a block in terms of other blocks and channels as if the block were a system itself. Such a mechanism is explained in D.4.

The definition of the structure of a block may include the following items:

- Block name
- The definition of the structure of a block may include the following items: Block name.
- Signal definitions: the specification of the types of signals interchanged internally to the block. It includes the specification of the types of values conveyed by the signals (sort list).
- Signal list definitions: the specifications of identifiers corresponding to lists of signals and/or other identifiers of signal lists. Such identifiers, grouping several signals together, can be used to save space and to allow a clearer specification.

- Signal route definitions: the specifications of the communication paths connecting the processes of the block to one another and to the environment of the block. A signal route definition includes the specification of the identifiers of signals transported by that signal route.
- Channel to route connections: the specifications of the connections between the channels external to the block and the signal routes internal to the block.
- Process definitions: the specification of the process types which describe the behaviour of the block. If the block is not described in terms of its substructure there must be at least one process type definition within the block. For the process definition a referencing mechanism is provided similarly to the case of blocks stated in § D.3.1.
- Data definitions: the specification of user defined newtypes, syntypes and generators visible in all defined processes of the block and/or in the block substructure.
- Macro definitions: guidelines on the use of macros are given in § D.5.1.

If there is a substructure of the block some of the items above are optional. (Refer to § D.4 for explanations on structuring).

In a block the following types are visible:

- predefined data types,
- user defined data types defined in the block itself.
- user defined data types visible in the parent block (in case of block partitioning),

In SDLIPR the keywords BLOCK and ENDBLOCK are used to circumscribe a block definition. Examples of the SDL/GR for a block definition can be found in § D.3.6.

D.3.3 *Channels*

Channels are the means of communication between different blocks of the system or between blocks and the environment. A channel may connect one block to another block or one block to the environment in one direction (unidirectional channel) or in both directions (bidirectional channel). Normally, a channel is a functional entity that may be used to denote specific communication paths. In fact, through the partitioning of channels (described in § D.4.5), it is possible to formally specify the behaviour of each channel.

For each denoted direction, or communication path, the channel specification contains a list of all the signals identifiers that can be carried by the channel in that direction. This signal list is provided as a means of ensuring that every signal sent by a process at one end of the channel can be received by the process in the block at the other end of the channel. Thus the channel specification becomes part of the interface specification for each block. In large multi-person projects, early agreement on the signals in a channel and on the specification of those signals reduces the probability that two processes will not be able to communicate as intended with one another.

The definition of a channel includes the following items:

- Channel name.
- One or two communication paths: a communication path specifies the origin and the destination of a list of signals. Blocks identifiers or the keyword "ENV" (for the environment) can be used in this context.
- One or two lists of signals: for each communication path a list of signal. transported in that direction must be specified. The list can include signal identifiers or also identifiers of other lists.
- An optional channel substructure definition (or the reference to it): refer to § D.4.5.

In SDLIPR a channel definition is enclosed between the keywords CHANNEL and ENDCHANNEL. The keywords FROM and TO are used to express the communication paths and the keyword WITH to express signal lists. In Figure D-3.3.1 there is an example of channel definitions in SDLIPR.

In SDL/GR a channel definition is represented by means of a line connecting the two parts involved in the communication. The channel name must be closer to the line than any other symbol. The paths are represented by means of arrows and the lists of signals must be enclosed in square brackets as illustrated in the examples in Figure D-3.3.2. The arrows must not be placed at any of the two endpoints of the line, in order not to confuse channels with signal routes (see § D.3.5).

In a bidirectional channel each of the two lists of signals must be closest to the corresponding arrow.

```
SYSTEM so_and_so;
     . . .
     SIGNALLIST s_list_id_1 = sig_b, sig_c, sig_d;
     . . .
     CHANNEL c1
          FROM block_a TO block_b WITH signal_1,signal_2,signal_3;
     ENDCHANNEL c1;

     CHANNEL chan_2
          FROM ENV TO block_c WITH ext_sig_1,ext_sig_2;
          FROM block_c TO ENV WITH int_sig_1;
     ENDCHANNEL chan_2;

     CHANNEL chan.name.3
          FROM bx TO by WITH sig_a,(s_list_id_1),sig_e;
     ENDCHANNEL chan.name.3;
     . . .
ENDSYSTEM so_and_so;
```

FIGURE D-3.3.1

Example of channel definitions in SDL/PR

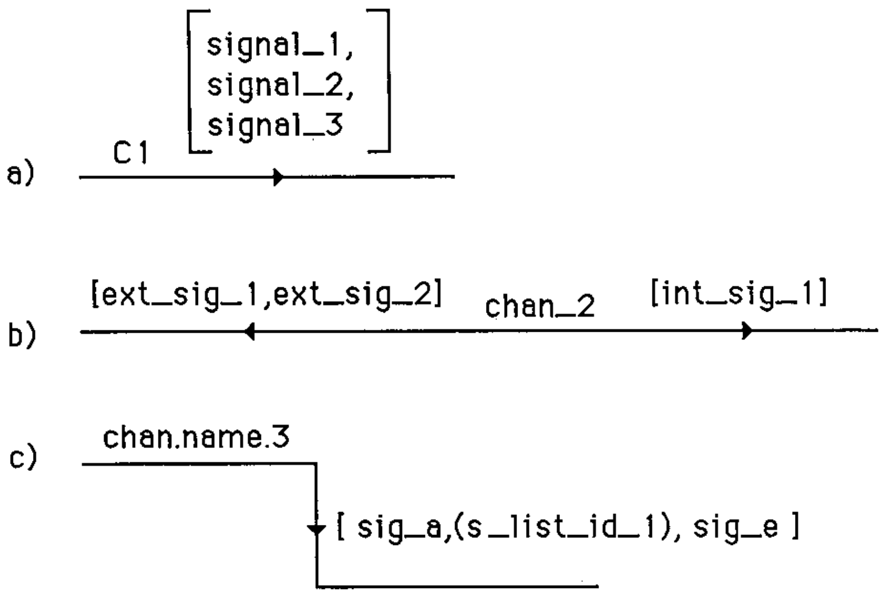


FIGURE D-3.3.2

Example of channel definitions in SDL/GR

D.3.4 *Signals*

Signals can be defined at system level, block level, or in the internal part of process definition. Signals defined at a certain level can be used at that level or also at the lower levels, however , in order to simplify each level, it is suggested to define the signals as local as possible. Signals defined within a process definition can be interchanged between instances of the same process type (ref. § D.3.8) or between services in the process (§ D.5.3).

The definition of a signal includes the following items:

- Signal name
- Sort list (optional): represents the list of the value types conveyed by that signal.
- Signal refinement (optional): explained in § D.4.7.

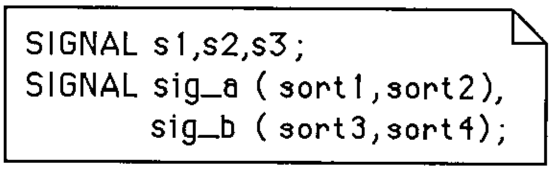
In SDL/PR a signal definition is specified by the keyword SIGNAL. Several signals (not refined) may be defined inside the construct, providing the signal name and the sort list. Examples of signal definitions in SDL/PR are given in Figure D-3.4.1.

```
SIGNAL  s1,s2,s3;

SIGNAL  sig_a (sort1,sort2),
        sig_b (sort3,sort4);
```

FIGURE D-3.4.1

Examples of signal definitions in SDL/PR

In SDL/GR a signal definition is specified by enclosing linear statements in a text symbol as shown in Figure D-3.4.2.



```
SIGNAL s1,s2,s3;
SIGNAL sig_a ( sort1,sort2),
       sig_b ( sort3,sort4);
```

FIGURE D-3.4.2

Examples of signal definitions in SDL/GR

D.3.5 *Signal routes*

Signal routes are used to express communication paths similarly to the channels. They can be used at block level and also at process level. Like channels signal routes can be unidirectional or bidirectional but they cannot be partitioned.

At block level they represent a means of communication between the processes of a block or between the processes and the environment of the block, that is a channel leading to or from that block.

At process level signal routes can be used when the process is substructured into services (Ref. § D.5.3). In this case they connect services one to another or to the signal routes of the process.

When a signal is delivered to a signal route leading to the block boundary, the signal is given to the channel connected to the signal route. When a signal arrives at the block from a channel and the channel is connected to one or more signal routes, the signal is delivered to the signal route that is able to convey that signal.

In SDL/GR the definition of a signal route begins with the keyword SIGNALROUTE. The syntax of communication paths and signal list is the same as in the case of channels.

In SDL/GR the only difference between a signal route and a channel is that for the signal routes the arrows must be at endpoints of the lines; close to each arrow there must be the proper signal list. Examples on signal routes can be found in Figures D-3.6.3 and D-3.6.5 in the following paragraphs.

D.3.6 *System and block diagrams*

In SDL/GR a system definition is represented by means of a set of diagrams. The structure of a system in terms of channels and blocks is represented by means of the system diagram.

The system diagram is composed of:

- The frame symbol: a rectangular shaped symbol enclosing all the other symbols. It represents the system boundary: outside this frame there is the system environment.
- The system heading: the keyword SYSTEM followed by the system name (put in the upper left corner of the frame).
- An optional page numbering (put in the upper right corner of the frame).
- Text symbols: such a symbol can enclose linear statements. It is generally used to show in the diagram the definitions of signals, signal lists and data.
- The block interaction area: this includes the specification of the blocks of the system, the channels and the lists of signals transported by the channels.
- Macro diagrams: guidelines on the use of macros are given in D.5.1.

In the system diagram the specification of a block can be one of the two following items:

- A block reference: A block symbol containing only the block name.
- A block diagram: A frame containing the specification of the block structure in terms of its processes and their interactions. If a block is sub-structrured in sub-blocks, the specification of the substructure or the reference to it must be provided inside the block frame (§ D.4.3).

The shape of the symbols used in system and block diagrams can be found in the SDLIGR summary. For symbol dimensions refer to § D.7.1.4.

In Figure D-3.6.1 there is an example of a system diagram for the system "s". In this example the system s is divided into two blocks B 1 and B2 connected to one another and to the environment by the channels Cl, C2, C3 and C4. For the blocks B1 and B2 just a reference is given in this example. Further explanations on channels and signal lists symbols are given in the following paragraphs.

The same example in SDL/PR is shown in Figure D-3.6.2.

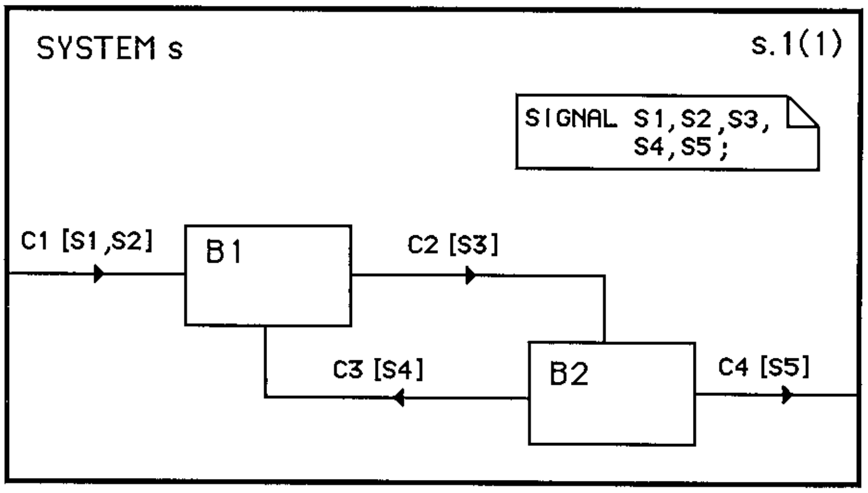


FIGURE D-3.6.1

Example of system diagram

```
SYSTEM s;
    SIGNAL  S1,S2,S3,S4,S5;
    CHANNEL C1 FROM ENV TO B1 WITH S1,S2;
    ENDCHANNEL C1;
    CHANNEL C2 FROM B1 TO B2 WITH S3;
    ENDCHANNEL C2;
    CHANNEL C3 FROM B2 TO B1 WITH S4;
    ENDCHANNEL C3;
    CHANNEL C4 FROM B2 TO ENV WITH S5;
    ENDCHANNEL C4;
    BLOCK B1 REFERENCED;
    BLOCK B2 REFERENCED;
ENDSYSTEM s;
```

FIGURE D-3.6.2

Example of a system structure definition in SDL/PR

The structure of a block in terms of processes and signal routes is represented in SDLIGR by means of the block diagram.

The block diagram is composed of:

- The frame symbol: a rectangular shaped symbol enclosing all the other symbols. It represents the block boundary: outside this frame there is the block environment.
- The block heading: the keyword BLOCK followed by the block name (put in the upper left corner of the frame).
- An optional page numbering (put in the upper right corner of the frame).
- Text symbols: such a symbol can enclose linear statements. It is generally used to show the definitions of signals, signal lists and data.
- The process interaction area: this includes the specification of the processes of the block and possibly the signal routes and the lists of signals transported by the signal routes. In this area it is also possible to show processes that create other processes; this feature is described in § D.3.8.1.
- Channel identifiers: if the diagram shows signal routes leading to or from the environment of the block, the identifiers of the channels connected to such signal routes must be specified outside the frame, in correspondence to the lines of signal routes.

– Macro diagrams: guidelines on the use of macros are given in D.5.1. In the block diagram the specification of a process can be alternatively:

In the block diagram the specification of a process can be alternatively:

– A process reference: a process symbol containing the process name and, optionally, the specification of process instances. Such a specification of process instances consists of a pair of integers separated by a comma and enclosed within round brackets. (Refer to § D.3.8).
– A process diagram: a frame containing a graph of connected symbols describing the process behaviour in terms of states, inputs, outputs, actions etc. (Refer to § D.3.8). If the process is substructured in services the process diagram contains the service interaction area (§ D.5.3).

If there is a substructure of the block some of the items above are optional. (Refer to § D.4 for explanations on structuring).

In Figure D-3.6.3 there is an example of a block diagram for the block "B 1" introduced in the example of Figure D-3.6.1. The Block B1 is described in terms of the two processes P1 and P2, connected by the signal routes R1, R2, R3, R4 and R5. For the processes P1 and P2 just the references are given in this example. Outside the frame the identifiers of channels C1, C2 and C3 are specified as well.

The same example in SDL/PR is shown in Figure D-3.6.4.

As it has been stated before, block diagrams can be included within the system diagram instead of having references. For example in Figure D-3.6.5 the diagrams of Figure D-3.6.1 and D-3.6.3 are merged into a single system diagram.

A general criteria for drawing such diagrams is that the diagrams should not be too complex in order to be readable and should be contained in one page.

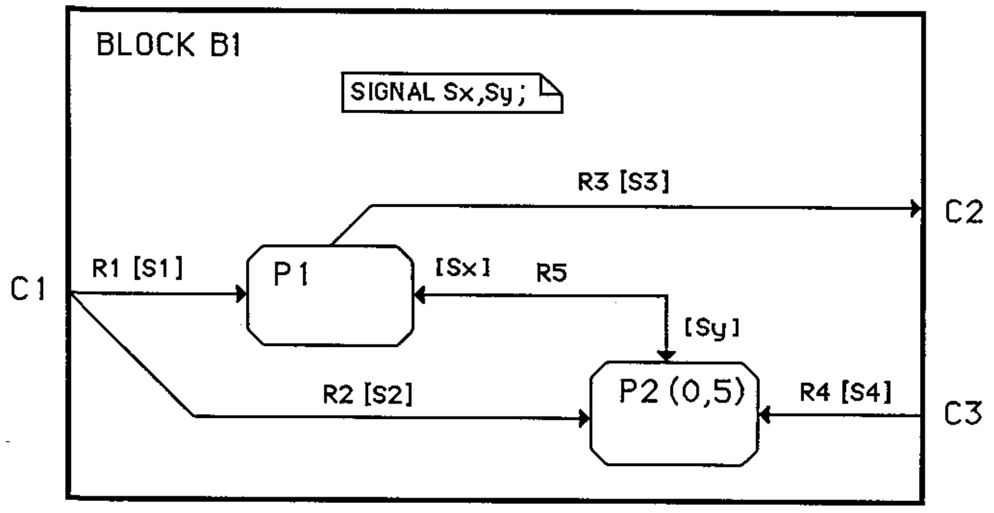


FIGURE D-3.6.3

Example of block diagram

```
BLOCK B1;

     SIGNAL Sx,Sy;

     SIGNALROUTE R1 FROM ENV TO P1 WITH S1;
     SIGNALROUTE R2 FROM ENV TO P2 WITH S2;
     SIGNALROUTE R3 FROM P1 TO ENV WITH S3;
     SIGNALROUTE R4 FROM ENV TO P2 WITH S4;
     SIGNALROUTE R5 FROM P1 TO P2 WITH Sy;
                      FROM P2 TO P1 WITH Sx;

     CONNECT C1 AND R1,R2;
     CONNECT C2 AND R3;
     CONNECT C3 AND R4;

     PROCESS P1 REFERENCED;
     PROCESS P2 REFERENCED;

ENDBLOCK B1;
```

FIGURE D-3.6.4

Example of a block structure definition in SDL/PR

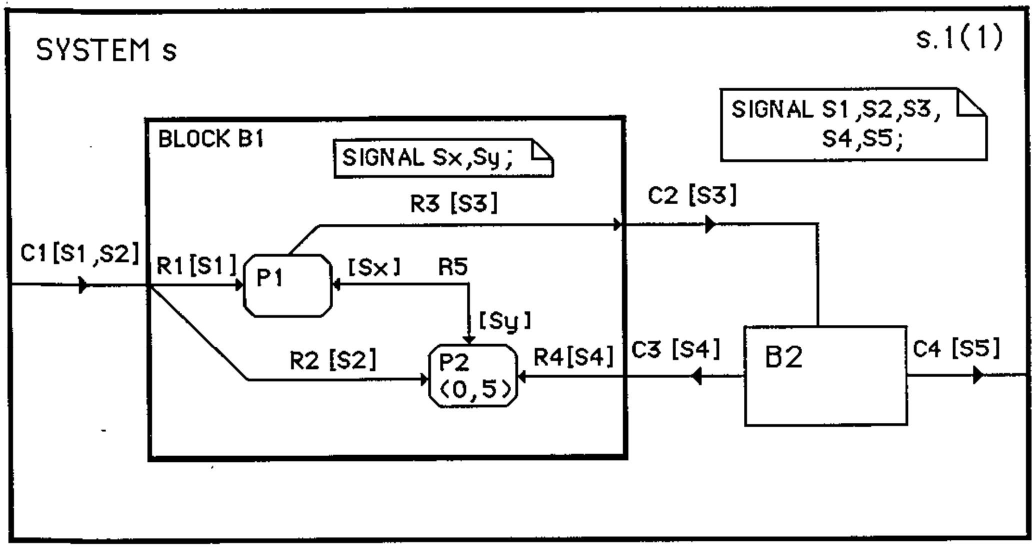


FIGURE D-3.6.5

Example of a system diagram including a block diagram

D.3.7 *Comments and text extension*

D.3.7.1 *Comments*

Comments may be added to an SDL specification to help the reader and to clarify some parts. In SDL there are two types of comments that have been introduced to suit the SDL/PR and the SDL/GR.

The first type is called "note" and is especially used in SDL/PR. It is delimited by "/*" at the beginning and "*/" at the end.

In SDL/PR such a comment can be inserted wherever a space may occur. The comment must not contain the special sequence "*/".

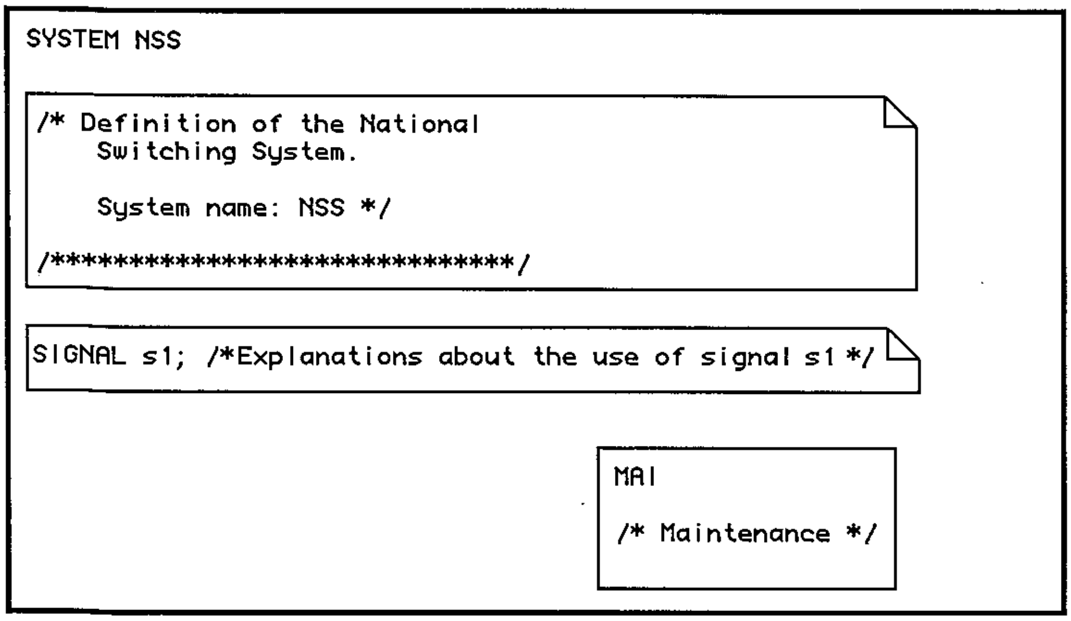
In SDL/GR such a comment may take place wherever a space may occur within linear statements.

In Figure D-3.7.1 and D-3.7.2 some examples of this form of comment are shown both in SDL/PR and SDL/GR.

```
SYSTEM NSS;

/* Definition of the National
   Switching System.

   System name: NSS */

/******************************/

   . . .
   SIGNAL s1; /* Explanations about the use of signal s1 */
   . . .
   BLOCK MAI REFERENCED; /* Maintenance */
   . . .
ENDSYSTEM NSS;
```

FIGURE D-3.7.1

Examples of the first form of comment in SDL/PR



```
SYSTEM NSS

/* Definition of the National
   Switching System.

   System name: NSS */

/********************************/

SIGNAL s1; /*Explanations about the use of signal s1 */

MAI

/* Maintenance */
```

FIGURE D-3.7.2

Examples of the first form of comments in SDL/GR

The second form of comment allows a one to one mapping between SDL/PR and SDL/GR and is more suitable for applications in which automatic translations are made.

In SDL/PR such a comment consists of the keyword COMMENT followed by a character string; it may be inserted, as a statement (i.e. followed by a ";"),wherever a task statement can be inserted. In addition it can be inserted before the symbol ";" (semicolon) at the end of any statement.

In SDL/GR this form of comment is represented by means of a comment symbol containing the comment text. The comment symbol is a rectangular shaped symbol with the left or right side missing. The symbol must be stretched both horizontally and vertically to contain all the text. It may be connected to any SDL/GR symbol or flow line. A dashed line must be used for the connection. If the association between the comment text and the comment symbol is not ambiguos, the comment symbol can be drawn simply as a square bracket.

In Figure D-3.7.3 and D-3.7.4 some examples of this form of comment are shown both in SDL/GR and SDL/GR.

```
SYSTEM s COMMENT 'comment about the system';
    CHANNEL C2
        FROM B1 TO B2
        WITH s3          COMMENT 'comment on the channel';
    ENDCHANNEL C2;
    BLOCK B1
    COMMENT 'comment about the block';
        . . .
        PROCESS p1;
            . . .
            TASK 't1';
            TASK 't2';
            . . .
        ENDPROCESS p1;
    ENDBLOCK B1;
ENDSYSTEM s;
```

FIGURE D-3.7.3

Examples of the second form of comment in SDL/PR

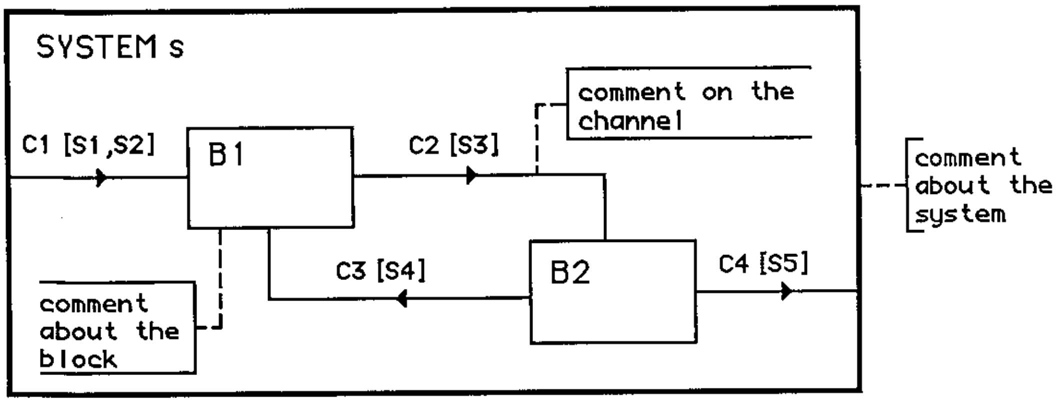
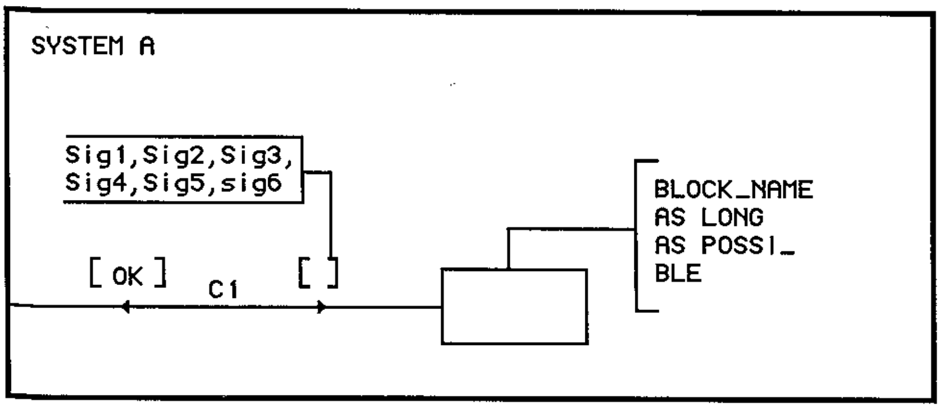


FIGURE D-3.7.4

Example of the second form of comment in SDL/GR

D.3.7.2 *Text extension*

Normally the text associated with a graphic symbol should be placed inside that symbol. However, this is often not possible or practical. An alternative is to place the text in a text extension symbol connected to the associated symbol. The text extension symbol is similar to the comment symbol; the only difference is that the connecting line is solid instead of dashed. See Figure D-3.7.5.



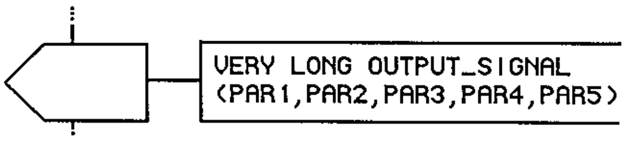


FIGURE D-3.7.5

Examples of the use of text extension symbol

An underline character ("_") can be used at the end of a text line as a continuation character. In this case the remaining spaces on the same line are not considered as part of the text. More detailed guidelines on the syntax of names can be found in § D.3.13.

D.3.8 *Processes*

A process is an extended finite state machine which defines the dynamic behaviour of a system. Processes basically are in a state awaiting signals. When a signal is received, the process responds by performing the specific actions that are specified for each type of signal that the process can receive. Processes contain many different states to allow the process to perform different actions when a signal is received. These states provide the memory of the actions that have occurred previously. After all the actions associated with the receipt of a particular signal have occurred, the next state is entered and the process waits for another signal.

Processes can either exist at the time the system is created or they can be created as the result of a create request from another process. In addition, processes can live forever or they can stop by performing a stop action.

A process definition represents the specification of a type of process; several instances of the same process type may be created and exists at the same time; they can execute independently and concurrently. A process definition consists of the following items (some of which are optional):

- Process name.
- A pair of integers: the first integer specifies the number of process instances created when the system is created, if it is omitted it has a default value of 1; the second integer specifies the maximum number of simultaneous process instances, if omitted the default maximum value is unbounded.
- Formal parameters: a list of variable identifiers associated with their sorts that is used for passing information to the process at creation time. In the process creation request a list of actual parameters can be provided for this purpose. Values of formal parameters of processes created at system creation time are undefined.
- Valid input signal set: a list of signal identifiers defining the signals that can be received by the process.
- Signal definitions: the specification of the signals that can be interchanged between instances of the same process or between services in the process (§ D.5.3).
- Procedure definitions: the specification of the procedures that can be called by the process. A procedure reference can be used in this context. (Procedures are explained in D.3.9).
- Data definition: the specifications of user defined newtypes, syntypes and generators local to the process.

– Variable definition: the declaration of the variables of the process. Optionally a variable can be declared to be shared among other processes of the same block (REVEALED variable) or exported to other processes also in other blocks (EXPORTED variable). For each variable declared the identifier of its sort must be specified. An initial value can optionally be specified.

– View definitions: the declaration of variable identifiers that can be used to obtain the values of variables owned by other process instances. For each variable identifer, the variable sort must be specified.

– Import definitions: the specification of identifiers of variables owned by other processes that the process wants to import. For each identifier, the variable sort must be specified.

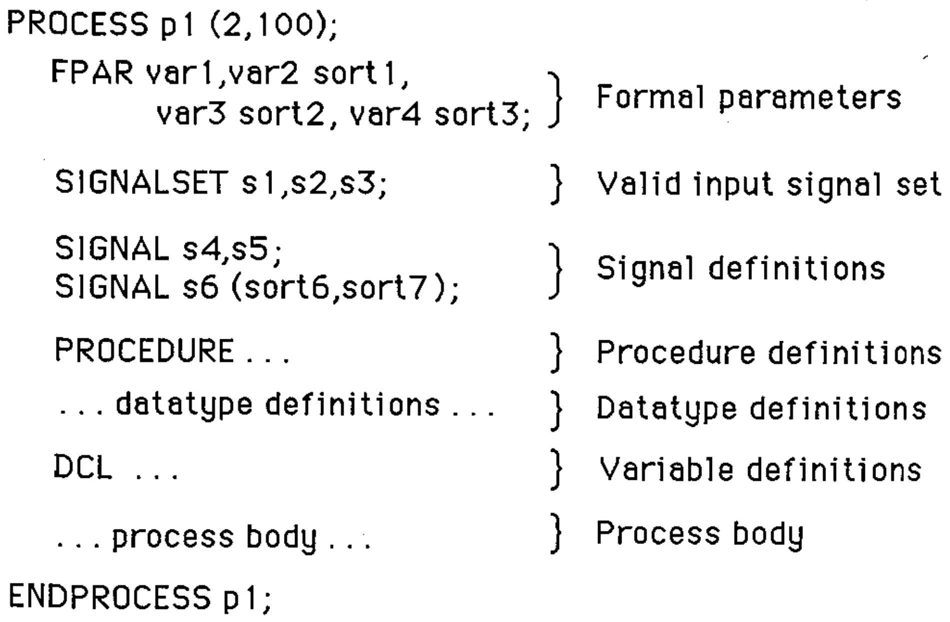
– Timer definition: explained in § D.3.11.

– Macro definitions: guidelines on the use of macros are given in § D.5.1.

– Process body: the specification of the actual behaviour of the process in terms of states, input, output, task, etc. If the process is structured in sub-parts (services), the process definition includes a section of service decomposition instead of the process body. Guidelines on this feature are given in § D.5.3.

Examples and explanations about data, variables, view definitions and import definitions are given in § D.3.10

A partial example of process definition in SDL/PR is given in Figure D-3.8.1 (The keywords of the language are in upper-case letters).



```
PROCESS p1 (2,100);
   FPAR var1,var2 sort1,
         var3 sort2, var4 sort3;  } Formal parameters

   SIGNALSET s1,s2,s3;            } Valid input signal set

   SIGNAL s4,s5;
   SIGNAL s6 (sort6,sort7);       } Signal definitions

   PROCEDURE ...                  } Procedure definitions
   ... datatype definitions ...   } Datatype definitions
   DCL ...                        } Variable definitions

   ... process body ...           } Process body
ENDPROCESS p1;
```

FIGURE D-3.8.1

Partial example of process definition in SDL/PR

The process body represent the actual graph of the finite state machine. It consists of a sequence of ordered statements in SDL/PR and, in SDL/GR, it is a sequence of symbols that are connected by directed arcs (similarly to a flow-chart). The specification of the process body must always begin with a START followed by a set of actions (transition). The interpretation of a process instance begins when the process instance is created.

The possible actions performed in a transition are:

– Task: variable assignment (or informal text).

– Export: variable export.

– Set: request of a timer activation.

– Reset: reset of a timer.

– Output: sending of a signal to another process.

- Create request: creation of an instance of the specified process type.
- Decision: selection of a set of actions depending on a question.
- Procedure call: request of interpretation of a separate, self contained set of actions (intended as in programming languages).
- Join: specification of a "jump" to another set of actions.

A transition may end with one of the following actions:

- Nextstate: specification of the state the process instance will assume.
- Stop: immediate halt of the process instance.

After the specification of the start action and of the optional starting transition, the process body includes the definitions of all the possible states of the process. Each state definition begins with the specification of possible stimuli awaited by the process in that state. The possible stimuli are:

- Inputs: signals that can be received.
- Saves: signals that need to be saved for future processing.
- Enabling conditions: explained in § D.3.8.5.
- Continuous signals: explained in § D.3.8.5.

Corresponding to each stimulus except saves, a transition must be specified. Such a transition represents the sequence of actions the process will perform if that stimulus occurs.

If a process performs a stop action and there are pending signals that were sent to it but not yet received, such signals are discarted.

In SDL/GR a process definition is represented by means of the process diagram. A process diagram consists of the following items:

- A frame symbol: a rectangular shaped symbol which contains all the other symbols. If there are no signal routes connected to the frame symbol it can be omitted.
- The process heading: the keyword PROCESS followed by the process identifier, then by the possible specification of process instances and by the specification of formal parameters. The process heading is put in the upper left corner of the frame.
- An optional page numbering (put in the upper right corner).
- Text symbols: in the case of a process diagram, a text symbol can be used to contain signal, variable, view, import , data and timer definitions.
- Procedure references: a procedure symbol containing a procedure name representing a procedure of the process that is separately defined.
- Procedure diagrams: one for each local procedure of the process which „is not referenced.
- The process graph area: the specification of the process behaviour in terms of start, states, inputs, outputs, tasks, ... and directed arcs. If the process is structured in services the process graph area contains the specification of the services or their references (see § D.5.3). The GR symbols used for the process body can be found in the SDL/GR summary.
- Macro diagrams: guidelines on the use of macros are given in § D.5.1.

In Figure D-3.8.2 there is an example of process definition in SDL/GR; more explanations and examples on process graphs can be found in the following paragraphs.

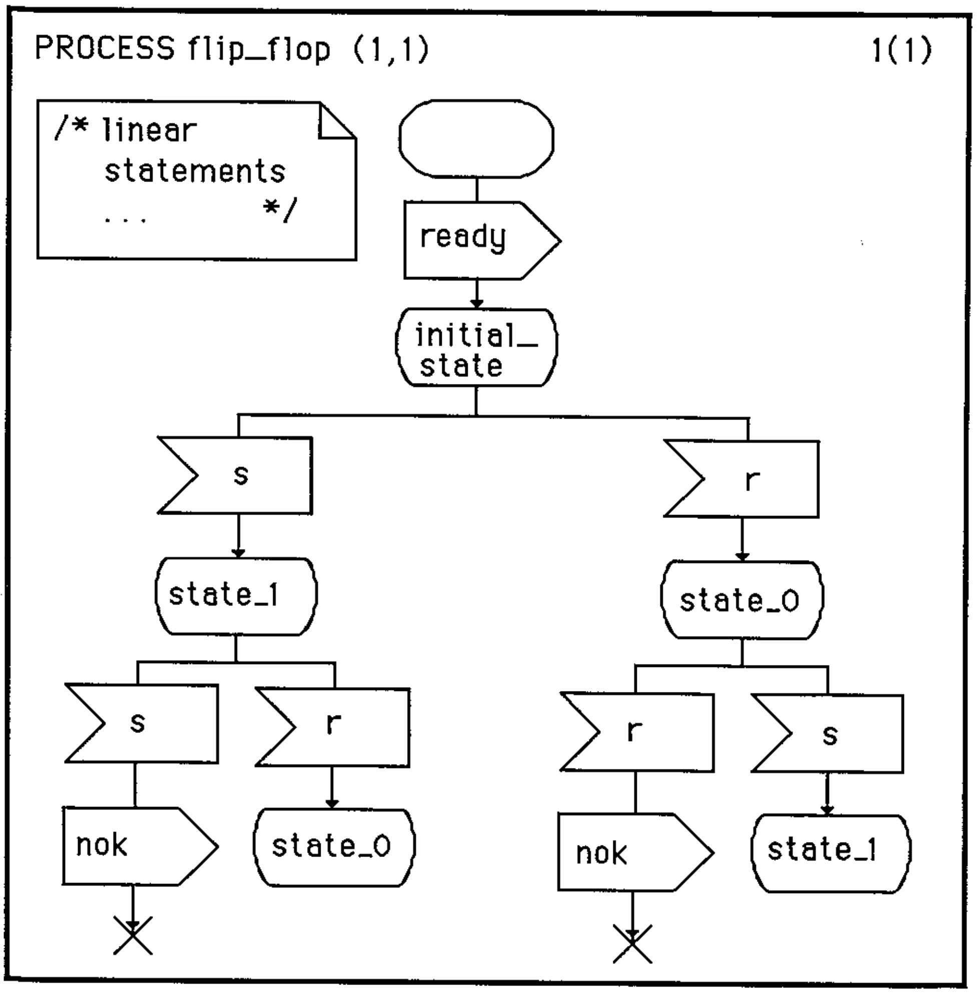


FIGURE D-3.8.2

Example of process diagram

If there is not enough room in a single page for a process graph, the diagram can be drawn in several pages noting that:

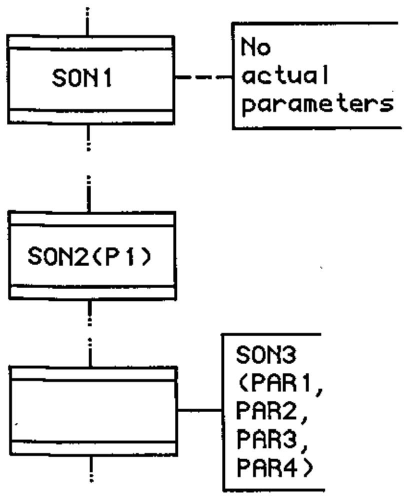
- A page numbering with the number of the page and the total number of the pages should be provided, i.e.: 1 (9).
- The join symbol or the nextstate symbol can be used to represent connections between different parts of the process graph.

A good criteria to divide a process graph in parts is to represent a state definition in each page. If there is no room enough in a page for a single state definition the join symbol can be used to express connections to a piece of graph in another page.

D.3.8.1 *Process creation*

As it has been stated in the previous paragraph, processes (i.e. process instances) can be created as a result of an explicit request or at the system creation.

The explicit create request may be issued only by another process in the same block of the process being created and allows the specification of actual parameters for transferring information to the new instance created. See Figure D-3.8.3 for example of creation in PR and GR form.



```
...
CREATE SON1 COMMENT
                'No actual
                 parameters';
 ...

...
CREATE SON2(P1);
...

...
CREATE SON3 (PAR1,
             PAR2,
             PAR3,
             PAR4);
```

a) GR form   b) PR form

FIGURE D-3.8.3

Examples of create requests in SDL/GR and SDL/PR

If one or more process instances of a given process type are created at system creation time, the definition of that process must not access formal parameters because they will be undefined. Consequently each process definition with formal parameters must either ensure its formal parameters are not accessed before they are given values or explicitly include the specification of zero process instances at system creation time. (see Figure D-3.8.4).

```
PROCESS explicitly_created_proc_1 (0,...);
   FPAR ...
   ...
ENDPROCESS explicitly_created_proc_1;
```

FIGURE D-3.8.4

Definition of a process with specification of formal parameters

When a process is created, process instance values can be determined by means of the following predefined expressions:

OFFSPRING : returns the PId value of the last created instance.

SELF : returns the PId value of the process instance itself.

PARENT : returns the PId value of the creating instance.

SENDER : returns the PId value of the process sending the last signal consumed.

When a process is created as a result of the system creation only the SELF expression returns a PId value, the expressions OFFSPRING and PARENT give the NULL value.

Such values are very important when there are more process instances of the same process type because they are the only way to address signals unequivocally to the instances. In fact, as better explained in § D.3.&6, when a process sends a signal it must specify the destination instance, unless the destination instance is unequivocally determinable.

The user has to be carefull in ensuring that created process instances will be able to communicate to one another if needed. To achieve this, often some kind of initialization processes must be provided. Such processes, created at system creation time, will have to create the other processes and possibly to communicate appropriate PId values to all processes needing to know them.

Other important notes about process creation are:

1) After the system is created, processes can only be created by another process in the same block. One means of allowing creation of processes in other blocks, is to have a special process in every block that will cause the creation of a process when it receives a signal from a process in another block.

2) Once created processes have a lifetime of their own. Processes die only when they perform a stop action during a transition. One way to model systems where external kill operations are allowed, is to have a special kill signal. When the kill signal is received, the process performs a stop action.

The relationship between creating and created processes can be shown in a block diagram by means of create line symbols as shown in Figure D-3.8.5.

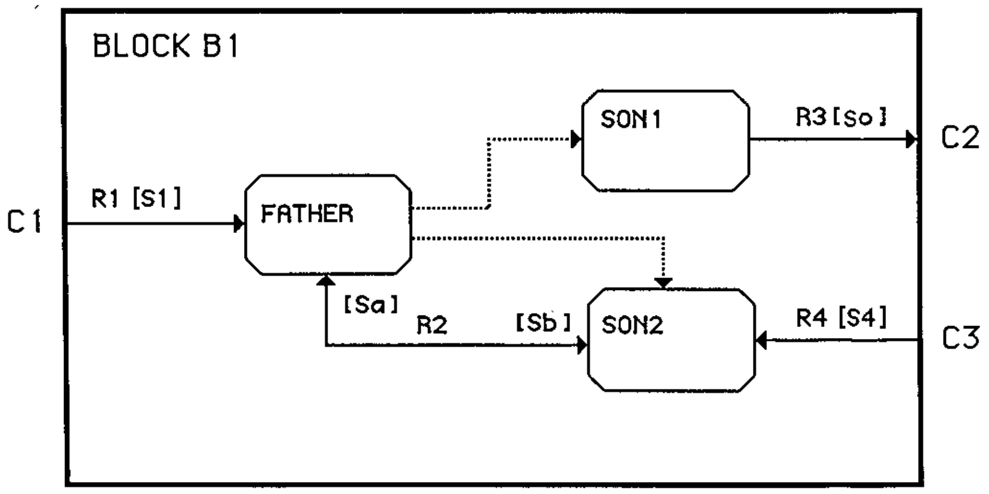


FIGURE D-3.8.5

Example of block diagram with create line symbols

D.3.8.2 *States*

A state is a point in the process where no actions are being performed but where the input queue is monitoring for the arrival of incoming signals. Based on the signal identifier given in the input signal, the arrival of the signal will cause the process to leave the state and perform a specific sequence of actions. A signal which has arrived and caused a transition has been "consumed" and ceases to exist.

In SDL/PR a state is represented by the keyword STATE followed by the state name. The specification of the state ends either at the following state statement or at the process end or by means of the explicit keyword ENDS TATE. An asterisk can be used in the state statement instead of the state name; this is a shorthand notation to indicate that the following inputs or saves and correspondent transitions should be interpreted at every state.

The keyword NEXTSTATE followed by the state name is used to indicate the following state. A dash can be used in the nextstate statement instead of the state name, to represent that the following state is the same state the current transition originated from.

In figure D-3.8.6 there is a partial example in. SDL/PR.

```
SYSTEM s;
   . . .
   BLOCK b;
      . . .
      PROCESS  p;
         . . .
         START;
          . . .
         STATE  st1;
          . . .
         STATE  st2;
            . . .
            NEXTSTATE  st1;
          . . .
         ENDSTATE  st2;

         STATE  st3;
           . . .
      ENDPROCESS  p;
   ENDBLOCK  b;
ENDSYSTEM  s;
```

FIGURE D-3.8.6

Partial example of SDL/PR for state definitions

In SDL/GR a state is represented by means of a state symbol containing the state name and is connected to input symbols or save symbols. The same state symbol with an incoming arrow is used to represent a nextstate statement.

For convenience, to simplify drawing or as an aid to better understanding, the same state may appear a number of times in an SDL diagram. Where this is done, the diagram is considered to be completely equivalent to the diagram which would result from merging all multiple appearances of the same state. Figures D-3.8.7 and D-3.8.8 give examples of this. In Figure D-3.8.7 b) a state symbol is used as a connector to the main state having the same name, when referred to in a nextstate symbol. In Figure D-3.8.8 a state is represented by multiple symbols each with only a subset of the inputs (or saves).

In Figure D-3.8.7 diagrams a) and b) are logically equivalent. Diagram a) contains only a single appearance of each state whilst diagram b) uses multiple appearances. In diagram b) the state has one main appearance where all of its related input (and save) symbols are shown. Where this state can be reached from other points in the diagram (as the terminating point of a transition) it is shown as a state without any associated inputs or saves. A comment referring to the state symbol from the nextstate symbol will improve clarity, particularly when the appearances are on different pages.

Figure D-3.8.8 uses multiple appearances of a state to build up the total set of inputs (and saves). Each occurrence of the state is shown with only a subset of these.

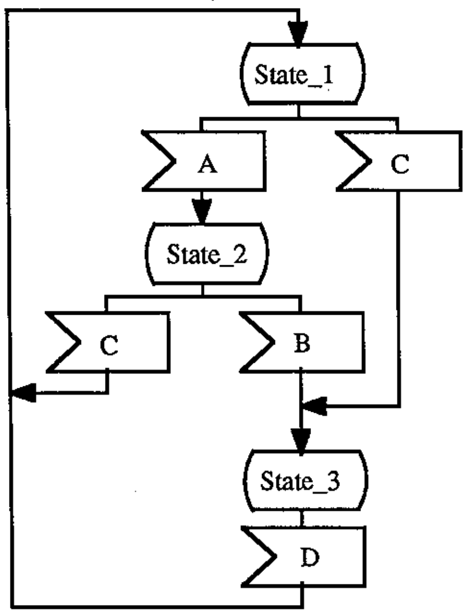


FIGURE D-3.8.7a

Single appearance of a state

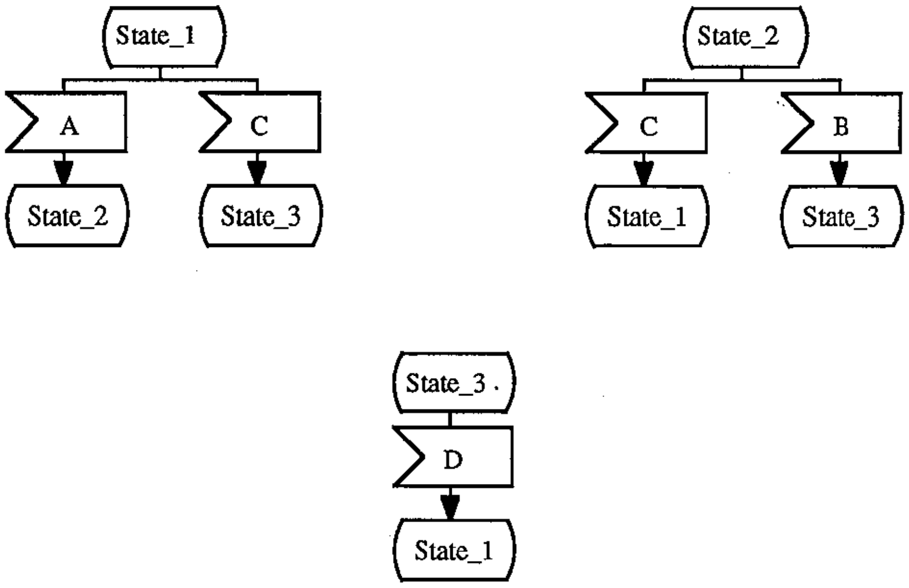


FIGURE D-3.8.7b

Diagram a) with main states and subsequent states used as connectors to the states

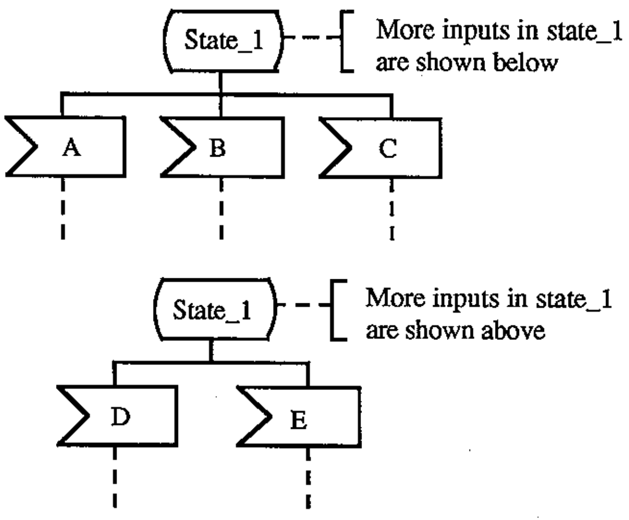

State_1
More inputs in state_1 are shown below
A
B
C
State_1
More inputs in state_1 are shown above
D
E

FIGURE D-3.8.8

Multiple appearance of a state for when all the inputs can not clearly be drawn from the same symbol

This approach has been successfully used where states have a relatively large number of input or saves but may have the danger that readers misinterpret the diagram if they are unaware that there are multiple appearances. To avoid this misunderstanding, the states showing only a subset of the inputs/saves should be annotated with a comment giving reference to the other states with their associated inputs and saves as shown in Figure D-3.8.8.

Multiple appearance may conveniently be used to focus the attention of the reader on certain aspects (e.g. the normal sequence of signals processed) deferring other aspects to other pages (e.g. alarm situation handling).

During a transition, a process does not know explicitly which input signal caused the transition. This can only be inferred from context (i.e. this transition can only occur if a particular signal had been received). In Figure D-3.8.9, task Ti is performed only if 11 is received. However, task T2 will be performed if either 12 or 13 are received. If it is important for T2 to know which input was received, then it is better to design the process as shown in Figure D-3.8.10.

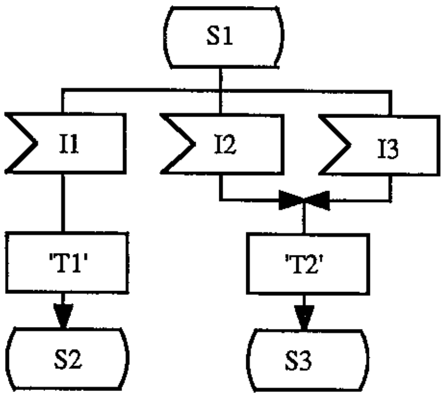


FIGURE D-3.8.9

Performing a task dependent of two out of three received signals but independant of which one

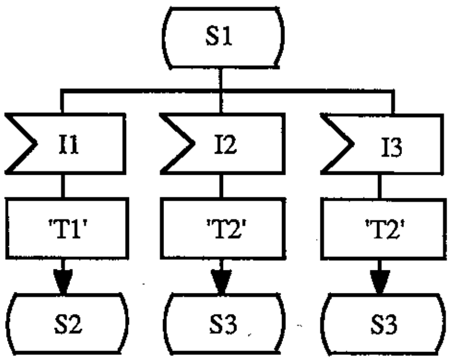


FIGURE D-3.8.10

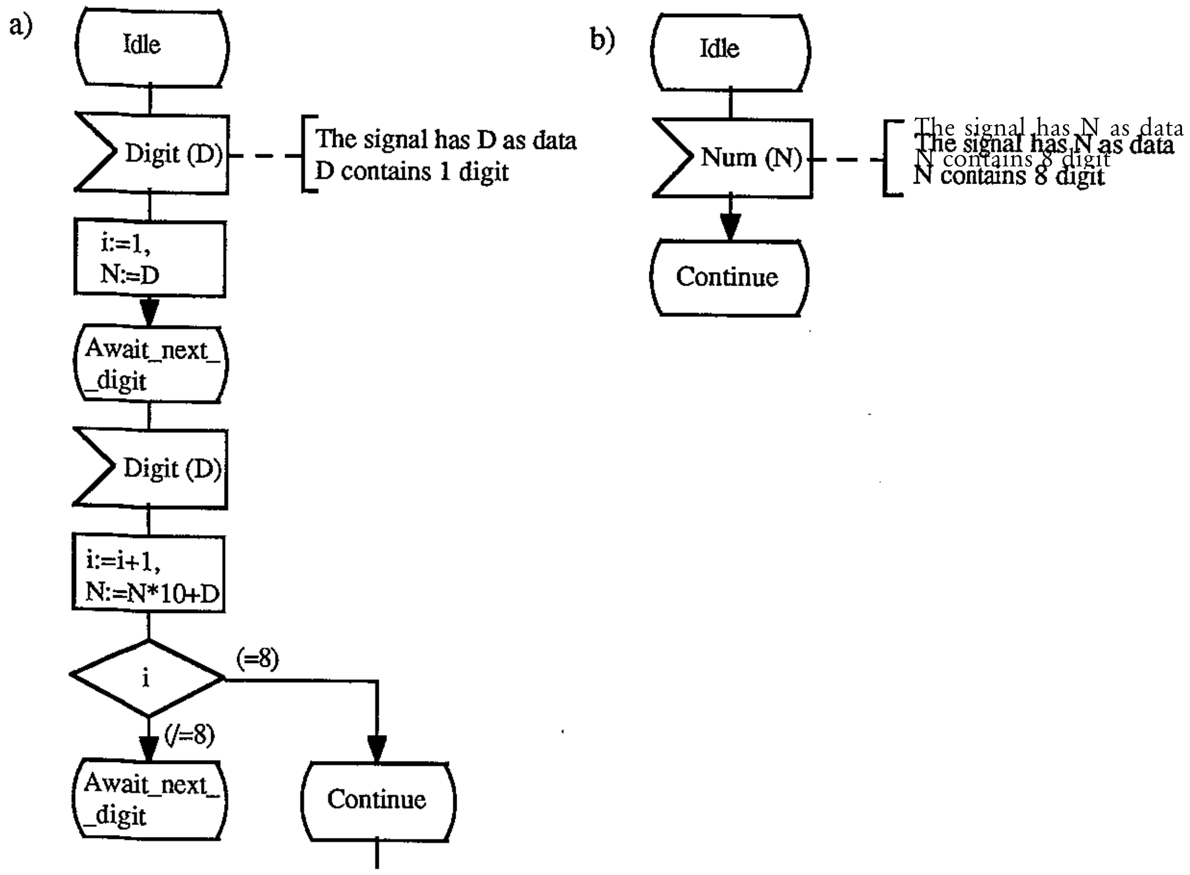
Performing a task dependant on the received signal

D.3.8.2.1 *Determination of the required states*

Usually the author of an SDL diagram has some flexibility available in the approach when defining the states of a process. He may require a strategy enabling him to identify the states of the process and this strategy can be informal or formal. Good judgement (obtained through practice) is required in order to produce SDL diagrams which are neither unnecessarily complicated through the identification of too many distinct states or which fail to exploit the inherent advantages of SDL through having an artificially reduced number of states. Before the author starts to draw the diagram, necessary preliminaries (discussed in § D.3.2) must have been completed, for example:

- the structuring of the system in functional blocks;
- the representation of the functional blocks by means of one or more SDL processes per block;
- the choice of input and output signals;
- the use of data in the process.

All the above factors have a crucial effect in-determining the states of each process. The effect of the "choice" of input signals upon the number of states in an SDL diagram is shown by the two examples in Figure D-3.8.11.



Note – Examples a) and b) represent the same function with different levels of detail. As a consequence the number of states is different.

FIGURE D-3.8.11

Reception of an 8-digit telephone number

D.3.8.2.2 *Reduction in the number of states*

Having used a strategy for identifying the states of a process, the author of an SDL diagram may feel that "too many states" have been used. The number of states is important because the size and complexity of an SDL diagram is often closely related to the number of states. There may be some means available for reducing the number of states, but the fact that an SDL diagram is complex is not, by itself, a reason for changing it, as the complexity of the diagram may simply be a reflection of the inherent complexity of the process which it defines. In general the set of states should be chosen to give maximum clarity to the sequential interaction between the process and its environment. Such clarity is in general not achieved by minimizing the number of states. The number of independent sequences handled by a process has a multiplicative effect on the number of states. It is therefore desirable to treat independent sequences in separate processes as this will reduce the number of states and increase clarity.

The number of states can be reduced by separation of common functionalities, by merging states, by using the procedure concept or by using the service concept. Particular data structures can also lead to a reduced number of states. Alternative representation can benefit from the use of macros.

D.3.8.2.3 *Separation of common functions*

It may become clear when planning an SDL diagram, that to define a particular and repetitive aspect of a process will require the representation of repetitive states. In Figure D-3.8.12 part of a line-signalling process SDL diagram is shown which illustrates the requirement that a line signalling tone must be present for a particular length of time before the line signal is considered to have been detected.

To specify this an intermediate state is required between the state No_line_signal_detected and the state Conversation. Let us assume that in a complete diagram, such a common function would have to be repeated at every point where the signal is detected. An alternative way is to define a separate process which is responsible for monitoring the line signalling tone and detecting signals on the basis of the specified recognition time. The existence of this new process would enable the diagram shown in Figure D-3.8.12 to be drawn as shown in Figure D-3.8.13. (In a given context, the figures can be made equivalent, at the expense of introducing a new signal Valid_line_signal.)

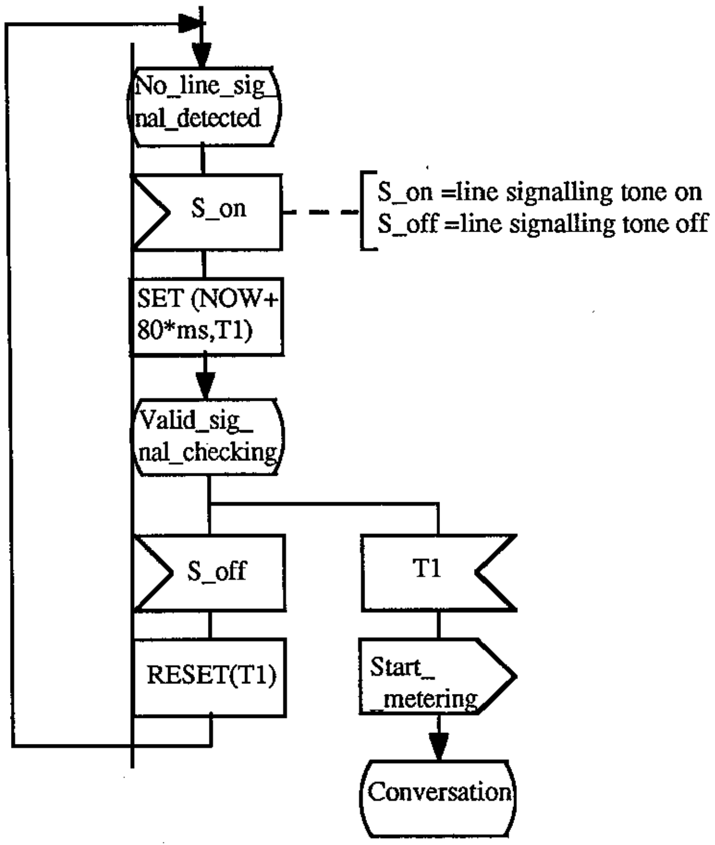


FIGURE D-3.8.12

Example of an SDL diagram for a composite call-handling and line signal detection

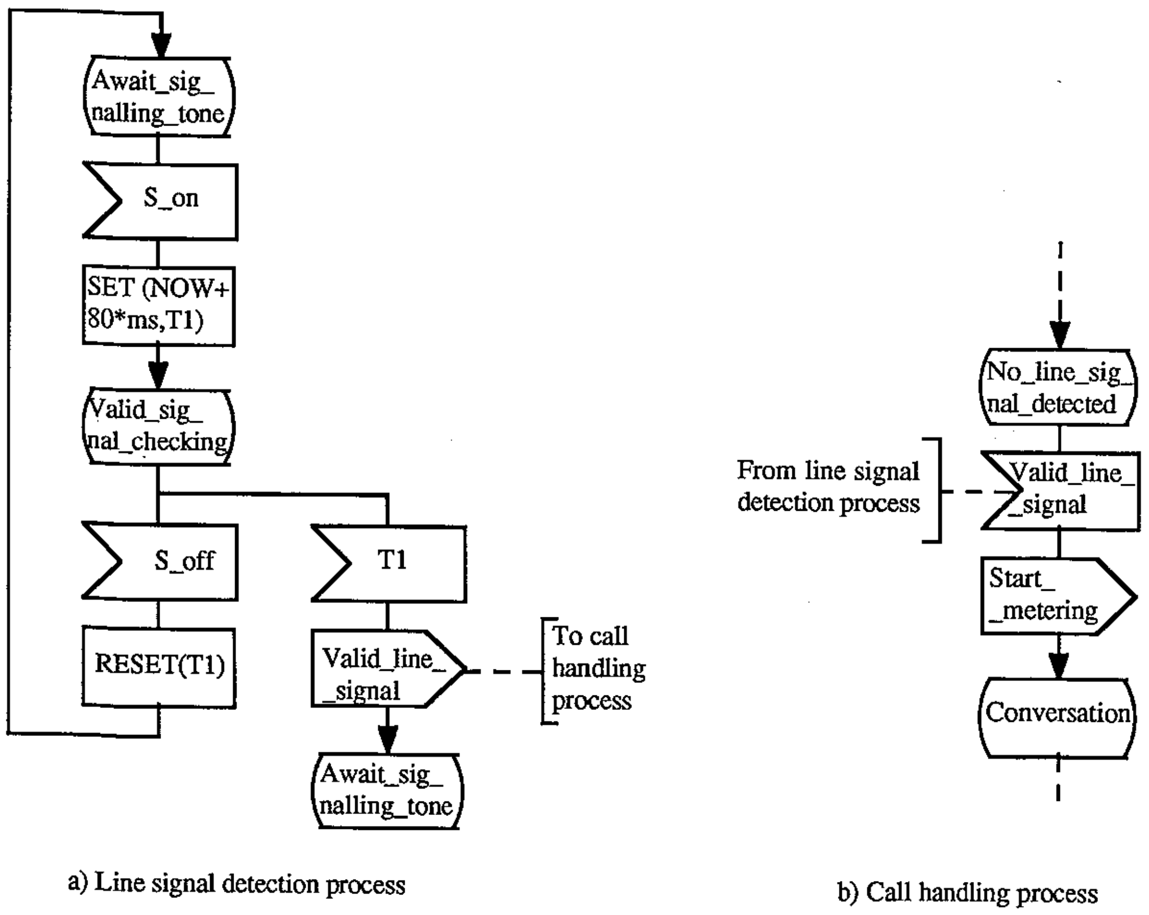


FIGURE D-3.8.13

Example of the segregation of a common function (line signal detection) to avoid repetitive states such as Valid_signal_checking

The slight difference between the process shown in Figure D-3.8.12 and the two shown in Figure D-3.8.13 should be noted.

The process in Figure D-3.8.12 starts metering immediately upon reception of the T1 whilst the call handling process (process b) in Figure D-3.8.13) starts metering upon the reception of the valid line_signal and this is generated and sent upon reception of the Ti by the process a) of Figure D-3.8.13. This implies that in this second case the metering starts after Ti + signal generation, sending and reception time. In addition there may be other signals that have been queued in the time it takes to generate and send the valid line_signal.

A second aspect to be noted is that this separation of a subfunction would not be possible, if the signal to be received by the subfunction has to be received by the main function as well, because a signal is always directed to one process only. If in the example the same signal S_off has to be received in another state, where it is not required to validate it for time tl it would not have been possible to separate the validation subfunction into another process.

In general, the solutions with separate processes are useful in cases where signals are to be processed in a manner which is independent of the states in the main process. In this case pre- and post-processes can handle the detailed sequences and relieve a main process of the details. This will often provide a useful modularity too because particularities of e.g. signalling systems can be isolated from the more service oriented main process.

Another way to approach the problem is to use the service concept, explained in D.5.3.

A different solution to this problem would be to use the MACRO notation as explained in D.3.3.1. In this case we obtain the desired compactness in the diagram without altering in the slightest the semantics of the original one. Furthermore the MACRO can be called from several states, if the logic of the process should require it.

D.3.8.2.4 *Merging of states*

If in an SDL diagram the future of two states is the same, then, irrespective of their history, they can be merged into one without affecting the logic of the diagram.

Figure D-3.8.14 shows part of an SDL diagram with two states which differ in their history but whose futures are "identical". In Figure D-3.8.15 the two states have been combined to form one state. This is a fairly trivial example where the reduction in complexity is small but the technique can be used to obtain significant simplification. The SDL semantics do not allow a decision following a state to determine the history of the process before the state (i.e. for this example, was A6 or 134 sent ?) unless this information had been explicitly stored prior to entry into the state. Note that naming of data in an input causes the value to be stored.

A state should have a clear relation with the possible logical situations of the process, and therefore it is not advisable to merge different logical situations in one state.

Care should be taken when a merged diagram is changed later on. The user should investigate whether the intended change effects the original two (or more) branches of flow in the same way or not.

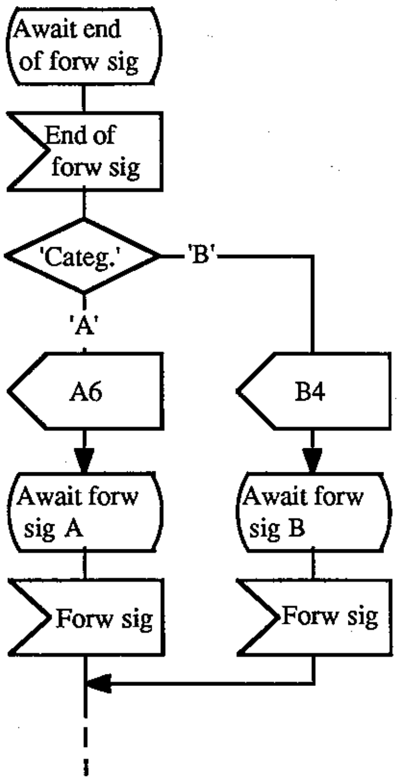


FIGURE D-3.8.14

Example of part of an sdl diagram before merging states

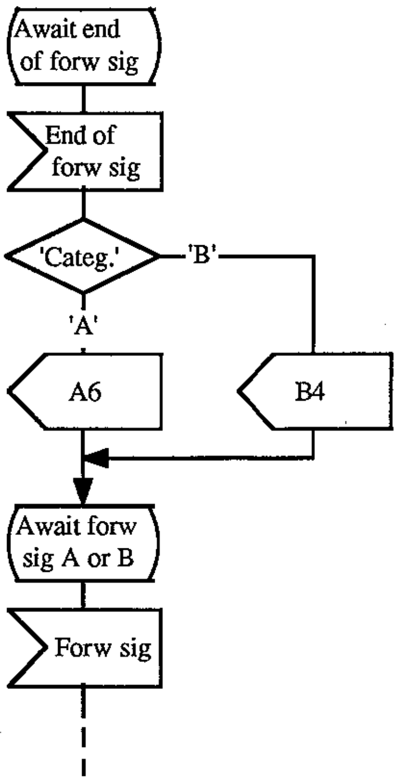


FIGURE D-3.8.15

Example of an sdl diagram with merged states

D.3.8.3 *Inputs*

Paragraph D.3.8.3 has been written to explain the input concept and the use of inputs in SDL diagrams without the save concept. The save concept and the use of inputs and saves together in an SDL diagram is covered in § D.3.8.4.

D.3.8.3.1 *General*

An input symbol attached to a state means that if the signal named within the input symbol arrives while the process is waiting in this state the transition which follows the input symbol should be interpreted. When a signal has triggered the interpretation of a transition, the signal no longer exists and is said to have been consumed.

A signal may have associated values. For example, a signal named "digit" serves not only to trigger the execution of a transition by the receiving process, but also to carry with it the value of the digit (0-9), which can be used by the receiving process.

In SDL/PR the INPUT statement contains a list of signal identifiers. Values contained in signals are named using variable identifiers. The variable identifiers have to be of the sort indicated in the signal definition, so their position is very important. These variable identifiers are contained within round brackets, and they are separated by commas (see Figure D-3.8.16). If one or more signal values are discarded, the corresponding variables are missing, and this is represented by two or more consecutive commas (Figure D-3.8.17).

```
. . .
SIGNAL sig1 (INTEGER,BOOLEAN,INTEGER);
. . .
PROCESS . . .
    . . .
    DCL a INTEGER, b BOOLEAN, c INTEGER;   /* declarations */
    . . .
    STATE st1;
        INPUT  sig1(a,b,c);   /* a correct input */
        . . .
    STATE st2;
        INPUT  sig1(a,c,b);   /* an incorrect input */
        . . .
ENDPROCESS . . .
```

FIGURE D-3.8.16

INPUT statements

```
INPUT a(var1,var2,,var4);
```

Note - In this statement the third value of the signal is discarded

FIGURE D-3.8.17

Input of signal a with only 3 of its 4 values defined

In SDL/GR an input is represented by means of an input symbol containing the list of signal identifiers and the corresponding variable identifiers for the transported values. To make these values available to the process, they must be named in the input symbols within parentheses.

Examples of the values reception from inputs are shown in Figures D-3.8.18, D-3.8.19 and D-3.8.20.

Named values becomes available to the receiving process when the input is interpreted.

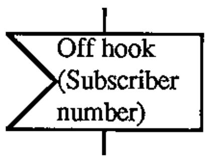
Figure D-3.8.18 shows reception of the signal "Off hook". The signal "Off hook" has associated the value 9269 (Subscriber number). This signal may be received in three ways as shown in a-c of the figure.

Figure D-3.8.19 shows how to send and receive several values in one signal. Each item must be separated from the next by a comma. In Figure D-3.8.20 c) we show how to ignore unwanted values by leaving a gap in the sort list.

Note that in the output signal we could write expressions for El, E2 or E3, but in the input signal we must use variables to receive the values sent.

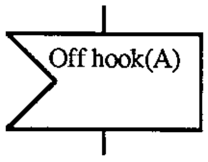
In SDL it is unnecessary to draw input symbols to represent signals whose arrival would initiate a null transition (i.e. transition containing no actions and which leads back to the same state). The convention is that for any signal not shown in an explicit input symbol at a particular state, there exists at that state an implicit input symbol and null transition. By this convention, the two diagrams shown in Figure D-3.8.21 are logically equivalent and either can be used.

a)



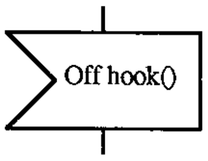
Note - The value 9269 is stored in a variable called "Subscriber number"

b)



Note - The value 9269 is stored in a variable called A.

c)

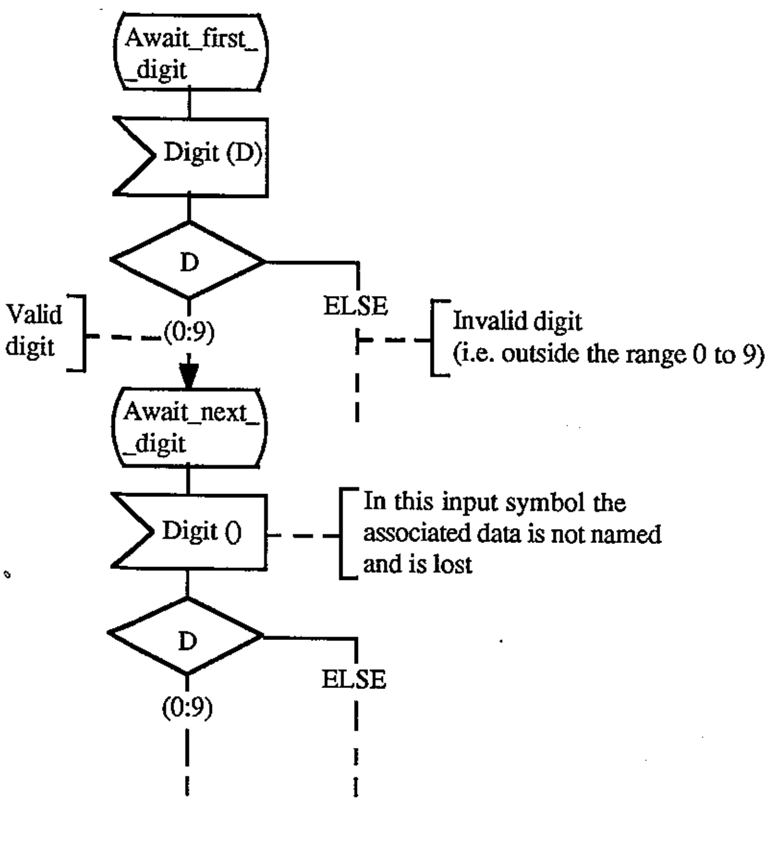


Note1 - There is no data name and the value 9269 is lost and is not available to the receiving process.

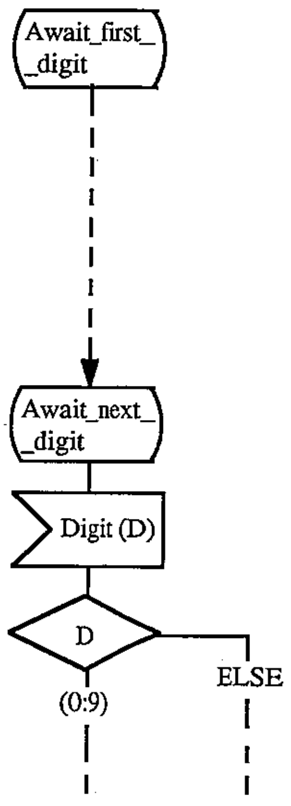
Note2 - The signal name (Off hook) must correspond to the output signal name but data names do not have to correspond.

FIGURE D-3.8.18

Example of values reception in a process


Await_first_ _digit
Digit (D)
D
Valid digit
(0:9)
ELSE
Invalid digit (i.e. outside the range 0 to 9)
Await_next_ _digit
Digit ()
In this input symbol the associated data is not named and is lost
D
(0:9)
ELSE

a) Incorrect version



b) Correct version

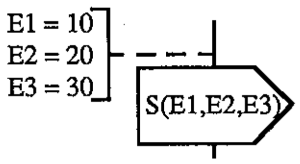
Note 1 – In a) the second decision will use the value of D obtained from the first signal.

Note 2 – In b) the value of D will be that of the current digit. Values of previous digits will be overwritten.

FIGURE D-3.8.19

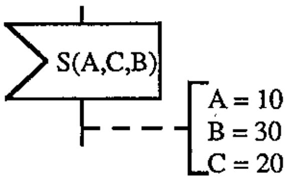
Part of a digit-receiving process

a)



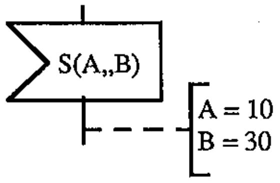
Note - The output signal S has three variables called E1,E2 and E3. These items have three values, say currently 10,20 and 30.

b)



Note - The corresponding input signal S in the receiving process, names these items A, C and B respectively.

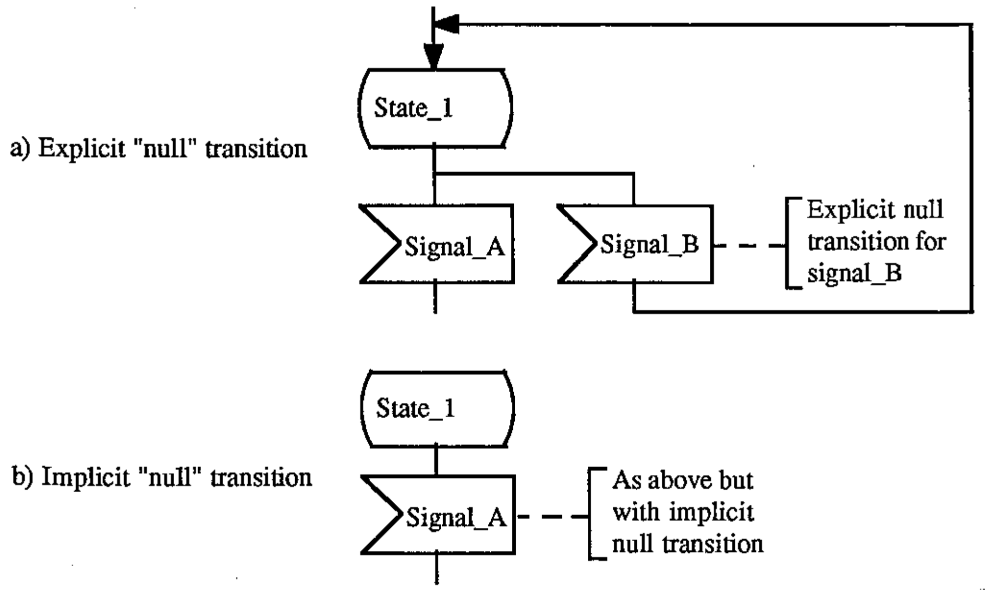
c)



Note - This input signal only names two variables. The middle value is lost.

FIGURE D-3.8.20

A signal with several associated values


a) Explicit "null" transition
State_1
Signal_A
Signal_B
Explicit null transition for signal_B
b) Implicit "null" transition
State_1
Signal_A
As above but with implicit null transition

Note - If data is associated with Signal_B, it is lost in both cases. If however a data name is shown (e.g. Signal_B(x)), in case a) the data value is kept. In this case the two examples are no longer identical.

FIGURE D-3.8.21

Explicit and implicit representation of a "null" transition

Where a: number of inputs lead to the same transition, all the relevant signal identifiers may be placed within one input symbol. A shorthand notation is provided in SDL to represent an input of all the signals (valid for that process) not explicitly mentioned in that state. Figure D-3.8.22 illustrates this point and the diagrams in the figure are logically equivalent. If all the signal identifiers are mentioned they must be separated by commas.

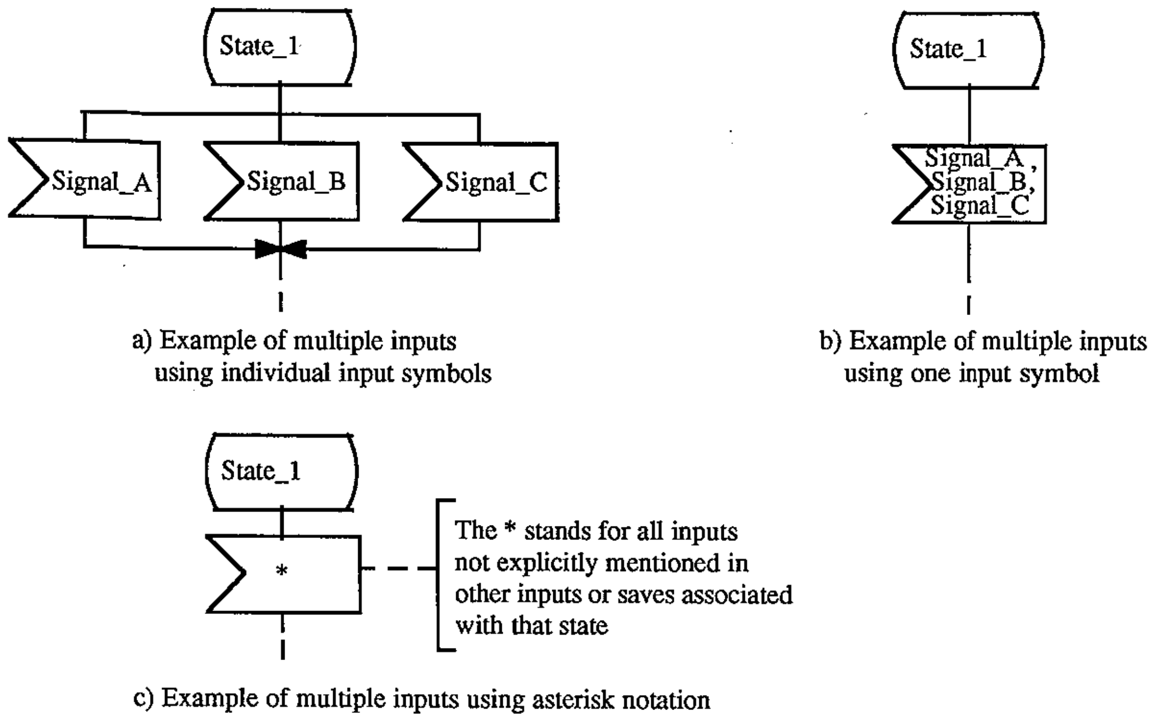

State_1
Signal_A
Signal_B
Signal_C
a) Example of multiple inputs using individual input symbols
State_1
Signal_A, Signal_B, Signal_C
b) Example of multiple inputs using one input symbol
State_1
*
The * stands for all inputs not explicitly mentioned in other inputs or saves associated with that state
c) Example of multiple inputs using asterisk notation

FIGURE D-3.8.22

Alternative representation of multiple inputs

D.3.8.3.2 *Implicit queuing mechanism*

One or more signals may be waiting for consumption when a process reaches a new state. This means that signals must be queued in some way. When a signal arrives at the destination process it is given to its input queue. The SDL semantics define a first-in first-out conceptual queuing mechanism for each process where signals are considered for consumption by a process in the order of their arrival at that process. When the process reaches a state, it is given one and only one signal from the queue. This means that if the queue is not empty, the process consumes the first signal from the queue. If the queue is empty, the process waits in the state until a signal arrives at the queue, whereafter it is consumed by the process.

Figure D-3.8.23 uses the conceptual queue to explain the operation of the depicted SDL process where transition times are non-zero. It should be noted that:

- The save concept is not used and therefore the signals are consumed in the order in which they arrive.
- The sequence of signal arrival is important. Had "C" arrived before "B" in the transition between State 1 and State 2, the state sequence would have been 1, 2, 3 instead of 1, 2, 4.
- Because the queue is not empty when the process arrives at both State 2 and State 4, the process does not wait at either of the states.
- It is not possible to assign any priority to a signal. For inter-services communications a special mechanism is provided so that the signals interchanged between services are handled before other signals (Ref. § D.5.3).

If transition times are zero, then every signal will be consumed at the time it arrives at a process, unless the save operation is used (§ D.3.8.4).

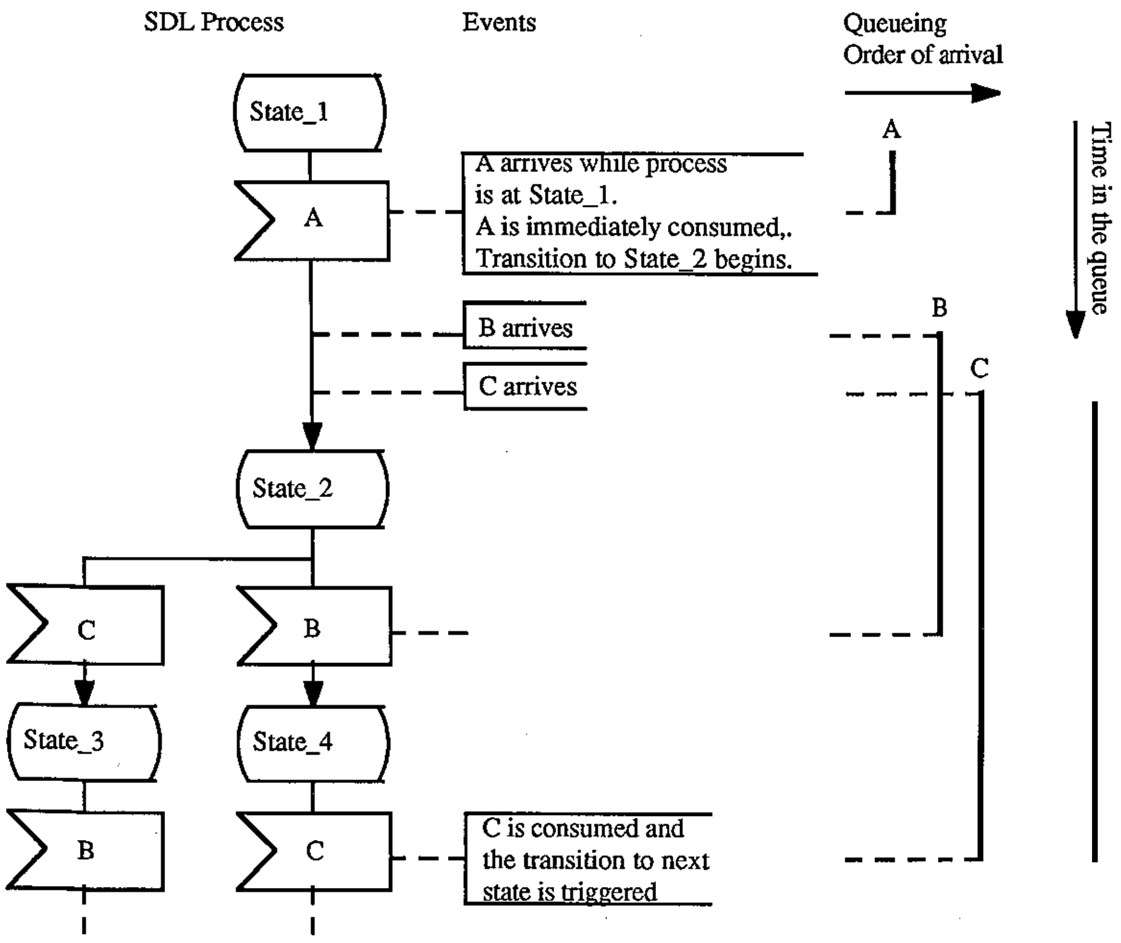

SDL Process
Events
Queueing
Order of arrival
State_1
A
A arrives while process is at State_1.
A is immediately consumed,.
Transition to State_2 begins.
A
B arrives
C arrives
B
C
Time in the queue
State_2
C
B
State_3
State_4
B
C
C is consumed and the transition to next state is triggered

FIGURE D-3.8.23

Example of the operation of the implicit queueing mechanism

D.3.8.3.3 *Reception of signals not normally occurring*

In each state all possible signals must be shown implicitly or explicitly. Exceptions (unexpected but theoretically possible signals) can occur in almost all states. Normally the author does not show such possibilities, with the effect that such a signal will be discarded if it arrives. If the author however wants to include exceptions in his diagram all states must be expanded with an extra input.

Another possibility is to make use of multiple appearances of a state together with the "all" symbol (*). For example if the signal A_ON_HOOK can be received in all states and if the subsequent actions are identical, the method shown in Figure D-3.8.24 might be used.

D.3.8.3.4 *Simultaneous arrival of signals*

In § 2 of the SDL Recommendation it is stated that signals can arrive simultaneously at a process and that they will be ordered arbitrarily.

If a user designs a process which may get simultaneous signals, he should take care that the order of arrival cannot upset the desired operation of the process.

SDL does not define priority of signals, i.e. that simultaneous arrival of signals means that one is chosen arbitrarily. It must be noted, however, that signals for inter-services communications are always handled first (Ref. § D.5.3).

If several signals are available when the process enters a state, only one signal is presented to the process and thus recognized as an input. The SDL semantics implies that the other signals are in fact retained.

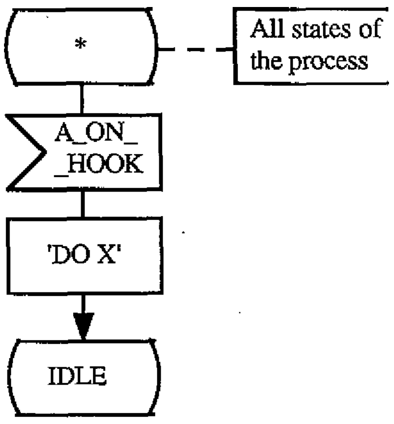


FIGURE D-3.8.24

Example of handling of signals to appear in several states

D.3.8.3.5 *Sender identification*

Each signal carries with it the process instance value (PId) of the sending process. When a signal is -consumed, the PId value of the sending process can be obtained by means of the expression SENDER. Figure D-3.8.25 shows an example of how this may be used.

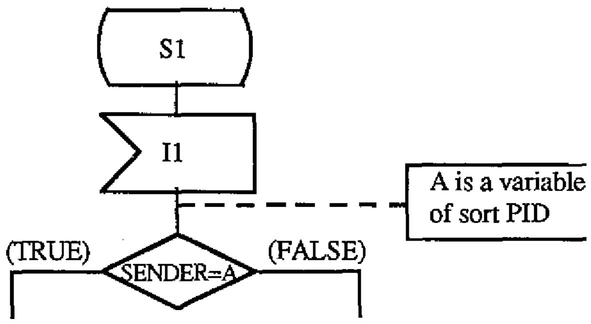


FIGURE D-3.8.25

Use of SENDER expression

D.3.8.4 *Saves*

The save concept allows the consumption of a signal to be delayed until one or more other signals, which arrive subsequently, have been consumed. As discussed in § D.3.8.3 unless the save concept is used, signals are consumed in the order in which they arrive.

The concept of save can be used to simplify processes in cases where the relative arrival order of some signals are not important and the actual arrival order is indeterminate.

At every state, every signal is treated in one of the following ways:

– it is shown as an input;

– it is shown as a save;

– it is covered by an implicit input leading to an implicit null transition.

The operation of the implicit queuing mechanism, introduced in § D.3.8.3, also applies to the save concept. On arrival, signals enter the queue and when the process reaches a state, the signals in the queue are reviewed one at a time and in the order in which they arrived. A signal covered by an explicit, or implicit, input is consumed and the related transition executed. A signal shown in a save is not consumed and remains in the queue in the same sequential position and the next signal in the queue is considered. No transition follows a save.

In SDL/PR the save construct is expressed by means of the keyword SAVE followed by a list of signal identifiers. A simple example of SAVE statement is given in Figure D-3.8.26.

```
...
STATE State_31;
      SAVE s;
      INPUT r;
      NEXTSTATE State_32;
STATE State_32;
      INPUT s;
      ...
```

FIGURE D-3.8.26

Example of the use of save

In SDL/GR the save concept is represented by means of a save symbol containing the signal identifiers.

Similarly to the case of inputs, an "asterisk notation" can be used to represent a save of all the signals (valid for that process) not explicitly mentioned in that state.

Figure D-3.8.27 shows an example of an SDL process which incorporates a save symbol. It should be noted that signals S and R are consumed in the order R, S – the reverse order to that in which they were received. A single save symbol can save a signal only while the process is at the state where the symbol appears, and saves it for the duration of the transition to the next state. At the next state, the signal will be consumed via an explicit or implicit input (as shown in Figure D-3.8.27, unless either the save symbol with the signal name is repeated, or there happens to be another saved signal available for consumption ahead of it in the implicit queue (as shown in Figure D-3.8.28).

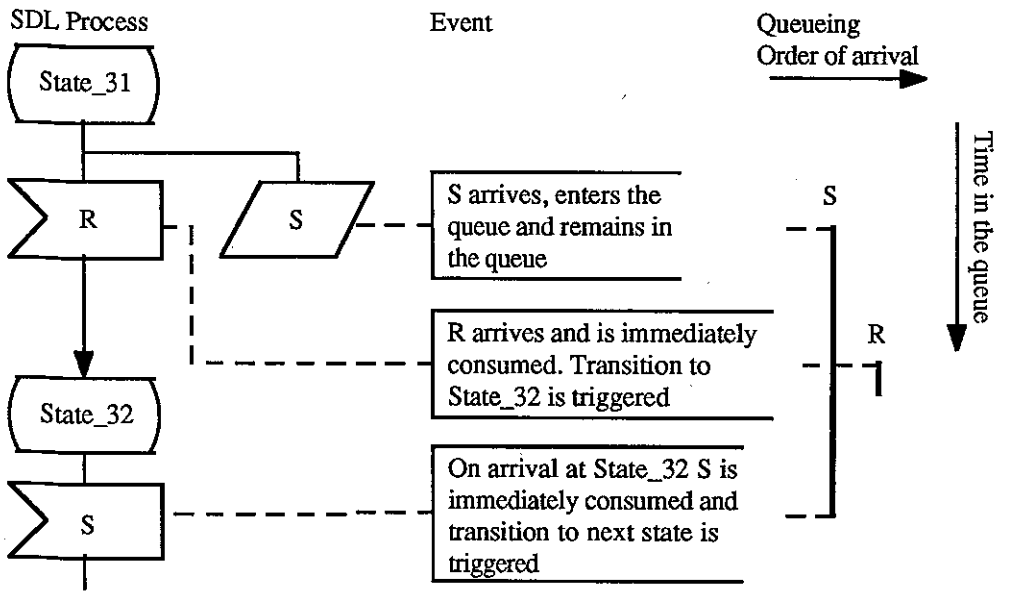

SDL Process
Event
Queueing
Order of arrival
Time in the queue
State_31
R
S
S arrives, enters the queue and remains in the queue
S
R arrives and is immediately consumed. Transition to State_32 is triggered
R
State_32
S
On arrival at State_32 S is immediately consumed and transition to next state is triggered

FIGURE D-3.8.27

Example of an SDL diagram with save symbol showing operation on implicit queueing mechanism

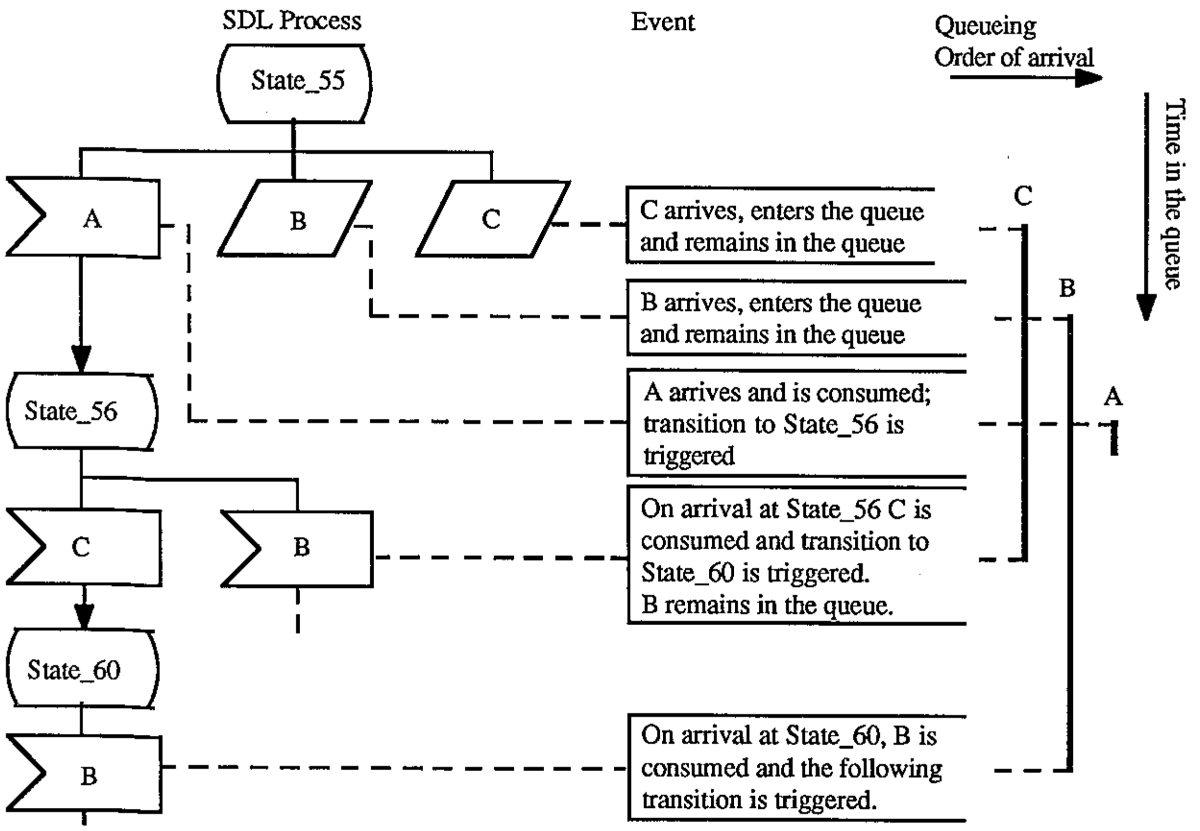

SDL Process
Event
Queueing
Order of arrival
Time in the queue
State_55
A
B
C
C arrives, enters the queue and remains in the queue
C
B arrives, enters the queue and remains in the queue
B
A arrives and is consumed; transition to State_56 is triggered
A
State_56
C
B
On arrival at State_56 C is consumed and transition to State_60 is triggered. B remains in the queue.
State_60
B
On arrival at State_60, B is consumed and the following transition is triggered.

FIGURE D-3.8.28

Second example of the use of the save symbol

A saved signal becomes available to a process only through a corresponding input symbol (explicit or implicit). In particular no questions about a saved signal may be asked in a decision prior to its consumption in an input; nor are its associated values available.

At a state where more than one signal is to be saved, either a save symbol may be given for each signal, or they may all be shown within one save symbol. If several signals are to be saved, the semantics of the save symbol implies that the order of their arrival is preserved.

A third example of the use of the save is given in Figure D-3.8.29 with a description of the operation of the conceptual queuing mechanism in Figure D-3.8.30.

The save can be used to simplify diagrams. For example by saving a signal it is possible to avoid receiving it and having to store its associated values until the next state.

Although the save can be used at every level of specification, at lower levels it may be necessary to describe the actual mechanism which implements the save concept.

If the save is not used with care, the queue of saved signals may grow very big or may hold signals for too long so that an old signal is consumed where a fresh one was needed.

SDL provides no bound on the lenght of the queue, which may lead to problems on implementation

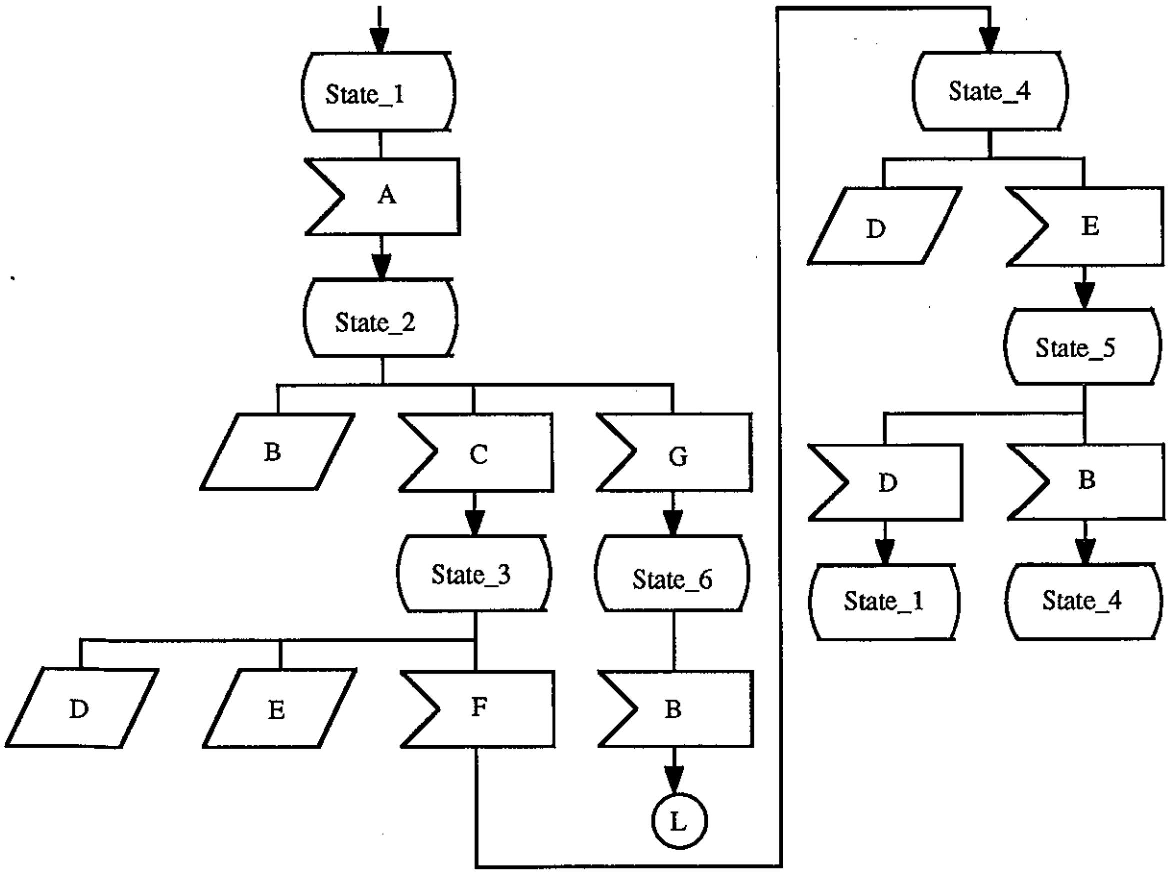
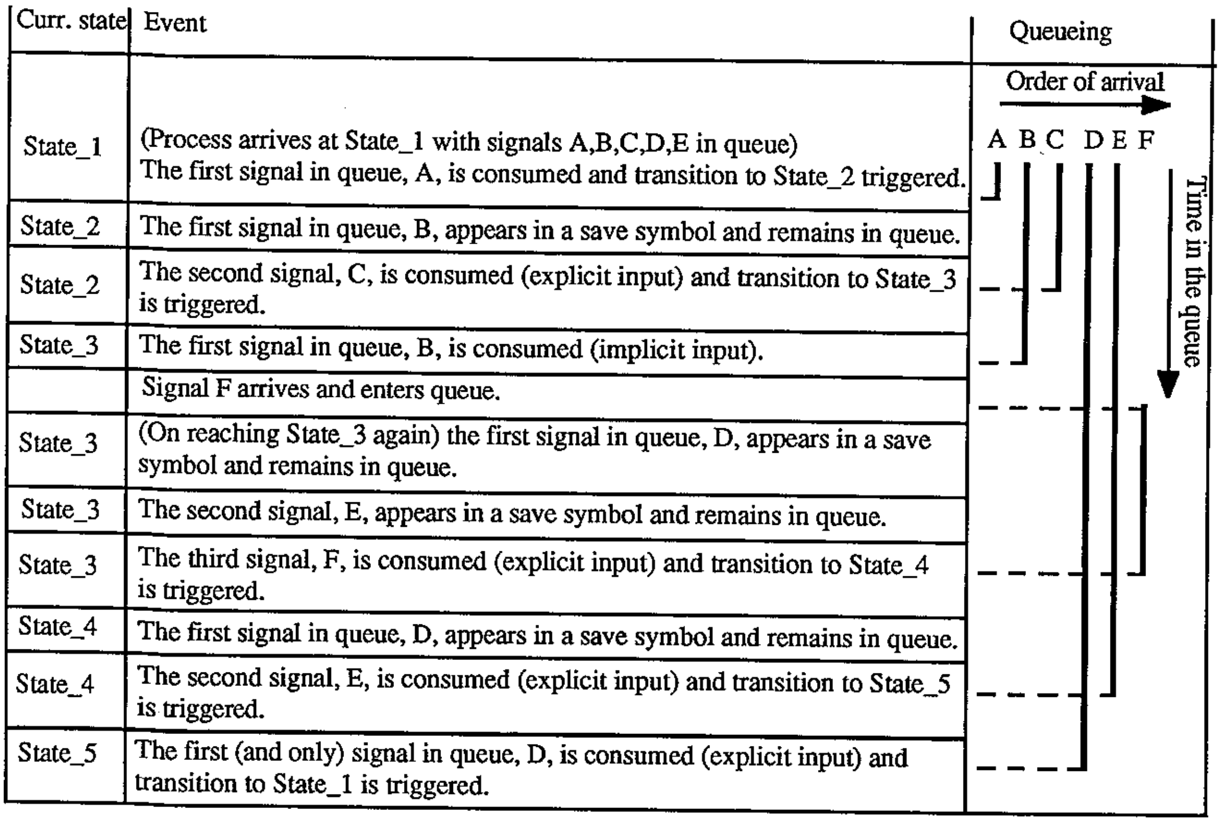


FIGURE D-3.8.29

Example of a more complex SDL diagram



Curr. state	Event
State_1	(Process arrives at State_1 with signals A,B,C,D,E in queue) The first signal in queue, A, is consumed and transition to State_2 triggered.
State_2	The first signal in queue, B, appears in a save symbol and remains in queue.
State_2	The second signal, C, is consumed (explicit input) and transition to State_3 is triggered.
State_3	The first signal in queue, B, is consumed (implicit input).
	Signal F arrives and enters queue.
State_3	(On reaching State_3 again) the first signal in queue, D, appears in a save symbol and remains in queue.
State_3	The second signal, E, appears in a save symbol and remains in queue.
State_3	The third signal, F, is consumed (explicit input) and transition to State_4 is triggered.
State_4	The first signal in queue, D, appears in a save symbol and remains in queue.
State_4	The second signal, E, is consumed (explicit input) and transition to State_5 is triggered.
State_5	The first (and only) signal in queue, D, is consumed (explicit input) and transition to State_1 is triggered.

FIGURE D-3.8.30

Operation of queueing mechanism

D.3.8.5 *Enabling condition and continuous signals*

Enabling conditions allow conditional reception of signals based on the specified enabling condition. If the condition is true, the signal is consumed and the transition is interpreted. If the condition is false, the signal is saved and the process remains in the state until either another signal arrives or until the condition changes from being false to being true. This can be illustrated by the example in Figure D-3.8.31. When P1 enters state S l the condition (i.e. whether IMPORT(X,P2) equals 10) is evaluated. If the condition is true, signal B may be received. Otherwise, signal B is saved. In this example, A arrives and causes a transition to S2. During the transition, X changed to 11 and P2 exports its new value; now the condition attached to signal B, in state S2, is true. Since B is the first signal in the queue, the transition following it is performed and the process ends in state S3.

Some important attributes of enabling conditions are:

1) The enabling condition is tested when the process arrives at a state and then it is continuously monitored while the process remains in the state. Thus if the exported value of X had changed from 9 to 11 and then to 12 while executing the transition following the reception of A, the process would have remained in S2.

2) Enabling conditions can be based on local variables and/or any language construct that may be included in an expression (e.g. IMPORT, VIEW, NOW).

3) While it is possible to use more than one enabling condition per state, it is not allowed to use more than one enabling condition for the same signal. Thus the condition shown in Figure D-3.8.32 is not allowed. If multiple conditions are required for a given signal, they can be combined in one Boolean expression as shown in Figure D-3.8.33.

The enabling conditions can be evaluated several times and in any order, so the user must be careful if the expressions have mutual side-effects (e.g. combining IMPORT and SENDER).

It must be noted also that the signal specified in the enabling condition cannot influence the boolean value of the condition because its transported values are not assigned before signal consumption. For example the statements:

```
INPUT x(A) PROVIDED A=5); . . .
INPUT y PROVIDED(SENDER=pidl);
```

are misleading as the values of A and of SENDER in the conditions correspond to the situation before the signal consumption.

Continuous signals have the same basic properties as the enabling condition except that no signal is attached to it. Thus when entering the state with no signals in the queue which can cause a transition, the continuous signals are checked and if one is true, the transition following it is performed. This can be illustrated by the example in Figure D-3.8.34. Initially, the process is in state S 1, and the exported value of X is 9. When signal A arrives, it causes the transition to S2. During the transition, X changed to 11 and its new value is exported. Since there were no other signals in the queue, the transition to S3 is performed.

Some important attributes of continuous signals are:

1) as with enabling conditions, the value of the condition is checked only when the process arrives at the state or is in the state;

2) multiple continuous signals are allowed for each state. When more than one continuous signal is attached to a state, the continuous signal with the higher priority (lowest priority number) will be tested first. No two continuous signals may have the same priority number. In all cases, the priority of the continuous signal is lower than that of any other signal. Once again, this is caused by the underlying system of SDL; however, the way that continuous signals are modelled in SDL by using the signals sent when exporting variables, allows priorities for continuous signals and in fact makes it necessary to prevent ambiguity, when more than one continuous signal is present. This is illustrated in Figure D-3.8.35. Initially, the process is in state S 1, and its import expressions give the values of 10 for X and 11 for Y. Since both continuous signals are true, the one with the higher priority (lower priority number) is chosen and the transition to S2 is performed. In S2, the condition on Y is no longer true and so even though the priority of the continuous signal for X is lower than the one for Y, the transition following it is performed and the process arrives at state S3.

3) When the transition from a continuous signal is followed, the expression SENDER returns the same value of SELF.

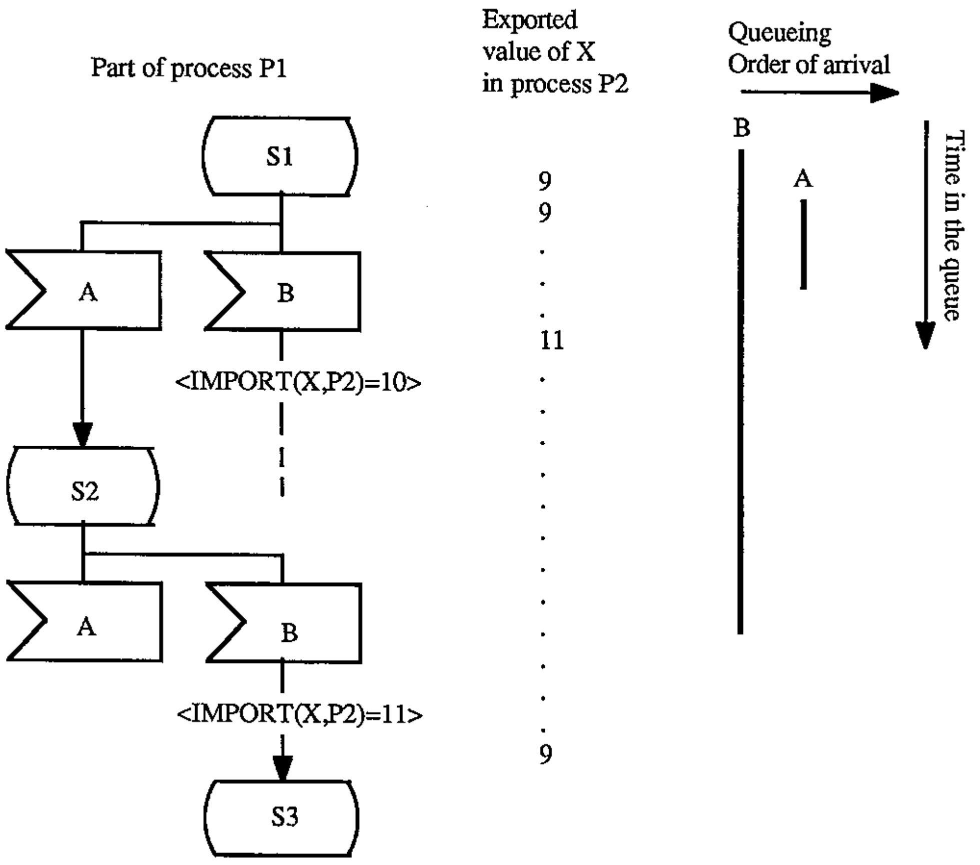


FIGURE D-3.8.31

Enabling condition

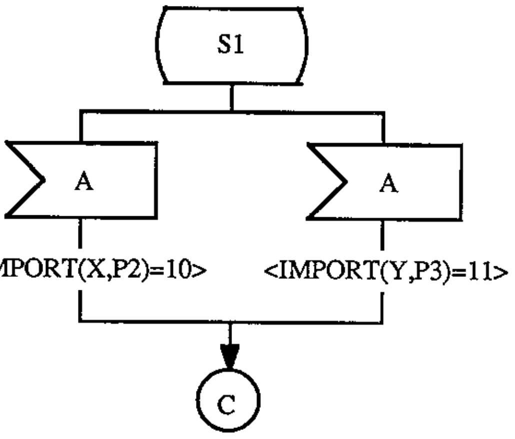


FIGURE D-3.8.32

Illegal enabling condition

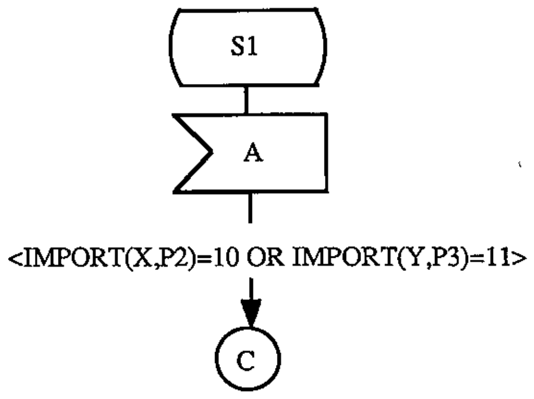


FIGURE D-3.8.33

Correct solution for Figure D-3.8.32

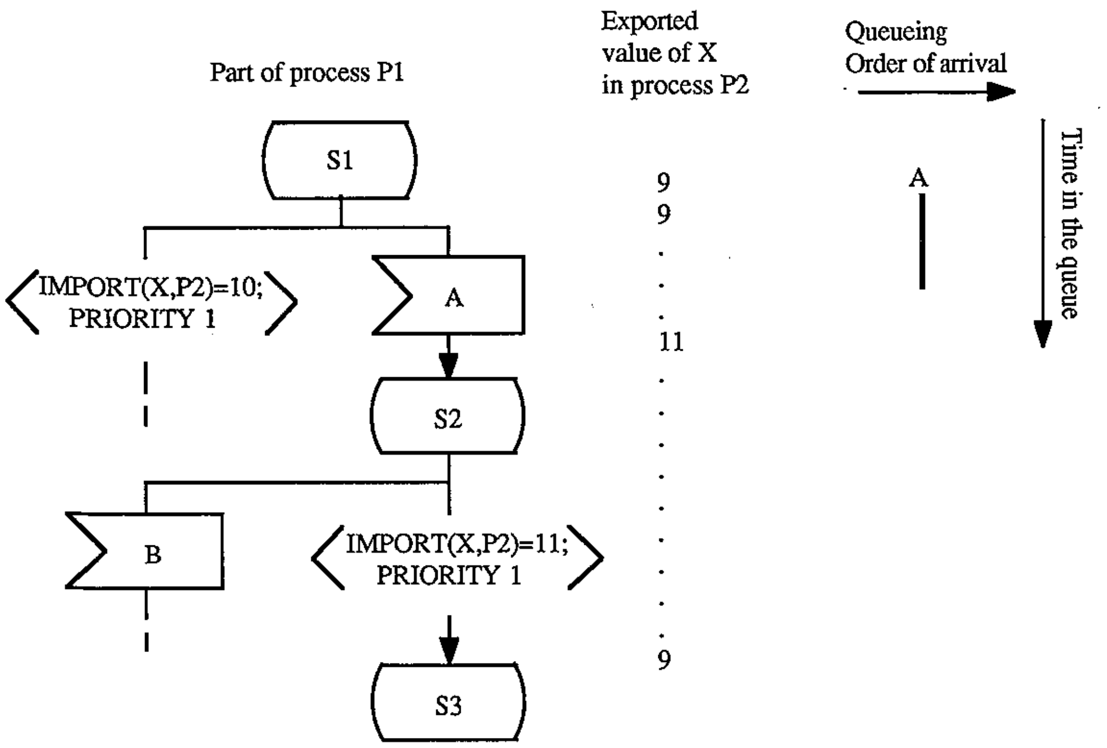
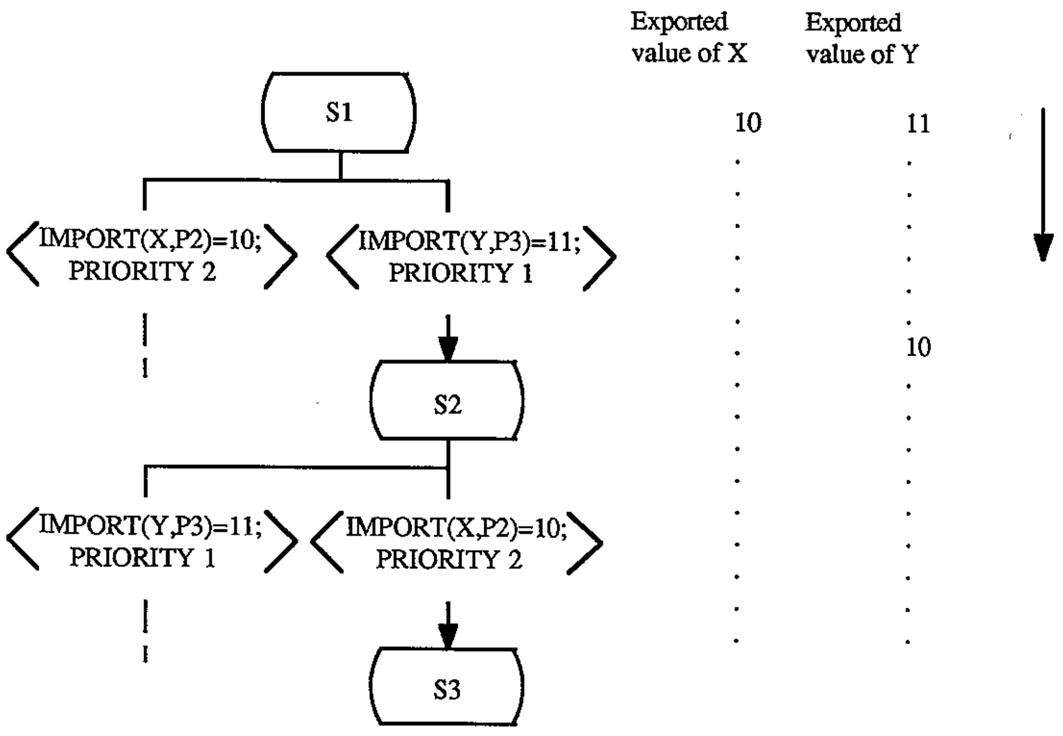

Part of process P1
Exported value of X in process P2
Queueing
Order of arrival
Time in the queue
S1
IMPORT(X,P2)=10; PRIORITY 1
A
S2
B
IMPORT(X,P2)=11; PRIORITY 1
S3
9
9
11
9
A

FIGURE D-3.8.34

Continuous signals



Exported value of X
Exported value of Y
S1
IMPORT(X,P2)=10; PRIORITY 2
IMPORT(Y,P3)=11; PRIORITY 1
S2
IMPORT(Y,P3)=11; PRIORITY 1
IMPORT(X,P2)=10; PRIORITY 2
S3
10
11
10

FIGURE D-3.8.35

Continuous signals with priority

D.3.8.6 *Outputs*

An output is the sending of a signal from one process to another (or to itself). Because control over the consumption of the signal is associated with the receiving process (see § D.3.8.3) the semantics directly relating to the output is relatively simple. From the point of view of the sending process an output can often be regarded as an instantaneous action which, once completed, has no further direct effect on the sending process, which will not be directly aware of the fate of the signal.

An output action represents the sending of a signal and the association of values if any. Values may be associated with an output signal by placing values in parentheses, or by placing expressions having values in parentheses (see Figure D-3.8.37).

In SDL/PR an output action is represented by means of the keyword OUTPUT followed by a list of signal identifiers. A list of actual parameters within round brackets can be associated to each signal identifier. If there is no actual parameter in the output corresponding to a certain sort in the signal definition, the corresponding variable in the receiving input will have the value "undefined".

The destination process instance must be expressed in the output statement by the keyword TO followed by a process instance expression. If the destination process instance can be uniquely determined by the context the TO clause can be omitted. An additional addressing condition can be provided in the output statement by means of the keyword VIA followed by a list of signal route or channel identifiers.

Figure D-3.8.36 shows some valid examples of output statements.

```
OUTPUT sig 1(2,true,10);
OUTPUT sig2, sig3(parl,par2) TO process_a;
OUTPUT sig4 TO process_b VIA channel_x,channel_y;
OUTPUT sig5 VIA signal_route_z;
```

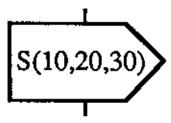
FIGURE D-3.8.36

Examples of output statements

In SDL/GR an output is represented by means of an output symbol containing the specification of signals, actual parameters and optionally destination and/or VIA construct.

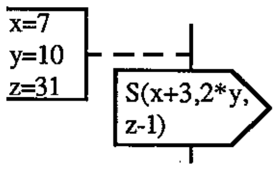
Each output must be directed to a specific process instance. Since it is usually impossible to know the process instance for any process at the time a specification is produced, the normal method of addressing signals is to use a variable or expression in the TO keyword. Figures D-3.8.38, D-3.8.39 and D-3.8.40 show some examples. In Figure D-3.8.38 the process parameter out_to is given the value of a process instance at the time the process is created. Out_to is then used within the process as the link between this process and its connected process. In designing the system, care should be taken to ensure that the type of the process indicated by out_to is able to receive the signals being sent. In Figure D-3.8.39 the built-in expression SENDER is used to send a signal back to the process which sent the signal. In Figure D-3.8.40, the signal is addressed to the process's most recently created offspring process.

a)



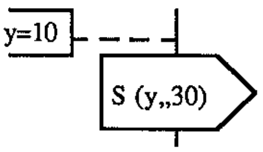
Note - Signal S has three values, 10,20 and 30 associated with it.

b)



Note - On interpreting the output, x,y and z have (in this example) the values 7,10,31 respectively. The output sends the values 10, 20, 30.

c)



Note - On interpreting the output, y has (in this example) value 10. The output sends the values 10, an undefined value and 30.

FIGURE D-3.8.37

Output with associated values

```
PROCESS X (0,5);
   FPAR (out_to PD);
   ...
   OUTPUT sig TO out_to;
   ...
```

FIGURE D-3.8.38

Addressing signals using formal parameters

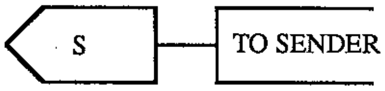


FIGURE D-3.8.39

Addressing signal back to SENDER

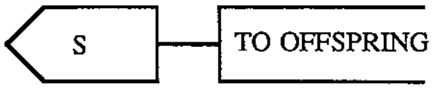


FIGURE D-3.8.40

Addressing a signal to an offspring process

D.3.8.7 *Task*

A task is used in a transition to represent operations on variables or to represent special operation by means of informal text.

In SDL/PR a task is represented by means of the keyword TASK followed by a list of statements or informal texts separated by commas and terminated by a semicolon. A statement in a task can be just an assignment. An informal text consists of a phrase delimited by apostrophes. In Figure D-3.8.41 there are some examples of valid tasks in SDL/PR.

```
TASK     a:=b;

TASK     'connect the subscriber';

TASK     c:=d+e;

TASK     varl:=var2*var3,

         var4:=var5 MOD var6;
```

FIGURE D-3.8.41

Examples of tasks

In SDL/GR a task consists of a task symbol containing the list of statements or informal texts.

SDL users may occasionally have difficulty in deciding whether some aspects of the system being defined should be represented by a task or a separate process. Consider the process shown in Figure D-3.8.42; should "connect_switchpath" be represented as a task or as a separate process? If a separate switch path control process has not been identified, then the task symbol would be appropriate (see Figure D-3.8.42 a)). If a separate switch path control process is identified, then signals communicating with the control process must be used (see Figure D-3.8.42 b)).

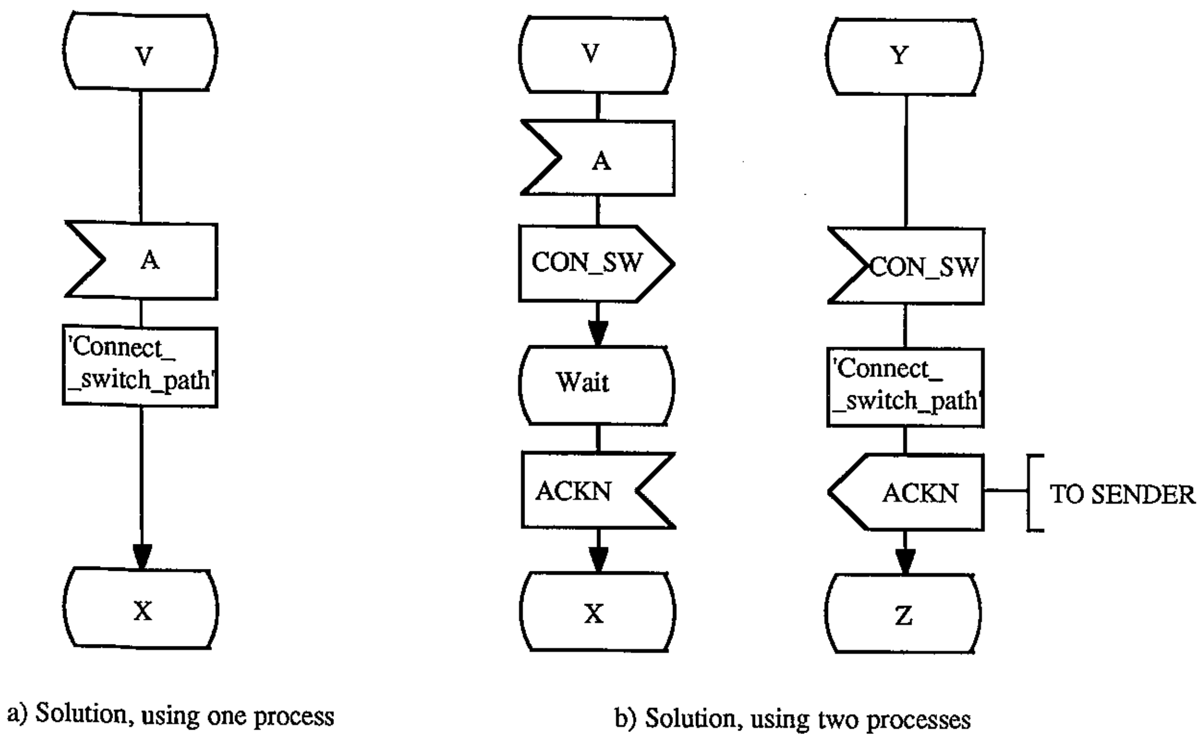


FIGURE D-3.8.42

Two possible solutions for 'connect switch_path'

D.3.8.8 *Decisions*

A decision is an action within a transition which asks a question regarding the value of an expression at the instant of executing the action. The process proceeds to one of the two or more paths following the decision according to the answer. SDL diagram authors should ensure that processes are defined so that they cannot attempt to execute decisions for which the answer is not available; such decisions would make the diagram invalid and could cause considerable confusion.

The question of a decision can be an expression or an informal text. The answers of a decision are represented by one or more possible values obtained by the evaluation of the expression in the question or by one or more informal texts. If the question or one of the answers is informal, then the whole decision is informal. Different answers are separated by commas. The values are represented by constant expressions, by constant expressions with an operator as a prefix, or by ranges whose upper and lower bounds are constant expressions.

The answer values have to be of the same sort as the expression contained in the question.

It is possible to indicate some answers explicitly and to group all the other possible answers using the keyword ELSE.

In SDL/PR the decision is represented by the keyword DECISION followed by the specification of the question and by the list of possible answers, each associated with the correspondent transition. The answers are indicated within round brackets. The set of outgoing transitions is delimited by the ENDDECISION keyword at the end (see Figure D-3.8.43).

```
DECISION question ;
    (answer_a): . . .
    (answer_b): . .
    (answer_c): . .
    ELSE: . . .
ENDDECISION;
```

FIGURE D-3.8.43

Skeleton of a decision

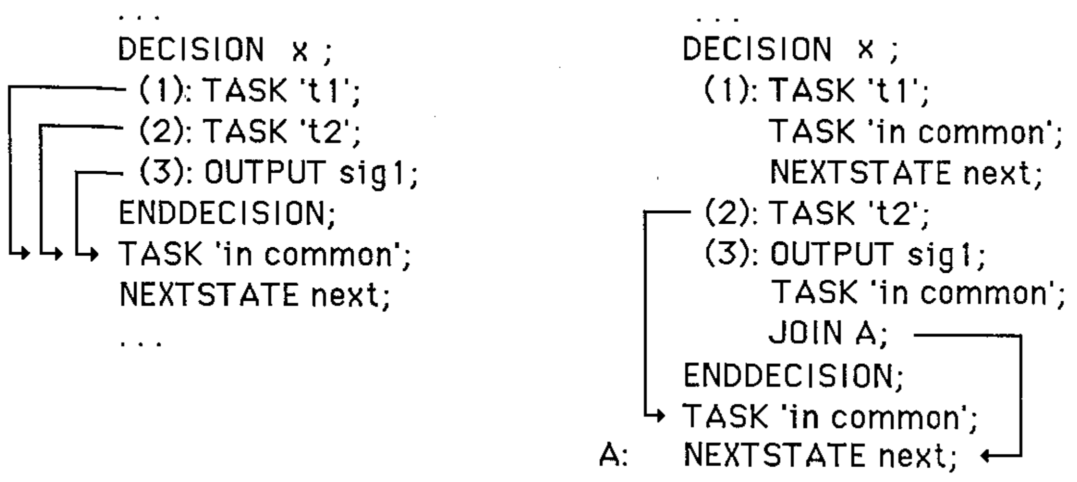
Some examples of decisions are shown in Figure D-3.8.44.

```
…
DCL x INTEGER,a BOOLEAN; DECISION x;
        (2): . . .;
        (2+5): . . . ;
        (6+8,10+9): . . . ;
        (=3): . . .;
        (20:30): . . .;
        (>=100): . . . ;
        ELSE: . . . ;
ENDDECISION;
…
DECISION a;
        (TRUE) : NEXTSTATE s1;
        (FALSE) : NEXTSTATE s2;
ENDDECISION;
…
DECISION 'Subscriber category';
        ('International','National'): . . .
        ('Local'): . . .
ENDDECISION;
…
```

FIGURE D-3.8.44

Examples of decisions in SDLIPR

All transitions terminate at the ENDDECISION keyword. Those transitions which are not terminated with a terminator statement (i.e. join, nextstate, stop) continue at the statement following the ENDDECISION, as shown in the two equivalent branching of Figure D-3.8.45.



```
. . .
DECISION x ;
  (1): TASK 't1';
  (2): TASK 't2';
  (3): OUTPUT sig1;
ENDDECISION;
TASK 'in common';
NEXTSTATE next;
. . .
```

```
   . . .
   DECISION x ;
     (1): TASK 't1';
          TASK 'in common';
          NEXTSTATE next;
     (2): TASK 't2';
     (3): OUTPUT sig1;
          TASK 'in common';
          JOIN A;
   ENDDECISION;
   TASK 'in common';
A: NEXTSTATE next;
```

FIGURE D-3.8.45

Equivalent branching from a decision

The decision statement can also be used to model the IF-THEN structure, the DO-WHILE and the LOOP-UNTIL structure as in structured programming.

In SDL/GR a decision is represented by means of a decision symbol containing the text of the question. The symbol must have two or more branches associated with the corresponding answers. Each answer must be placed to the right or on the top of the corresponding branch or also over the branch interrupting the flow line. In SDL/GR the parenthesis to delimit answers are optional but it is suggested to use them in order to avoid misunderstandings.

Some examples of decisions in SDL/GR are shown in Figure D-3.8.46.

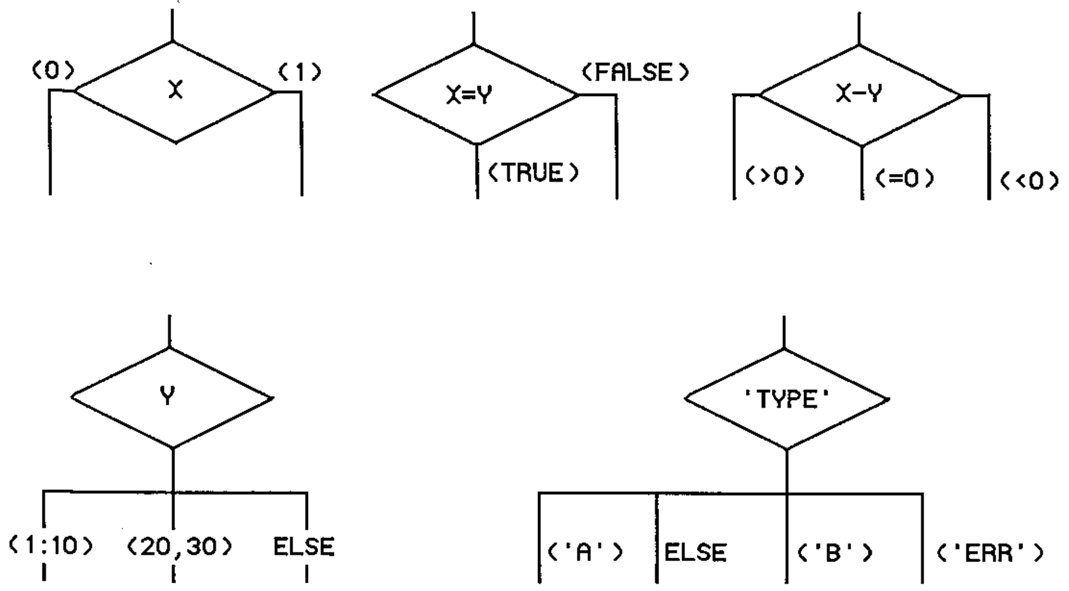


FIGURE D-3.8.46

Examples of decisions in SDL/GR

If an answer is leading back to the decision in the same transition some actions must be performed which influence the question in the decision. However, even with this rule infinite loops might be created as shown in Figure D-3.8.47. Care should therefore always be taken when having answers leading back to a decision in the same transition.

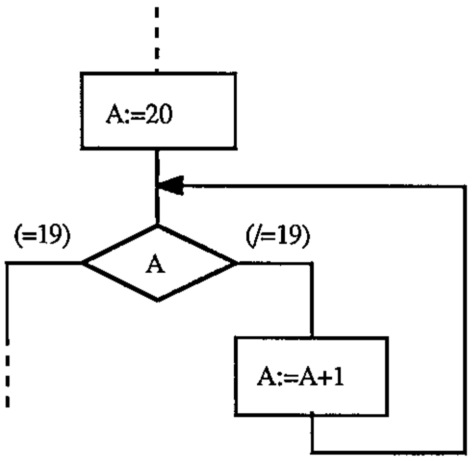


FIGURE D-3.8.47

Example of a legal use of a decision which creates an infinite loop

Decisions can be made using any values currently available to the process, including: values received by an input;

- values passed as actual parameter at process creation time;
- shared values.
- The expression in the question can include constants and any of the above kinds of values.

D.3.8.9 *Joins and connectors*

The joins allow control to be transferred from one point to another of a process body (as well as inside of a procedure body or inside of a service body).

In SDL/PR they are equivalent to "GO TO" statements. Labels are used as entry points associated to statements as shown in figure D-3.8.48. Inside a process body (or procedure body) it is not possible to transfer control (and therefore associate labels) to the type of statements shown in figure D-3.8.49. Labels are always local to a process therefore it is not possible to transfer control from a process to another by means of a join.

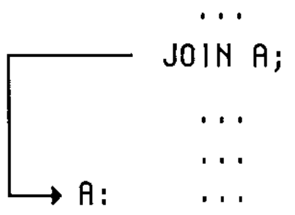


FIGURE D-3.8.48

Label

STATE,
ENDSTATE,
INPUT,
SAVE,
ENDDECISION

FIGURE D-3.8.49

Non-allowable label points

In SDL/GR the joins correspond to connectors (out-connectors and in-connectors). They may be used to split the diagrams, due to the lack of space, or also to avoid the crossing of flowlines which would make diagrams somewhat unclear. Besides, it is normally preferable to draw an SDL diagram with the flow from the top to the bottom of the page.

In GR any flowline may be broken by a pair of associated connectors, with the flow assumed to be from the out-connector to the in-connector. Each connector symbol contains a name, associated connectors have the same name. For each name only one in-connector exists but there may be one or more out-connectors.

It is desirable in GR that the page reference for the appropriate in-connector should be specified for the out-connector, and that the page reference(s) for the appropriate out-connector(s) should be given for the in-connector. (See example in Figure D-3.8.50)

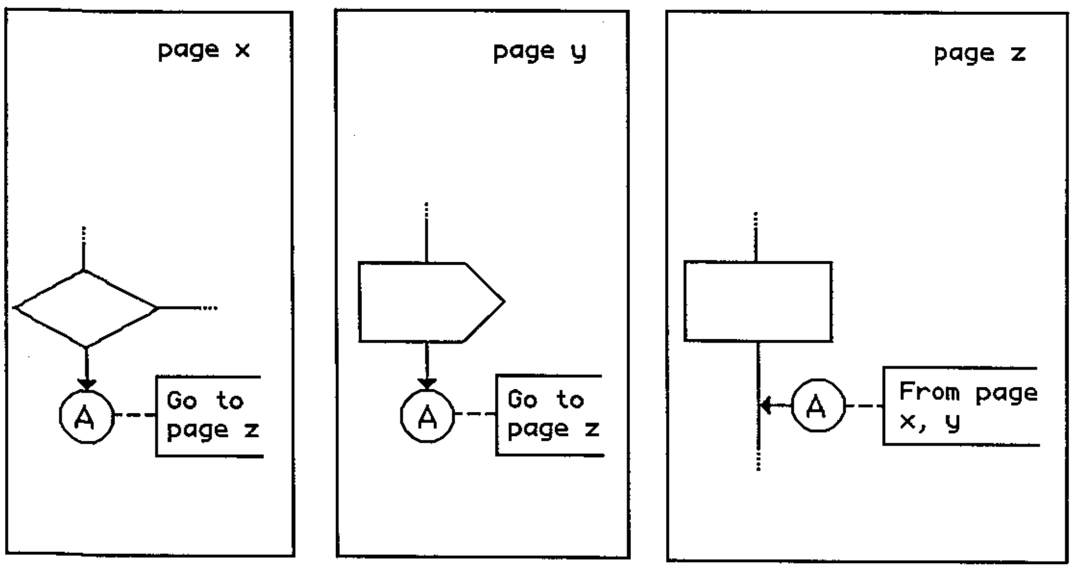


FIGURE D-3.8.50

Page references for connectors

D.3.9 *Procedures*

Procedures in SDL are similar to procedures in CHILL and other programming languages. Procedures are intended to:

a) permit the structuring of a process into several levels of detail;

b) maintain the compactness of specifications by allowing a complex assembly of items which may be regarded in isolation, to be represented by a single item;

c) allow commonly used assemblies of items to be defined and used repeatedly.

A procedure definition may only be contained in a process definition, in a service definition, or in a procedure definition and therefore a procedure is visible only to the process or procedure in which it is defined.

A procedure definition consists of the following items (some of which are optional):

- Procedure name.
- Procedure formal parameters: a list of variable names associated to their sorts. They are used to transfer information to/from the procedure at calling time. Procedure parameters can be passed by value (IN parameter) or by reference (IN/OUT parameter). If a parameter is passed by value the specification of the formal parameter defines a variable local to the procedure; if it is passed by reference the specification defines a synonym for the variable.
- Procedure definitions: procedures that can be called just by the procedure itself.
- Data definitions: the specification of datatypes local to the procedure.
- Variable definitions: local variables within the procedure.
- Procedure body: the specification of the actual behaviour of the procedure in terms of states and actions (similarly to the process body).

A partial example of procedure definition in SDL/PR is given in Figure D-3.9.1 (the keywords of the language are in upper-case letters). Note that the formal parameters with no explicit attributes have an implicit IN attribute (var5 in the Figure).

```
PROCEDURE prcd1;
   FPAR IN/OUT var1,var2 sort1,        ⎫ Procedure
        IN var3 sort2,                 ⎬ formal parameters
        IN/OUT var4 sort3, var5 sort4; ⎭
   PROCEDURE ...                       ⎫ Procedure definitions
   PROCEDURE ...                       ⎭
   ...data definitions ...             } Data definitions
   DCL ...                             } Variable definitions
   ...procedure body ...               } Procedure body
ENDPROCEDURE prcd1;
```

FIGURE D-3.9.1

Partial example of procedure definition in SDL/PR

D.3.9.1 *Procedure body*

The procedure body is very similar to the process body with the following differences:

- The procedure terminates its interpretation with a "return" instead of a "stop". In SDL/GR the return statement is represented by the keyword RETURN.
- In SDL/GR the procedure start symbol is slightly different from the process start symbol.
- The procedure start and return symbols can be found in the SDL/GR Summary.

A procedure may use the join construct, but only to refer a label within itself. Join may neither be used either to enter a procedure from the outside nor to leave it.

In SDL/GR a procedure definition is represented by means of a procedure diagram that is very similar to the process diagram. A procedure diagram consists of the following items:

- An optional frame symbol: a rectangular shaped symbol which contains all the other symbols.
- The procedure heading: the keyword PROCEDURE followed by the procedure name and by the specification of procedure formal parameters. Usually the procedure heading is put in the upper left corner of the frame or, if there is not a frame, in the upper left corner of the media upon which the diagram is drawn.
- An optional page numbering. (Put in the upper right corner)
- Text symbols: in the case of a procedure diagram a text symbol can be used to contain the specification of formal parameters, data and variable definitions.
- Procedure references: procedure symbols, each containing a procedure name representing a local procedure separately defined.
- Procedure diagrams: to directly define local procedures.
- The procedure graph area: the specification of the procedure behaviour in terms of start, states, inputs, outputs, tasks, ... and directed arcs.

In Figure D-3.9.2 there is an example of procedure definition in SDL/GR. The referenced procedure "TERM_P" in the example is local to the calling procedure.

As noted in the case of process diagrams (§ D.3.8), if there is not enough room in a single page for a procedure diagram the diagram can be represented in several pages repeating the frame symbol with the heading and page numbering.

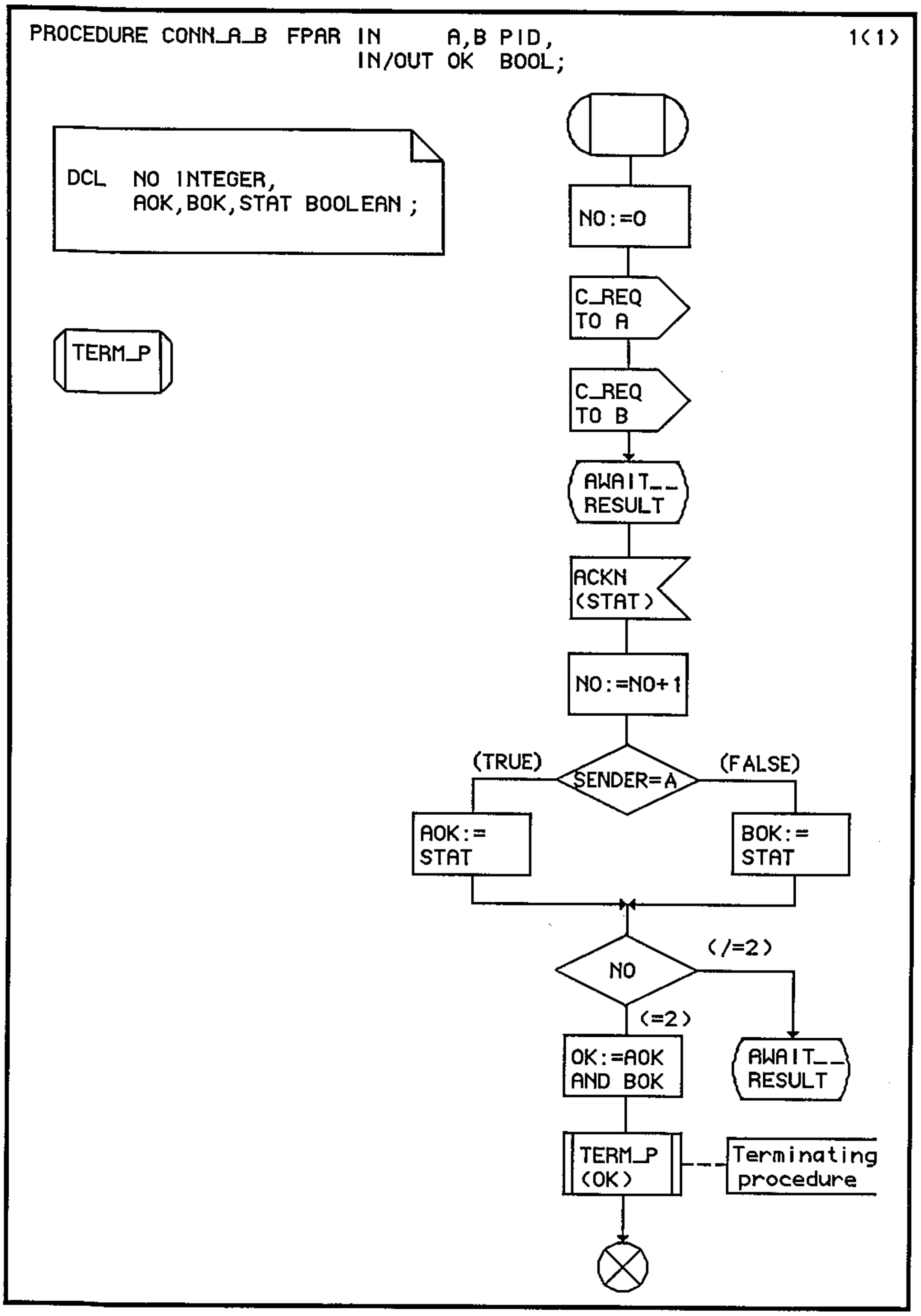


FIGURE D-3.9.2

Example of procedure diagram

D.3.9.2 *Procedure call*

Procedure calls may occur wherever a task is allowed in either a process or procedure graph. In some sense, a procedure can be interpreted as a task with the following exceptions:

1) A procedure may contain states and, if so, will receive signals.

2) A procedure can send signals. The originating process instance is the one which called the procedure.

When a procedure is called, the procedure environment is created and the procedure begins to be interpreted. Interpretation of the procedure continues until the RETURN is reached. While the procedure is being interpreted all signals addressed to the process are either implicitly saved or explicitly handled by the procedure. The procedure does not have its own input queue, but uses the input queue of the process that called it.

In SDL/PR a procedure call is represented by the keyword CALL followed by the procedure identifier and the list of actual parameters within round brackets. If a parameter is not given, it must be indicated by two consecutive commas. In this case the corresponding formal parameter has the value "undefined". Note also that the declaration of IN, or IN/OUT is made in the procedure definition, so that it must not be repeated by the calling statement. Some examples of call in SDLIPR are shown in Figure D-3.9.3.

```
CALL  proced1;
CALL  proced2(var1,var2);
CALL  proced3  (5,a+b,Pid1);
CALL  proced4  (W,X,,Z);
```

FIGURE D-3.9.3

Examples of call statements in SDL/PR

In SDLIGR a procedure call is represented by means of a procedure call symbol containing the procedure name and the list of actual parameters within round brackets. An example of call in SDL/GR is provided in Figure D-3.9.2.

D.3.10 *Data handling*

D.3.10.1 *Variable declarations*

Variables are local to a process instance, which means that all variables have one and only one owning process instance. Only the owner process instance can change the value of the variables.

The variables declared in Figure D-3.10.1, are local to each instance of process P, and therefore they can only be accessed and modified by each instance of the process P (each process instance can access or modify its own copy of variables).

```
SYSTEM S;
    . . .
    BLOCK B;
    . . .
        PROCESS P(3,10);
            DCL
             A Integer,
             D Integer;
             . . .
             . . .
        ENDPROCESS P;
    ENDBLOCK B;
    . . .
ENDSYSTEM S;
```

FIGURE D-3.10.1

Example of variable declaration

A variable can be initialized directly after declaration as shown in Figure D-3.10.2.

```
DCL A Integer :=1;
```

FIGURE D-3.10.2

Variable initialization

SDL allows two ways of initializing variables. It is possible to declare an initial value for all the variables of a certain sort using the DEFAULT statement in the data type definition (see § D.6.4.5). In this case the initialization is valid, for all the variables of that sort.

On the other hand, it is possible to initialize each variable of a certain sort by a value as shown in Figure D-3.10.2. If there is both a DEFAULT statement and an initialization on declaration then the latter prevails. If a variable is not initialized, its initial value is considered "undefined" in the system.

Of course the shorthand notation of variables initialization shown in Figure D-3.10.2 is only usable for simple variables, or for data types that allow a compact concrete notation for variables (another example is shown in Figure D-3.10.3).

```
PROCESS P1;
    NEWTYPE S Struct
        I Integer;
        B Boolean;
    ENDNEWTYPE;
    DCL
     A S := (. 1,True .);
    . . .
ENDPROCESS P1;
```

FIGURE D-3.10.3

Another example of variable initialization

Figure D-3.10.3 shows that the Struct sort has a shorthand concrete notation to indicate a structure value. In the case of an array generator it is suggested to explicitly initialize the variables simulating a "while construct" in the initial transition string of the process (see Figure D-3.10.4).

```
PROCESS P;
        NEWTYPE Arr1 Array (Nat1,Integer)
        ENDNEWTYPE Arr1;
        SYNTYPE Nat1 Natural CONSTANTS 1:3
        ENDSYNTYPE Nat1;
        DCL
         A Arr1,
         I Natural;
        START;
        TASK I:=1;
   Lab1: DECISION I<=3;
            (True): TASK A(I):=I;
                    TASK I:=I+1;
                    JOIN Lab1;
            (False): ;
        ENDDECISION;
        . . .
ENDPROCESS P;
```

FIGURE D-3.10.4

A more complex example of variable initialization

D.3.10.2 *Revealed/viewed variables*

Two processes can exchange information by other means than by signals. A process can access the value of a variable owned by another process by means of the VIEW operation. There are, however several rules to note:

- Both processes must belong to the same block.
- The process performing the VIEW operation must specify the identifier of the viewed variable in the view definition.
- The process revealing the variable must declare it with the REVEALED attribute.
- The sort identifier (or syntype identifier) in the variable declaration and in the view definition must be the same.

The value the viewing process obtains via the VIEW operation is the same as the revealing process obtains via ordinary access.

As the viewing process does not own the viewed variable, it can not modify its value. Of course the viewed value can be assigned to a variable the viewing process owns itself.

The SDL user may find that the definition of revealed/viewed variables permits an easy way of specifying communication between two processes. However, a number of problems arise in implementing systems so specified, and this section is intended to guide users in avoiding and overcoming such problems. Describing systems in SDL which have implemented revealed/viewed value is less difficult, because the problems will have been overcome in the implementation and it should be possible to map the solution chosen onto SDL.

In the remainder of this section it is supposed that the process R (Revealer) owns and reveals variables, and process V (Viewer) refers to them in its view definition.

An attempt to view variables in the process V before the process R is created results in an SDL error. The user may avoid this problem in two ways. Either:

- ensure that the revealing process instance R is created and has initialized the relevant variables before the viewing process instance V, or
- ensure that V does not enter transitions which use revealed/viewed variables until R has been created and has initialized the relevant variables.

A simple way to achieve the former case is to make R the parent (or an ancestor) of V (as in the example of Figure D-3.10.5), or to let R be created at the same time as the system (implied creation). In the latter case it may be arranged that the relevant transition in V can only be triggered by a signal from R.

Variables of R cannot be viewed after R has stopped. Any attempt to view data would then be an SDL error. The user may avoid this problem in two ways. Either:

- do not use a stop in R at all, or
- ensure that V is aware that R is about to stop and does not make any further attempts to view the relevant data.

The first solution has the disadvantage for the implementor that R has not released the data storage it was using.

An example of of revealed/viewed variables can be found in Figure D.3.10.5.

D.3.10.3 *Exported/Imported values*

A process can declare one or more of its variables as "exportable" which has the effect that all other processes (no matter the block they belong to) can import a copy of the value of the variable upon request. The importing process must declare the variable in its import definition.

When the exporting process executes an export the value of the variable is copied in an implicit variable. An importing process obtains, by means of the import expression, the value of this copy. So the value obtained by the import expression can differ from the value obtained by ordinary access by the owner, even if they are executed at the same moment.

An example is shown in Figure D-3.10.5.

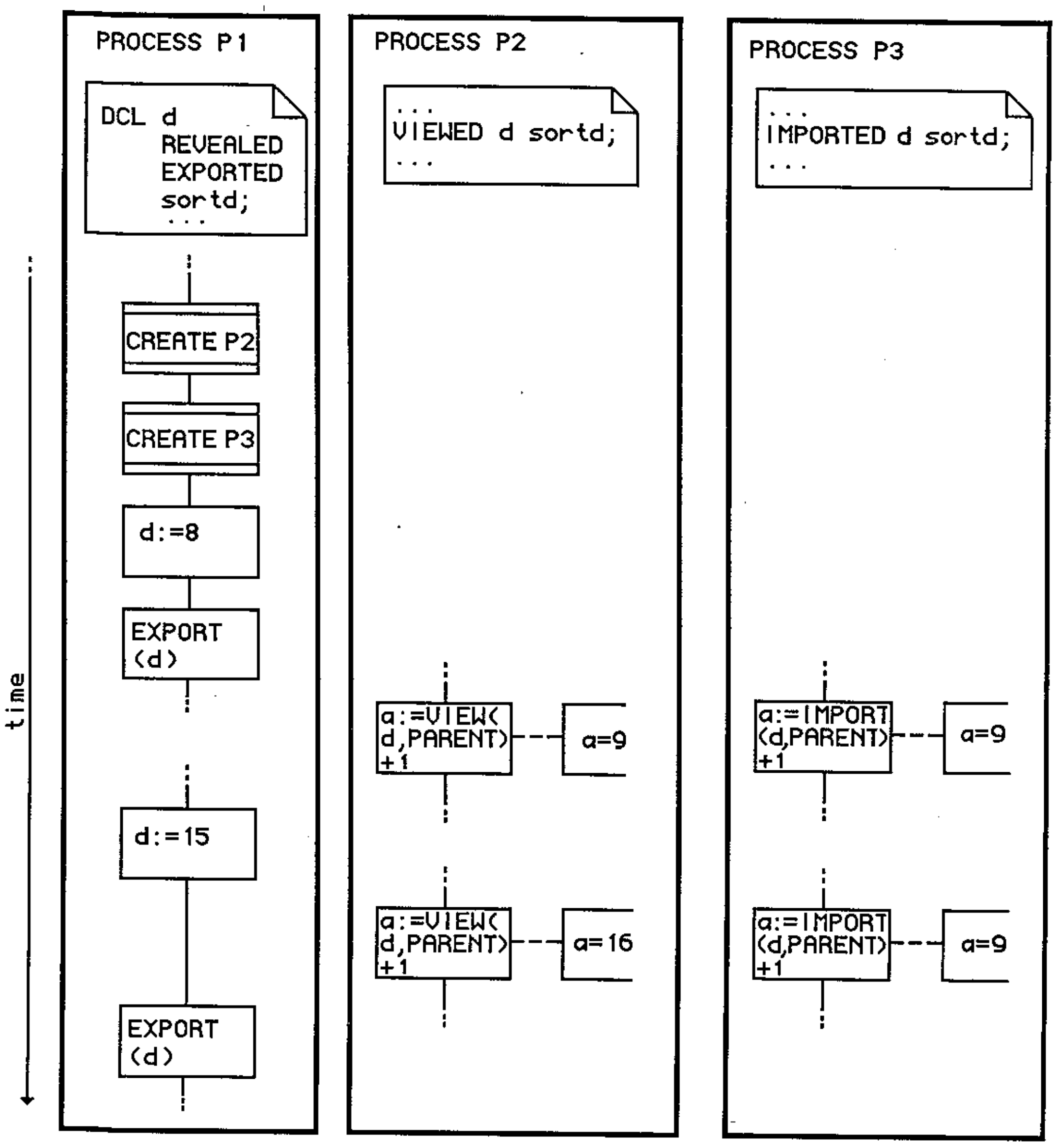


FIGURE D-3.10.5

Example of differences between use of revealed/viewed and exportable variables

In Figure D-3.10.5 process P1 is supposed to be the parent of processes P2 and P3, therefore the process instance identifier in the IMPORT and in the VIEW expressions is indicated by the attribute PARENT.

D.3.10.4 *Expressions*

Expressions in an SDL process can be used as formal text in decisions, alternatives, selects, tasks, continuous signals, enabling conditions, and set constructs. Expressions are also used as actual parameters of the output, of the procedure call, of the creation construct. Pid expressions are used in the TO part of the output construct. The expressions in the alternative and select constructs (statically evaluated) should be of predefined data sorts. In an expression there can be ground terms(i.e. terms containing only representations of constant values), and terms with variables.

SDL has a predefined set of infix operators. These operators can be used for any data type, and they are the only allowed infix operators. For these operators the precedence rules are predefined too and cannot be changed. The predefined infix operators are:

=>, OR, XOR, AND, IN, /=, =, >, <, <=, >=, +, -, // *, /, MOD, REM

SDL provides also the predefined operators:

-, NOT

which are unary prefix operators.

Using the predefined operators, care should be taken as far as the precedence rules are concerned because these operators, being predefined independently from the domain of application, may mislead.

For example consider the newtype and the expression in Figure D-3.10.6. Thinking of the precedence rules of multiplication and addition one evaluates the expression as A=>((B*C)+D). But this is different from the precedence of AND and OR and such a change may lead to confusion. Furthermore this kind of change may lead to inconsistent axioms and invalidate the SDL specification.

```
NEWTYPE Newbool
    INHERITS Boolean
    OPERATORS ( "+"="AND", "*"="OR", "=>" )
ENDNEWTYPE Newbool;

. . .

A=>B*C+D
```

FIGURE D-3.10.6

Example of misleading notation in an expression

All the other operators defined by the user are functions and must be used in the prefix notation.

The two operators VIEW and IMPORT have a particular semantics explained in the previous paragraphs. Anyway, they give back a value of a certain sort that can be another operand in an expression,

D.3.11 *Expressing time in SDL*

The need to measure time and request timeouts in a system is met by timers and a set of operations performed upon them.

In the SDL model "Timers" are meta-processes that are able to send signals to the process upon request. The use of timers must be declared in the timer definition within process definitions. The operations "SET" and "RESET" are used to activate timers. The SET operation requests a timeout to occur at a specified time, and the RESET operaration cancels the specified timeout. (Note that a SET operation includes implicitly a RESET operation of any not expired timings from that timer).

The set construct contains the time expression of the requested delay, the name of the involved timer and, optionally, a list of expressions. The list of expression specifies values that will be contained in the timer signal in the same order.

The list of expressions may be specified in the reset construct to reset a particular instance of the timer instance having the same values.

The timer definition must include the list of the corresponding sort-reference identifiers for the sorts used in the expressions in set/reset.

An example of Set statement is shown in Figure D-3.11.1.

In the set construct an absolute time must be specified. Relative time is transformed to an absolute time value by adding the primitive function "NOW" that represents the current time. The expression of the requested delay should be a Time expression. Time is a predefined sort, inherited from the Real sort.

The possibility of receiving a timeout is specified by means of the timer name in an input as shown in Figure D-3.11.1.

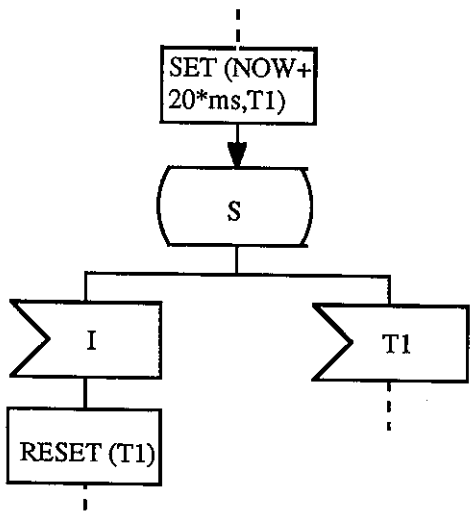


FIGURE D-3.11.1

Example of the use of timers

It is possible to define Synonyms to indicate the desired Durations. Once chosen the Time and the Duration units, the user can define the Synonyms needed to represent the Durations as shown in Figure D-3.11.2.

```
PROCESS p;
    TIMER T1 subscriber_id;
    . . .
    DCL Sa subscriber_id;
    . . .
    SYNONYM
        Sec  Duration = 1000.0,
        Min  Duration = 60000.0,
    . . .
    START;
        . . .
        SET  (NOW +20*Min +30*Sec , T1 (Sa) );
        . . .
        RESET  (T1 (Sa));
        . . .
ENDPROCESS p;
```

FIGURE D-3.11.2

Synonyms for representing durations

In the Recommendation it is stated that the setting of a timer to an already 'passed' time is allowed. This decision was taken to ease the simulation of systems, however, since such a setting might result in an unclear specification this usage should be avoided.

D.3.12 *Qualifiers usage*

In SDL qualifiers are used for referring items in a specification, when the name does not determine the item uniquely. Of course when defining the item only the name must be specified, but when referring to it outside its defining occurrence an identifier composed by a qualifier and the name may be needed.

This also applies when remote definitions are used: the defining occurrence uses the name when referencing the definition; the remote definition uses a qualified name to specify its context.

In Figure D-3.12.1 there is an example of qualifiers usage for a process that can receive two different signals having the same name but different identifiers! The first input refers to a signal type defined at block level; the second input refers to a signal type defined in the process definition. In the second input the qualification might be omitted because when identifiers are not qualified the innermost definition is referred to.

```
SYSTEM  s;
    . . .
    BLOCK  b;
        SIGNAL  x;
        . . .
        PROCESS  p;
            . . .
            SIGNAL  x;
            . . .
            STATE  wait;
                INPUT    SYSTEM  s/BLOCK b    x;
                    . . .
                INPUT    PROCESS p    x;
                    . . .
            ENDSTATE  wait;
            . . .
        ENDPROCESS p;
    ENDBLOCK b;
ENDSYSTEM s;
```

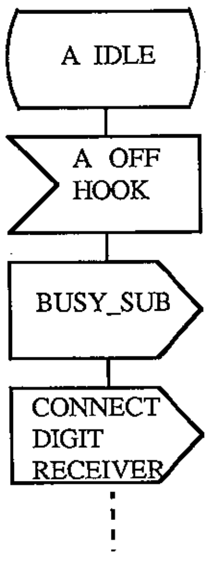
FIGURE D-3.12.1

Example of qualifiers usage

D.3.13 *Syntax of names*

Names in SDL may either consist of a single word or may consist of a list of words separated by delimiters (spaces or control characters). The second possibility allows a more readable specification when long names are used, especially when using SDL/GR as graphical symbols have a limited size. Some examples of names consisting of several words are shown in Figure D.3.13.1.

```
. . .
STATE A IDLE;
    INPUT A OFF HOOK;
      OUTPUT BUSY_SUB;
      OUTPUT CONNECT
               DIGIT
                RECEIVER;
      . . .
```



SDL/PR SDL/GR

FIGURE D-3.13.1

Examples of names consisting of several words

Whenever the usage of several words in a name is ambiguous (see examples in Figure D.3.13.2), the delimiters between two words must be substituted by an underline character ("_").The string of characters obtained by transforming the delimiters in underlines still denotes the same name; so, for example, BUSY SUB denotes the same name as BUSY_SUB. In Figure D.3.13.3 there are some examples of unambiguous usage of names.

```
a)      A := BLOCK B1 B / PROCESS P1 P C;
b)      DCL A B C D;
c)      NEWTYPE X Y Z (PAR) ENDNEWTYPE;
```

FIGURE D-3.13.2

Examples of ambiguous usage of names

```
a)      A := BLOCK B1_B / PROCESS P1_P C;

b)      A := (BLOCK B1 B) / ( PROCESS P1_P C);

c)      DCL A B_C_D;

d)      DCL A_B C_D;

e)      DCL A_B_C D;

f)      NEWTYPE X Y_Z(PAR) ENDNEWTYPE;

g)      NEWTYPE X_Y Z(PAR) ENDNEWTYPE;
```

FIGURE D-3.13.3

Examples of unambiguous usage of names

It is important to note that the underline character can also be used in names as a continuation character, to allow the splitting of names over more than one line. In this case, a name containing an underline character followed by one or more delimiters can also be denoted by eliminating both the underline and delimiters.

Examples of different denotations for the same name are given in Figure D.3.13.4

```
a)      CONNECT DIGIT RECEIVER

b)      CONNECT_DIGIT_RECEIVER

c)      CONNECT DIGIT_RECEIVER

d)      CONNECT
        DIGIT
        RECEIVER

e)      CONNECT DI_
        GIT RECEIVER

f)      CONNECT DI_ GIT RECEIVER

g)      CONNECT_
        _DIGIT_
        _RECEIVER
```

FIGURE D-3.13.4

Examples of different denotations for the same name

D.4 *Structuring and refining SDL systems*

D.4.1 *General*

This chapter will discuss some techniques and SDL constructs that allow a top-down specification of large systems. Usually the terms "partitioning" and "refinement" are used for these techniques with the following meanings:

- Partitioning: subdivision of a part of a system into smaller parts whose global behaviour is equivalent to the unpartitioned part. It can be applied to blocks (structuring them in new subblocks, channels and subchannels), to channels (structuring them in blocks, new channels and subchannels) and to processes (structuring them in services).
- Refinement: addition of new details to the system functionalities. Seen from the environment the refinement of a system causes an enrichment in its behaviour as more kinds of signals and information can be handled.

Note that the internal structure of a system part provides more details about the structure, not necessarily more details about the behaviour of the system. Conceptually it is possible to distinguish the aspect of a more detailed representation of the behaviour (e.g. the handling of a new signal) from the aspect of a more detailed structure but in practice the two aspects are usually merged together, so that with new details about the system structure we also provide new details on the system behaviour.

The minimum structure of a system in SDL is what is described in Chapter 2 of the Recommendation, i.e. a system consists of a set of blocks connected with channels – and the blocks contain processes.

Concepts for partitioning in several levels of detail and signal refinement are covered by Chapter 3 in the Recommendation. For systems not needing further partitioning such concepts are not necessary.

D.4.2 *Criteria for partitioning*

The technique of starting with a high level view of a system representation and breaking it down into manageable pieces is called partitioning. This process of partitioning adds structure to a system.

The criteria leading to the partition of the system representation are several, including:

a) to define blocks or processes of intellectually manageable size;

b) to create a correspondance with actual software and/or hardware divisions;

c) to follow natural functional subdivisions;

d) to minimize interaction between blocks;

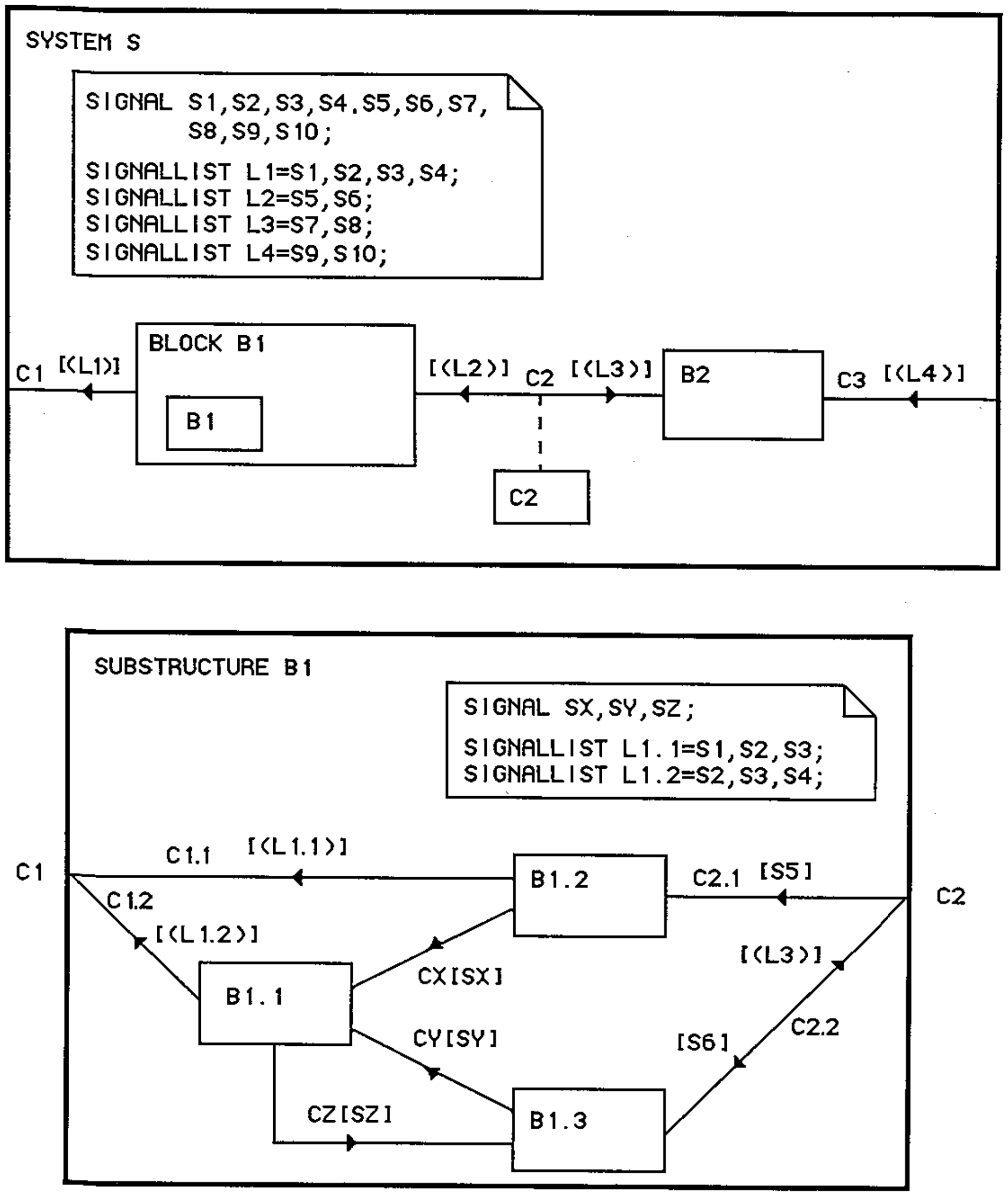
e) to reuse already existing specifications (e.g. a signalling system);

The actual criteria adopted may depend on a number of factors including the degree of detail required.

Since the relationship between levels will depend on the chosen partitioning criteria, it is important to state clearly which criteria have been chosen in order to allow easy comprehension of the representation. The partitioning criteria depend upon the user but some limitations exist to ensure a correct representation in SDL. These will be discussed in the following paragraphs.

D.4.3 *Block partitioning*

A block can be partitioned into a set of blocks and channels in almost the same way that a system is partitioned into blocks and channels. In SDL/GR this is represented by means of a block substructure diagram. An example of block substructure diagram is shown in Figure D-4.3.1. In the first diagram of the Figure there is the block diagram of block B1 containing a reference to its substructure. In the second diagram there is the substructure diagram for the block B l. The block symbol attached to the channel C2 by a dashed line in the first diagram represents a reference to the substructure of the channel C2 (See § D.4.5).


```
SIGNAL S1,S2,S3,S4.S5,S6,S7,
       S8,S9,S10;
SIGNALLIST L1=S1,S2,S3,S4;
SIGNALLIST L2=S5,S6;
SIGNALLIST L3=S7,S8;
SIGNALLIST L4=S9,S10;
```

```
SIGNAL SX,SY,SZ;
SIGNALLIST L1.1=S1,S2,S3;
SIGNALLIST L1.2=S2,S3,S4;
```

FIGURE D-4.3.1

Block partitioning

In SDL/PR the partitioning of a block is represented by a set of definitions between the keywords SUBSTRUCTURE and ENDSUBSTRUCTURE inside the block definition. The definitions in a block substructure are the same we have in the system definition; besides the specification of the connections between channels and sub channels must be provided as shown in Figure D-4.3.2.

```
SUBSTRUCTURE B1;

    SIGNALLIST L1.1 =S1,S2,S3;
    SIGNALLIST L1.2 =S2,S3,S4;

    SIGNAL SX,SY,SZ;

    CHANNEL C1.1
        FROM B1.2 TO ENV WITH (L1.1);
    ENDCHANNEL C1.1;
    CHANNEL C1.2
        FROM B1.1 TO ENV WITH (L1.2);
    ENDCHANNEL C1.2;
    CHANNEL C2.1
        FROM ENV TO B1.2 WITH S5;
    ENDCHANNEL C2.1;
    CHANNEL C2.2
        FROM ENV TO B1.3 WITH S6;
        FROM B1.3 TO ENV WITH (L3);
    ENDCHANNEL C2.2;
    CHANNEL CX
        FROM B1.2 TO B1.1 WITH SX;
    ENDCHANNEL CX;
    CHANNEL CY
        FROM B1.3 TO B1.1 WITH SY;
    ENDCHANNEL CY;
    CHANNEL CZ
        FROM B1.1 TO B1.3 WITH SZ;
    ENDCHANNEL CZ;

    CONNECT C1 AND C1.1,C1.2;
    CONNECT C2 AND C2.1,C2.2;

    BLOCK B1.1 REFERENCED;
    BLOCK B1.2 REFERENCED;
    BLOCK B1.3 REFERENCED;

ENDSUBSTRUCTURE B1;
```

FIGURE D-4.3.2

SDL/PR for the example of Figure D-4.3.1

Inside a block definition the specification of processes and of a block substructure may coexist. In this case the processes of the block represent the behaviour of the block at a certain level of detail, other processes inside the definitions of sub-blocks will represent the same behaviour in a more detailed way. An example of blocks described both in terms of processes and substructures can be found in D.4.7 (Figure D-4.7.1).

If in a block there are no processes but only the block substructure, and if the block substructure is not referenced, a shorthand notation is provided in SDLIGR to simplify the diagram. Such a notation allows the nesting of blocks by considering the external block frame as implying the substructure frame. By means of this notation the example of Figure D-4.3.1 can be drawn as in Figure D-4.3.3.

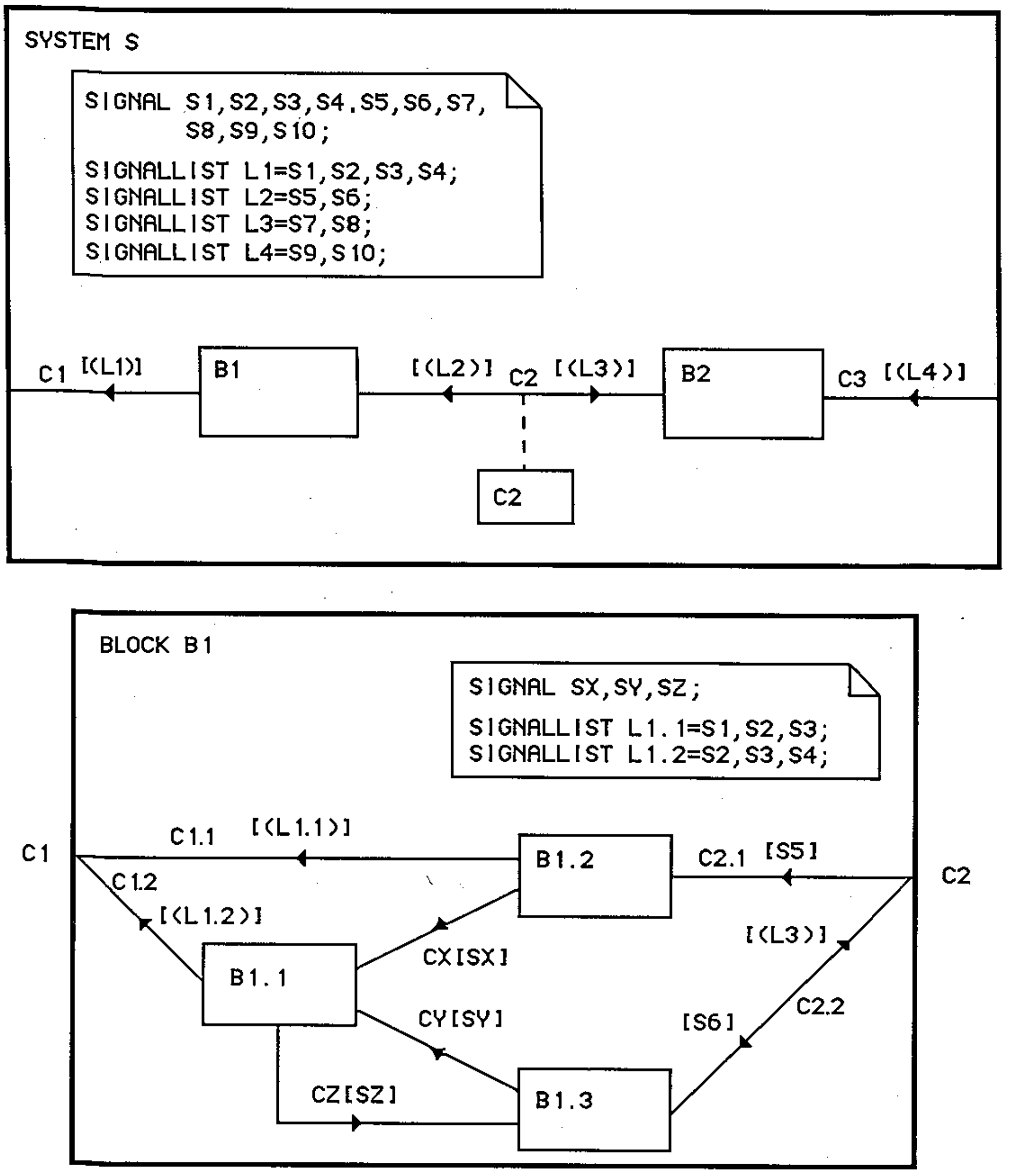


FIGURE D-4.3.3

SDL/GR shorthand notation for block partitioning

Each block deriving from the partitioning of a block can be partitioned itself resulting in a hierarchical tree structure of blocks and their subblocks. An auxiliary diagram called "block tree diagram" showing such a general structure is explained in § D.4.4.

Unless signal refinement is involved, the partitioning must obey the following rules:

1) The subchannels connected to an incoming channel must contain no new signals in their signal lists and their signal lists must contain all the signals in the original channel list (For bidirectional channels the incoming path list must be considered). Thus for the example shown in Figure D-4.3.1, C2.1 and C2.2 transport on the incoming paths all the signals in L2. In addition, no signal transported by C2.1 can appear in the incoming path of C2.2.

2) The subchannels (e.g. C1.1 and C1.2) connected to an outgoing channel (e.g. Cl) must contain no new signal names in their signal lists and their signal lists must contain all the signal names in the original channel. Thus L1.1 and L1.2 contain all the signal names in L1. The lists L1.1 and L1.2 may contain the same signal identifiers.

3) If the original block contains processes, two options are available. Firstly, a copy of each process can be redefined in one or other of the new subblocks. Secondly new processes can be defined in the subblocks in such a way that the interface remains unchanged.

4) Data definitions in the parent block are available to its subblocks, so that each of them can use a data type defined in the parent block without having to redefine it.

5) If a datatype defined in the parent block is redefined with the same name in a subblock, the new definition applies to the defining subblock while the old one holds for the other subblocks. A redefinition made just for the sake of characterizing a subblock should be discouraged, because it may be overlooked by a reader who will assume the old definition as valid. In some cases such a redefinition should be made, particularly when a refinement in the behaviour is involved. Care should be taken to emphasize this redefinition by appropriate annotation.

D.4.4 *Block tree diagram*

A block tree diagram represents the structure of a system in terms of blocks and sub-blocks. The intention with the diagram is to give the reader an overview of the total structure of a system.

The diagram is a hierarchical tree of block symbols and "partitioned into" lines as shown in Figure D-4.4.1.

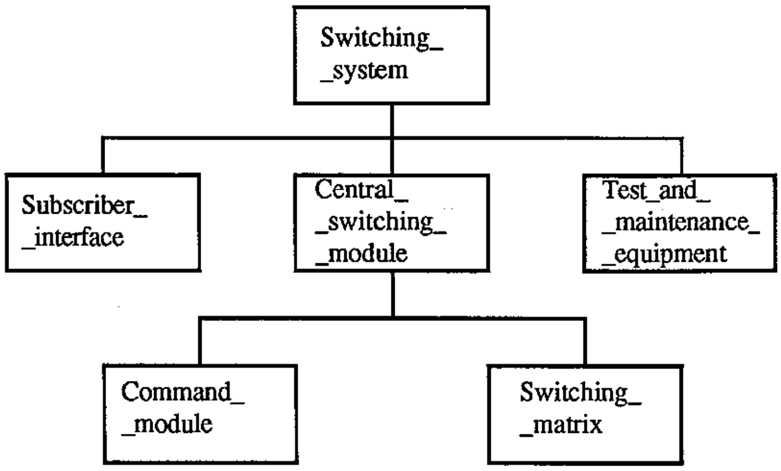


FIGURE D-4.4.1

Example of block tree diagram

The diagram should preferably be drawn with all the block symbols having a uniform size. This allows the blocks at the same level of partitioning to appear as a uniform level in the diagram. If the diagram is so large that it requires more than one page it should be partitioned into "partial" block tree diagrams as shown in Figure D-4.4.2.

It is often useful to partition a block tree diagram into partial diagrams.

The splitting into several partial diagrams is done so that the first diagram, having the system as root, is chopped off so that a set of further partitioned blocks appear as not partitioned. The blocks, where the original diagram was chopped off, appear as roots in diagrams showing the further partitioning.

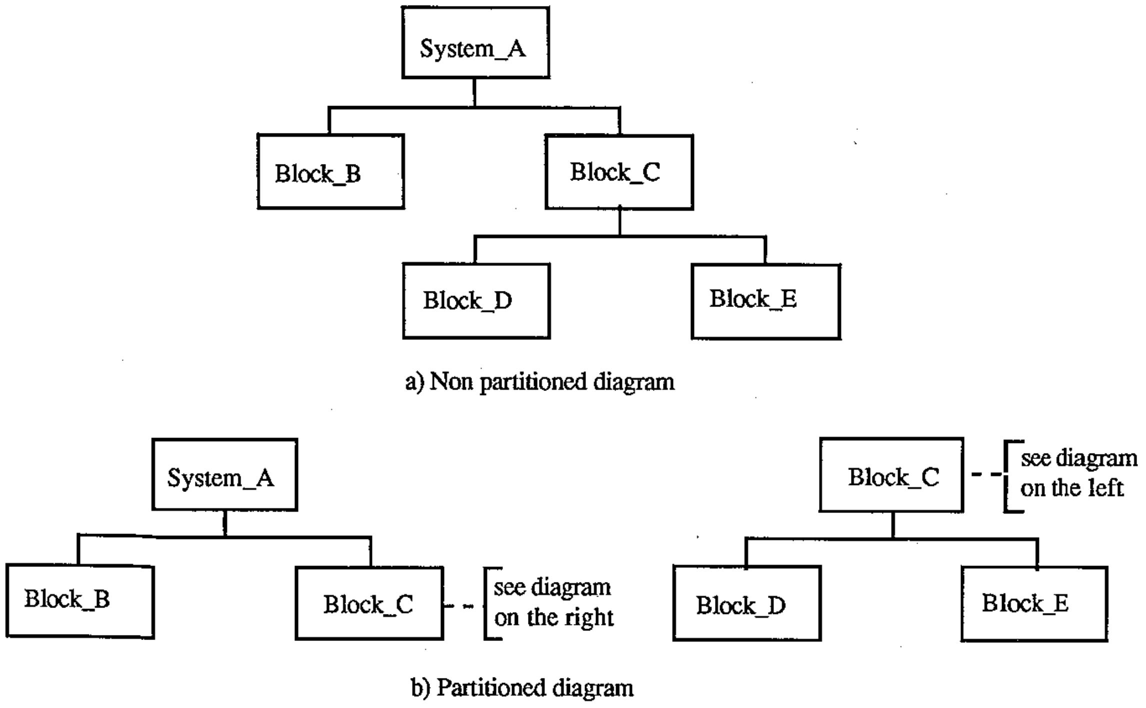


FIGURE D-4.4.2

Example of partial block tree diagrams

If partial diagrams are used, and it is not obvious that a block is further partitioned and/or where to find the continuation diagrams, references should be inserted using the comment symbol.

D.4.5 *Channel partitioning*

A channel can be partitioned independently of the blocks it connects. This allows for the representation of the behaviour of the channel. To get an exact representation of the way in which a signal is conveyed, it may be in some cases necessary to represent the channel behaviour. This is made by considering the channel as an item of its own having as environment the two blocks it connects. Looking at the channel in this way, we can show its structure in terms of blocks, channels and processes.

In SDL/GR the channel partitioning is represented by means of a channel substructure diagram as shown in Figure D-4.5.1. (The example represents the substructure of the channel C2 of Figure D-4.3.1).

A channel substructure diagram describes how a channel is partitioned into sub-components. The diagram resembles the system diagram (except for the connections to blocks). All the guidelines given in § D.4.3 are also valid for the channel substructure diagram.

In the block interaction diagram where the partitioned channel appears there should be a reference to the channel sub-structure diagram describing the partitioning.

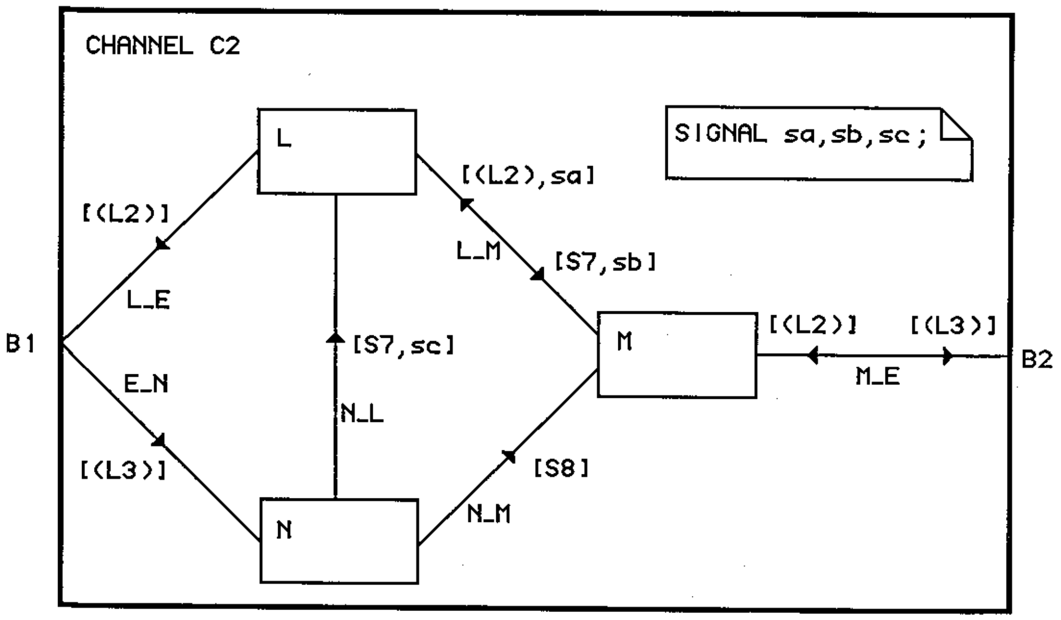


FIGURE D-4.5.1

Example of channel substructure diagram

In SDL/PR the form is similar to the form of the block substructure definition, the only difference is that in the CONNECT statement the endpoint subchannels are connected to the external blocks. (B1 and B2 in Figure D-4.5.2) and not to external channels.

```
SUBSTRUCTURE C2;

     SIGNAL sa,sb,sc;

     CHANNEL L_E
          FROM L TO ENV WITH (L2);
     ENDCHANNEL L_E;

     CHANNEL E_N
          FROM ENV TO N WITH (L3);
     ENDCHANNEL E_N;

     CHANNEL M_E
          FROM M TO ENV WITH (L3);
          FROM ENV TO M WITH (L2);
     ENDCHANNEL M_E;

     CHANNEL L_M
          FROM L TO M WITH S7,sb;
          FROM M TO L WITH (L2),sa;
     ENDCHANNEL L_M;

     CHANNEL N_M
          FROM N TO M WITH S8;
     ENDCHANNEL N_M;

     CHANNEL N_L
          FROM N TO L WITH S7,sc;
     ENDCHANNEL N_L;

     CONNECT B1 AND L_E,E_N;
     CONNECT B2 AND M_E;

     BLOCK L REFERENCED;
     BLOCK M REFERENCED;
     BLOCK N REFERENCED;

ENDSUBSTRUCTURE C2;
```

FIGURE D-4.5.2

SDL/PR for the example of Figure D-4.5.1

Import/export of values is allowed between a block and a channel substructure. This allows a straightforward representation of the OSI model, where peer layer communication is modelled through signal interchange and contiguous layer communication through shared values.

D.4.6 *System representation in case of partitioning*

In the cases where a system is represented as a set of blocks interconnected by channels, and where the behaviour of each block is expressed by one or more processes, we have a single level representation. This means that we can see all the representation elements at the same level. When we introduce partitioning, we insert a hierarchial relation between the various documents. We will have a document containing the representation of the structure of the system where the system is composed of "n" blocks.

A different document may present the system as composed by a different set of blocks some being derived from the blocks contained in the previous document (some blocks in the previous document have been substituted by the subblocks obtained by the partition of the blocks). This latter document must be related to the previous one.

It is not just a matter of relating the documents to each other to obtain a complete representation of the system, but also they should be organized in such a way that it is possible to access the system representation by levels, starting with a general overview and moving to more and more detailed representations. This implies the grouping of the various documents so that they form various levels of system representation.

Not all the levels should contain the same elements. At a first level the system representation may consist of the block and channel representations without including the processes describing the behaviour of each block. At a lower level we may wish to include the representation of the behaviour of some blocks but not the one of others. The lowest level of representation (the most detailed one) should include the complete representation of the behaviours of all the blocks, i.e. the complete set of processes expressing this behaviour.

Looking at the block tree we can make several considerations.

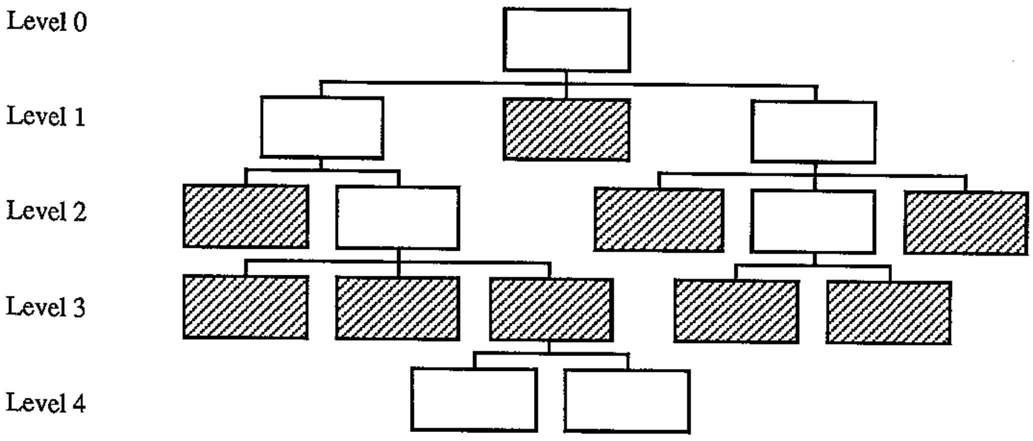
First of all the tree always has one and only one root, the system. This is so, even in the cases where a system is represented from the beginning as composed of several blocks. In the block tree these blocks will be represented at, for instance, level 1. The root may consist of the system name only. The channel definitions may be given for the blocks at level 1 even though this is not required unless the blocks contain processes.

A second point comes up looking at the leaves of the tree: not all of them have to be at the same level. This may be the result of a different number of partitioning on the blocks of the tree. The number of partitionings depends on several aspects, most of which are the subjective judgement of the specifier/designer.

SDL requires that a leaf block does not need further partitioning only if its behaviour is completely specified (i.e. the processes definitions associated are sufficient to represent its behaviour). Consequently a leaf block must contain at least one process.

When a system representation is given on several abstraction levels, we can select any of these levels to represent the system. The selection of a certain level implies the consideration of the blocks at that level, of their associated processes and of all the blocks which are leaf blocks at upper levels together with their associated processes (see Figure D-4.6.1).

It is often convenient to select different levels for different purposes, e.g. an overview level for presentation and a more detailed level for implementation.



Note - The representation level 3 is shown by means of striped boxes.

FIGURE D-4.6.1

Representation level 3

A system representation at a certain level may be incomplete in the sense that some of the blocks at that level do not have associated processes.

The reasons for talking about representation levels and of selecting a certain level of representation are:

- We may have reached in our design a certain "level" of detail, which is represented by that level of representation (in this case the blocks at that level are leaf blocks and they axe not complete because further work is needed!).

- We want to look at the system representation at a certain level of detail; and therefore we select that level of representation which corresponds in the best way to the abstractions we are looking for. Note that in some cases a certain representation level may consist of documents having different levels of abstraction. A part of the system may be represented in a very detailed way at level two, whilst another part may be still abstract at level four. This means that when we select a representation at level three we can have very detailed parts together with parts that can only be considered as overview.

- The representation/design methodology can be such that each level has a precise meaning, e.g. level one corresponds to the specification, level two provides the overall system structure, level three provides the module structure (racks, software functions), level four provides the detailed structure (printed boards, procedures, software modules). In this case the selection of a certain level corresponds to a certain reader's need; in this case it should be noted that the methodology will avoid the situation of discrepancies in the level of details of parts composing a certain representation level.

D.4.6.1 *Consistent partitioning subset*

In addition to the total system specification and to the one given by levels, SDL has the concept of "consistent system subset". It can be considered as a single level specification where all the blocks can be taken from any level of the system structure given that:

- A block can be chosen to be part of the consistent system representation if it can be considered as a leaf block (either it is a leaf block or it has associated all the processes required to represent its behaviour).
- If a block is chosen all the blocks obtained from the partitioning of its parent block should be included either directly or by including their sons.
- All the documents defining the signals flowing in the channel connecting a block in the representation should be provided. This may imply, depending on the partitioning strategy chosen, that once a block has been taken, its parent should also be taken at least for the parts defining the data and the signals.

In cases where some channels have been partitioned, considering them as systems the specification of each of these "systems" has to be provided. Their specification consists of the same kind of documents as the usual system representation. Notation should be added to relate these systems to the main system specification they belong to. Thus, in a sense, we may consider these partitioned channels as inner systems. Each of these systems can have several levels of representation and may also have inner systems if some of the channels contained in them are further partitioned.

D.4.7 *Refinement*

The refinement mechanism has been introduced in SDL to "hide" low level signals for higher levels of abstraction and to allow a top-down specification of the behaviour of the system.

The refinement allows the user to partition signals into subsignals resulting in a hierarchical structure as in the case of blocks and sub-blocks. That is, inside a signal definition it is possible to define a set of new signals which are said to be refined signals or sub-signals of the signal being defined. Like the block tree for a block definition, a signal tree can, for explanatory purposes, be drawn for a signal definition.

The refinement is tightly related to the block partitioning because only those signals transported by a channel connected to a partitioned block can be refined. That is, a signal contained in the list of a channel may be replaced by its sub-signals when the connected block is partitioned. The correspondent subchannels generated in the partitioning of the block will specify the sub-signals in their signal lists.

When a signal is defined to be carried by a channel, the channel automatically will be carrier for all the subsignals of the signal, even if some of the subsignals are flowing in the opposite direction (in that case the channel is regarded as implicitly bidirectional).

An example of refinement is given in Figure D-4.7.1. This example represents a system in which a process in one block is sending text files to a process in another block. At the highest refinement level it is achieved by sending signals each representing a text file (signal sf). On the next refinement level we want to specify that such a text file consists of a number of records which are sent one by one (signal sr) and that the receiver has to answer back (signal nr) after consumption of each record. The sender will send an end-of-file signal at the end. In this example at the highest refinement level processes in Bl and B2 are communicating using the signal sf; on the next lower level processes in B11 and B21 are communicating using the signals sr,nr and eof.

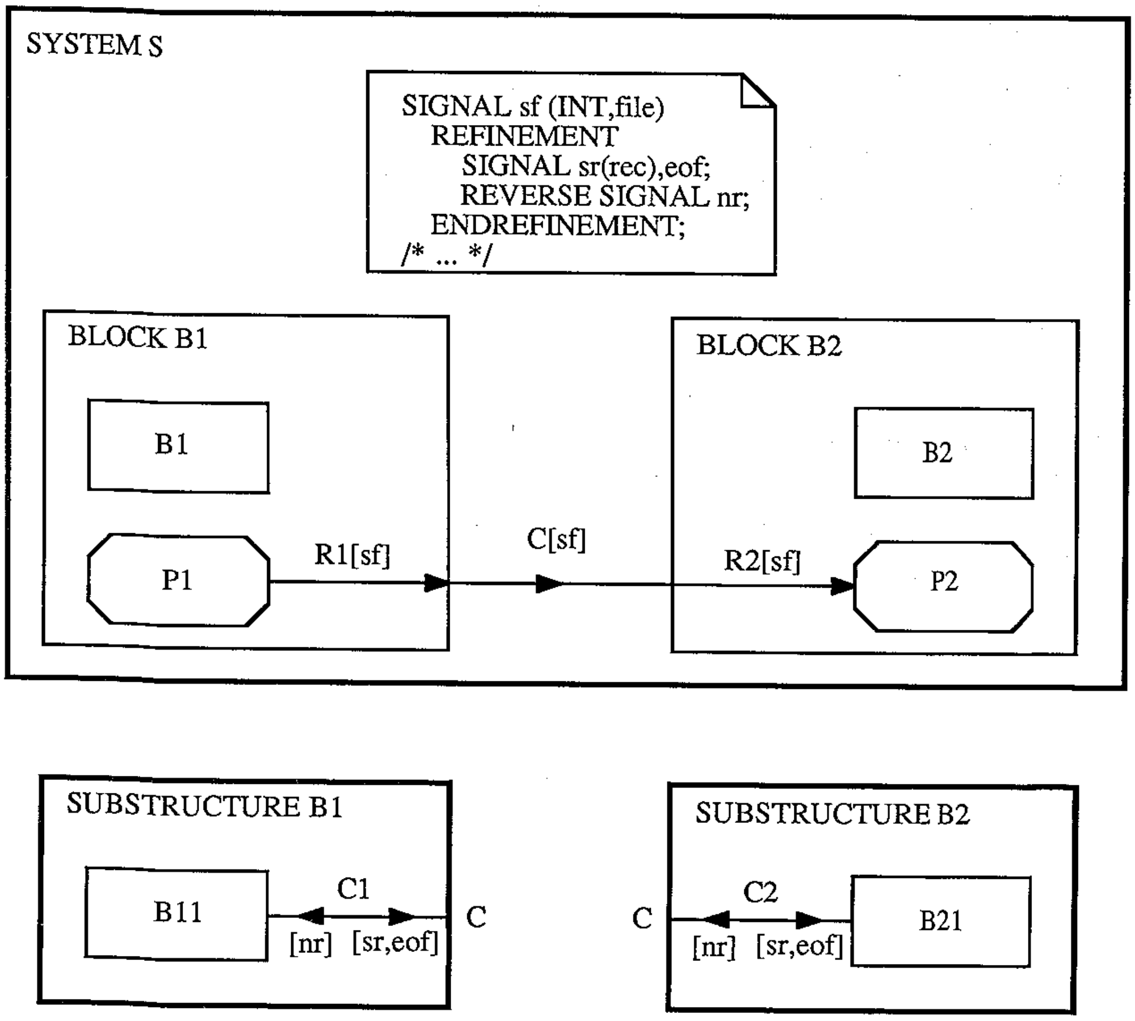


FIGURE D-4.7.1

Example of refinement

Some real cases in which the refinement concept can be applied are the following:

- Name transformation: a signal is refined into another signal with a different name. This is a one to one transformation where just the signal name changes. This possibility is usually applied when we want to have each level completely understable on its own (it may be convenient to adapt the name of a signal to its context).
- Splitting transformation: It is a one to many transformation where one signal is split into several signals as a result of a characterization. For example a generic "Alarm" signal is split into "Register Alarm", "Central_Processor_Aiarm" and "Subscriber_Alann".
- Algorithm transformation: The original signal is transformed into a set of signals that activate an algorithm in order to provide the original information.

D.4.7.1 *Consistent refinement subset*

When refinement is applied in a system specification, the concept of consistent partitioning subset is restricted to avoid communications with different signals between different levels of refinement. In this case the system definition is said to contain several consistent refinement subsets. If a consistent refinement subset contains a process communicating with sub-signals, then the process may not communicate with the parent signal and the other end of the communication link must communicate with the same sub-signals.

D.4.7.2 *Transformation between signals and sub-signals*

The user may need to describe the transformation between different level of refinement for simulation purposes or to check the system behaviour at different levels of detail. This can be done in an informal way by means of additional SDL processes describing the dynamic transformation from a signal and its sub-signals or viceversa.

In Figure D-4.7.2, two processes are introduced to describe the transformation applied in Figure D-4.7.1. A refine process defines how the high level signal is refined into a set of signals on the next lower level. A retrieve process describes the reverse transformation.

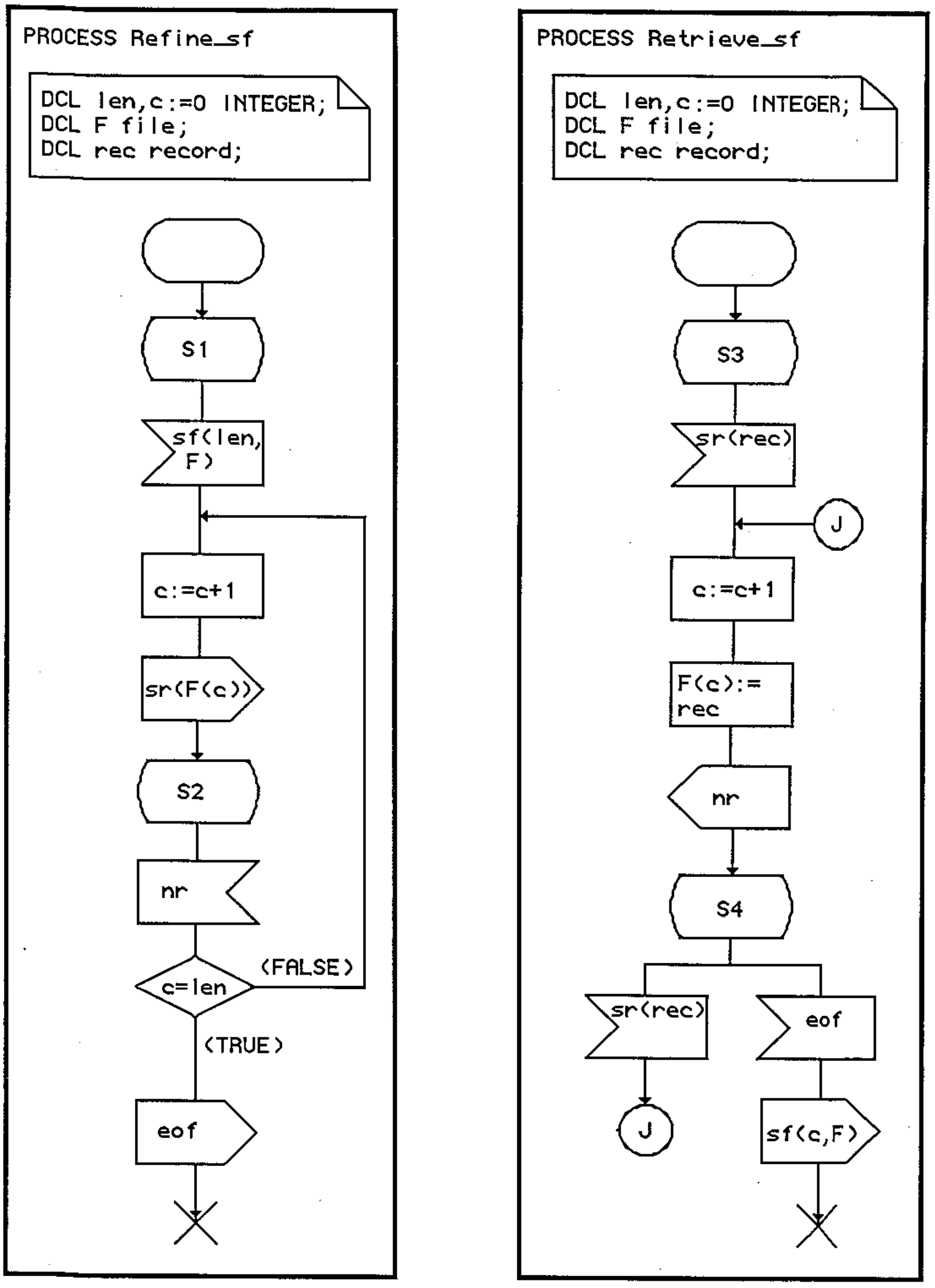


FIGURE D-4.7.2

Example of specification of a signal transformation

D.5 *Additional concepts*

D.5.l *Macros*

The macro construct is a means to handle repetition and/or to structure a description. It consists of a macro definition, containing a part of an SDL specification, that can be referenced (macro call) elsewhere in a system specification.

The macro definition can be given at all places where data definitions are allowed. However, the macro name has no scope. Thus, a macro defined within a block may be referenced outside that block.

In SDL/PR it is possible to have a macro definition replace any sequence of lexical units. This differs from SDL/GR macro definitions which may replace only collections of syntactical units.

To map SDL documents containing macros between SDL/GR and SDL/PR, the following restrictions on the use of SDL/PR macros must be applied.

1. A macro can only replace one or more of the following syntactic constructs: (These correspond to SDL/GR symbols.)

   start
   state
   input
   enabling condition
   continuous signal
   save
   action statement
   terminator statement

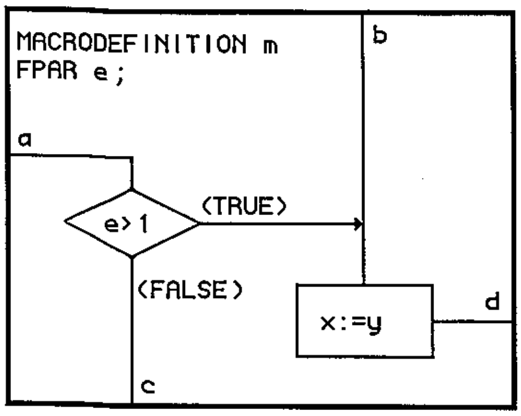
2. Macro formal parameters may not be used in places that determine the type of the SDL/PR construct. That is, it must not be used where the SDL/PR keywords would be used. Similarly actual parameters should not contain keywords which correspond to SDL/GR symbols: START, STATE, PROCEDURE, INPUT, TASK, OUTPUT, DECISION, CREATE, STOP, PROVIDED, CALL, COMMENT, JOIN, RETURN, SAVE or OPTION.

3. Every statement in the macro definition must be reachable from at least one macro inlet.

In SDL/PR, the macro always has at most one inlet and one outlet, so that it is necessary to use labels and joins to represent in SDL/PR a SDL/GR macro with more than one inlet or outlet respectively. If the macro is to be invoked in more than one place, the labels will have to be passed as parameters.

In SDL/GR there are two different ways to represent inlets and outlets of a macro definition. Either the user can draw a frame for the macro and connect the inlets/outlets to the frame or also can use explicitly inlets and outlets symbols (the macro frame is optional).

The example in Figure D-5.1.1 illustrates SDL/GR and SDL/PR for a macro with two inlets and two outlets. (In this example the inlets and outlets are connected to the macro frame).



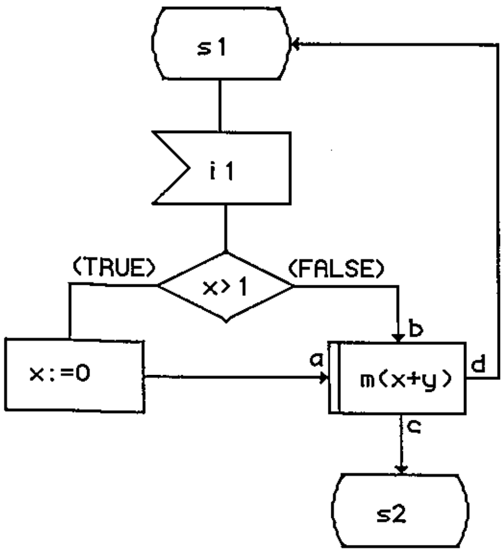
```
MACRODEFINITION m
   FPAR e,a,b,c,d;
   a: DECISION  e>1 ;
      (TRUE):  JOIN b;
      (FALSE): JOIN c;
      ENDDECISION;
   b: TASK x:=y;
   JOIN d;
ENDMACRO m;
```

SDL/GR SDL/PR

FIGURE D-5.1.1

Example of a macro definition

This means that in the main SDL/PR specification (Figure D-5.1.2) there are corresponding joins and labels. Note that the label will probably be different for each different macro call.



```
...
STATE s1;
   INPUT i1;
     DECISION  x>1 ;
     (FALSE):  JOIN BB;
     (TRUE):   TASK x:=0;
               JOIN AA;
     ENDDECISION;
     MACRO m (x+y,AA,BB,CC,DD);
     CC: NEXTSTATE s2;
     DD: NEXTSTATE –;
 ...
```

SDL/GR SDL/PR

FIGURE D-5.1.2

Example of a macro call

Figure D-5.1.3 shows two macro definitions which define a syncronization mechanism. Note the use of the pseudo-formal parameter MACROID to make unique state names. The same example is replied in Figure D-5.1.4 making use of inlets and outlets symbols.

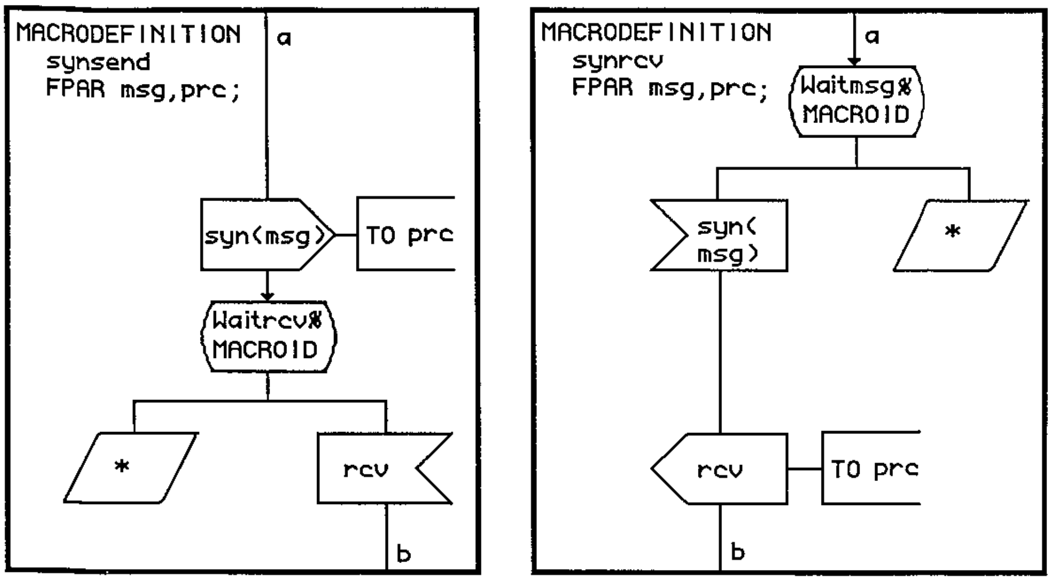


FIGURE D-5.1.3

Example of two macros using MACROID

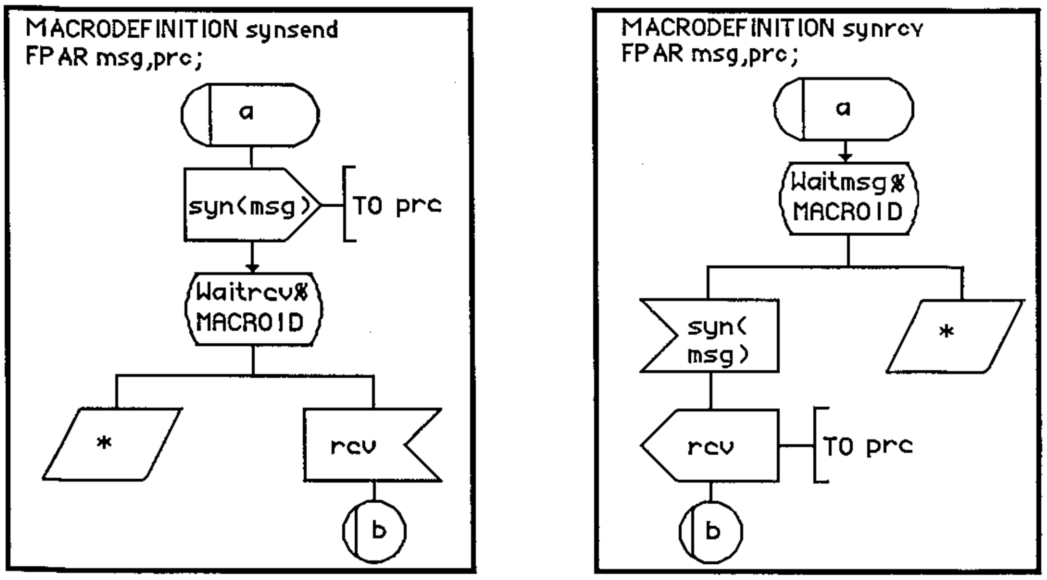


FIGURE D-5.1.4

Example of the use of inlet and outlet symbols

In the case of nested macros, i.e. when there is one or more macro calls inside a macro definition, it should be noted that the expansion of the outermost macros does not affect the expansion of the innermost ones. More precisely the expansion of nested macros is to be considered as if the expansion of a macro should be completed before starting the expansion of possible macro calls inside.

D.5.2 *Generic systems*

In SDL it is possible to define different systems in a single specification, by means of the system parameters. The system parameters have an undefined value that can be provided externally to obtain a specific system definition according to the customer need.

System parameters are actually external synonyms and can be used in every place a synonym can be used. Of course before interpreting a system all the external synonyms must be assigned a value. Figure D-5.2.1 shows some valid examples of the use of external synonyms.

```
SYSTEM s;
    . . .
    SYN inst.numb INTEGER = EXTERNAL;
    SYN rate.incr  REAL = EXTERNAL;
    . . .
    BLOCK b;
        PROCESS p (inst.numb);   /* parametric instance number */
            . . .
            val := val + rate.incr ;  /* parametric increment */
            . . .
        ENDPROCESS p;
    ENDBLOCK b;
ENDSYSTEM s;
```

FIGURE D-5.2.1

Example of use of system parameters

SDL provides also two additional constructs to allow more powerful selections conditioned by external synonyms:

- SELECT construct: conditional selection of a piece of specification. The condition is specified by a boolean expression which must be evaluable statically, before system interpretation. The piece of specification is selected when the expression is TRUE.
- ALTERNATIVE construct: allows the conditional selection of one piece of specification, between two or more alternative pieces. It can be used only to select different transitions in the body of processes, procedures or services.

D.5.3 *Services*

D.5.3.1 *General*

The intention of the service concept is to offer the possibility of partitioning a process definition without introducing parallellism. Each service can be seen as a "function" offered by the process. It is a partial process definition representing a 'sub-behaviour' of the process. Such a 'sub-behaviour' is one part of the total process behaviour. Consequently using the service concept is a way of structuring a process.

There are many reasons for structuring a process e.g. to manage complexibility and increase readability. But partitioning is also a way to isolate certain parts of a process and describe them separately. Such parts can be 'sub-parts' of a function offered by the system. So by the service concept a system function can be isolated by describing a set of subordinate services in one or more processes.

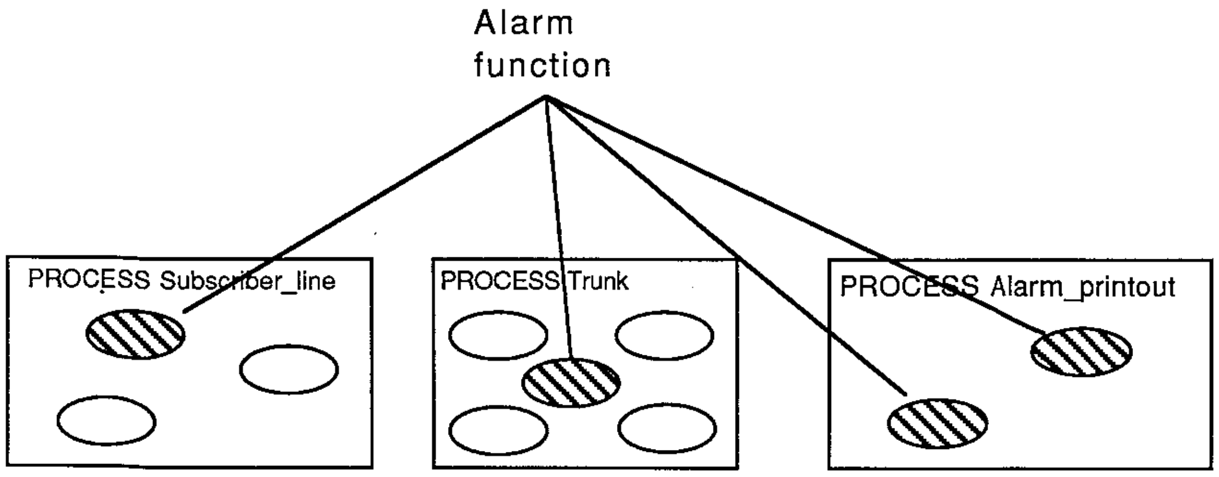


FIGURE D-5.3.1

Informal example showing how services in different processes can compose a superiour function in the system

Note that the language SDL does not have any facilities to compose a system function by picking services from a number of processes. Such a composition is up to the user and entirely outside the language.

Since a service is isolated from other services and described as a finite state machine with its own state space it may be developed and changed without affecting the other services. However it should be noticed that the services in a process often have common data which means that they may affect each other by data manipulation.

Since the behaviour of the process is not changed when it is partitioned into services, services are just an alternative description of the same behaviour. Of course one could at design time decide to model the behaviour in several processes instead of one process. However, this would give a different behaviour because several processes run in parallell and can not share (read and write) data without complicated signal interchange.

Note that structuring a process in services does not mean that the process will entirely disappear. It only means that the process body, describing the behaviour of the process, has been completely replaced by a number of service bodies. Formal parameters, declarations and definitions at process level will remain. In addition to these a number of local definitions and declarations may be included in the service definitions.

A service definition consists of the following items, some of which are optional:

- Service name
- Valid input signal set: a list of signal identifiers defining the signals that can be received by the service.
- Procedure definitions: the specification of the procedures which are local to the service. A procedure reference may also be used.
- Data definitions: the specification of user defined newtypes, syntypes, and generators local to the service.
- Variable definitions: the declarations of the variables, which are local to the service. These variables can not be revealed or exported to other processes (the keywords REVEALED and EXPORTED are not allowed). For each variable declared, the identifier of its sort must be specified. An initial value can be specified optionally.
- View definitions: the declaration of the variable identifiers that can be used to obtain the values of the variables owned by other processes. For each variable identifier the variable sort must be specified.
- Import definitions: the specification of variable identifiers, owned by other processes, which the service wants to import. For each identifier, the variable sort must be specified.
- Timer definitions: explained in § D.3.11.
- Macro definitions explained in § D.5.1.
- Service body: the specification of the actual behaviour of the service in terms of states, inputs, outputs, tasks etc. Compared to a process body, a service body can also contain sending and receiving of priority signals (§ D.5.3.2).

In SDL/GR a service definition is represented by means of a service diagram. A service diagram consists of the following items:

- An optional frame symbol: a rectangular shaped symbol which contains all the other symbols.
- The service heading: the keyword SERVICE followed by the service identifier. The service heading is placed in the upper left corner of the frame.
- An optional page numbering (put in the upper right corner).
- Text symbols: a text symbol is used to contain data, variable, view, import and timer definitions which are local to the service.
- Procedure references: a procedure symbol containing a procedure name representing a procedure, local to the service and separately defined.
- Macro diagram: see guidelines in § D.5.1
- Service graph: the specification of the actual behaviour of the service in terms of states, inputs, outputs, tasks etc. Compared to a process graph, a service graph can also contain sending and receiving of priority signals (§ D.5.3.2).

An example of the service concept is given in Figure D-5.3.2 for the simple process 'Timer'. The process is structured in two services which are referenced and defined in two service diagrams (Fig. D-5.3.3).

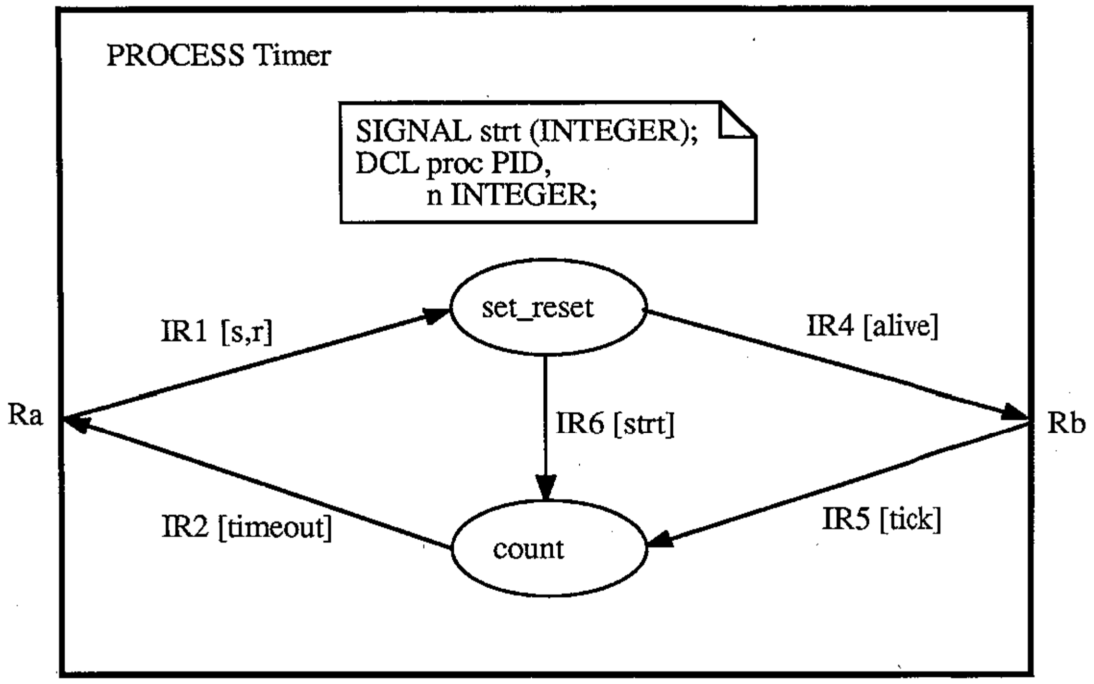


FIGURE D-5.3.2

Process diagram with referenced services

There is an internal interface between the two services by means of the signal route 'IR6' conveying a "priority" signal (§ D.5.3.2).

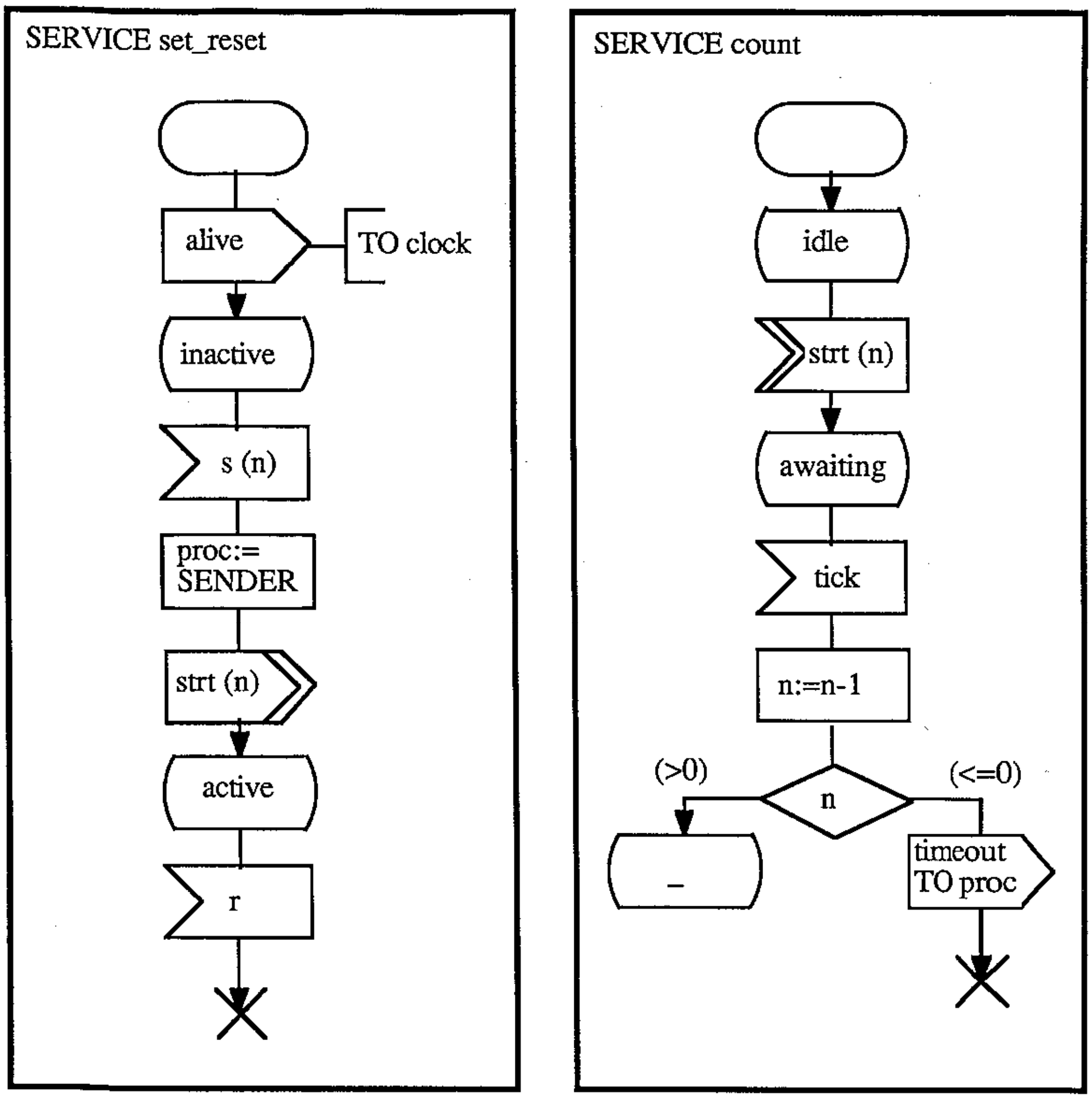


FIGURE D-5.3.3

Service diagrams

It should be noticed that, since actions (output) are included in the start transition in the first service, no actions are allowed in the start transition in the second service.

Another example of the service concept is given in § D.10.

D.5.3.2 *Priority signals*

A priority signal is used for communication between two transitions in different services in a process when no external signals (from other processes) are allowed to be consumed in the time gap between the transitions. In this way the transitions are 'concatenated'.

The following figure (D-5.3.4) is an illustration of concatenation of transitions when priority signals are used.

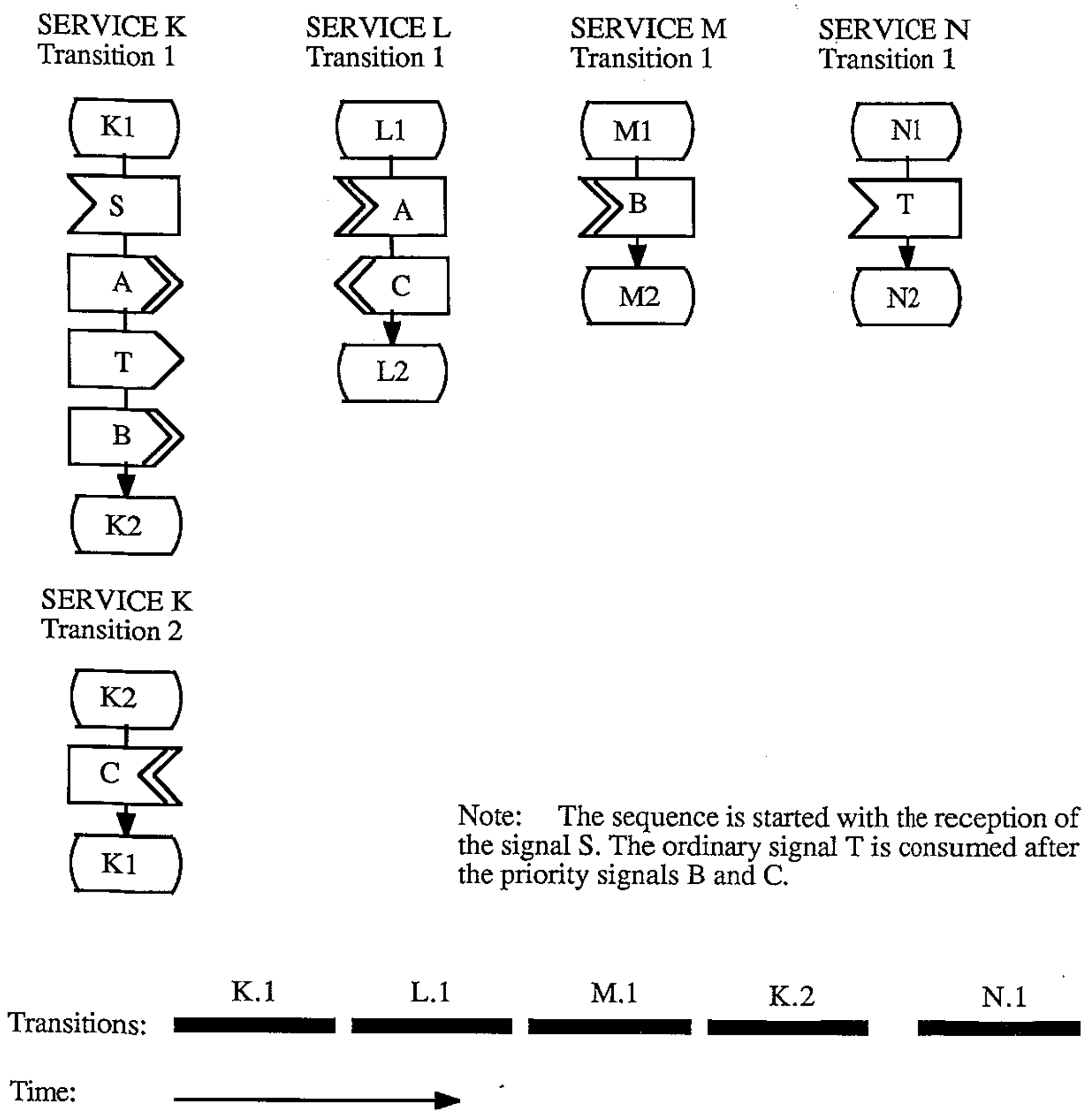


FIGURE D-5.3.4

Concatenation of transitions in different services in a process (informal SDL)

The construct to obtain priority signals using the ordinary input queue is explained in the recommendation. By sending the priority signals to a preliminary state, they can be dealt with as ordinary signals, causing transitions to a main state. (See also D.5.3.3.1).

The following example is an illustration of the consequences of using priority signals. The example is explained without using the "preliminary/main state" model.

The sequence chart (fig. D-5.3.5) is fetched from the interaction between the services in the process SUBSCRIBER_LINE' described in § D.10.2.

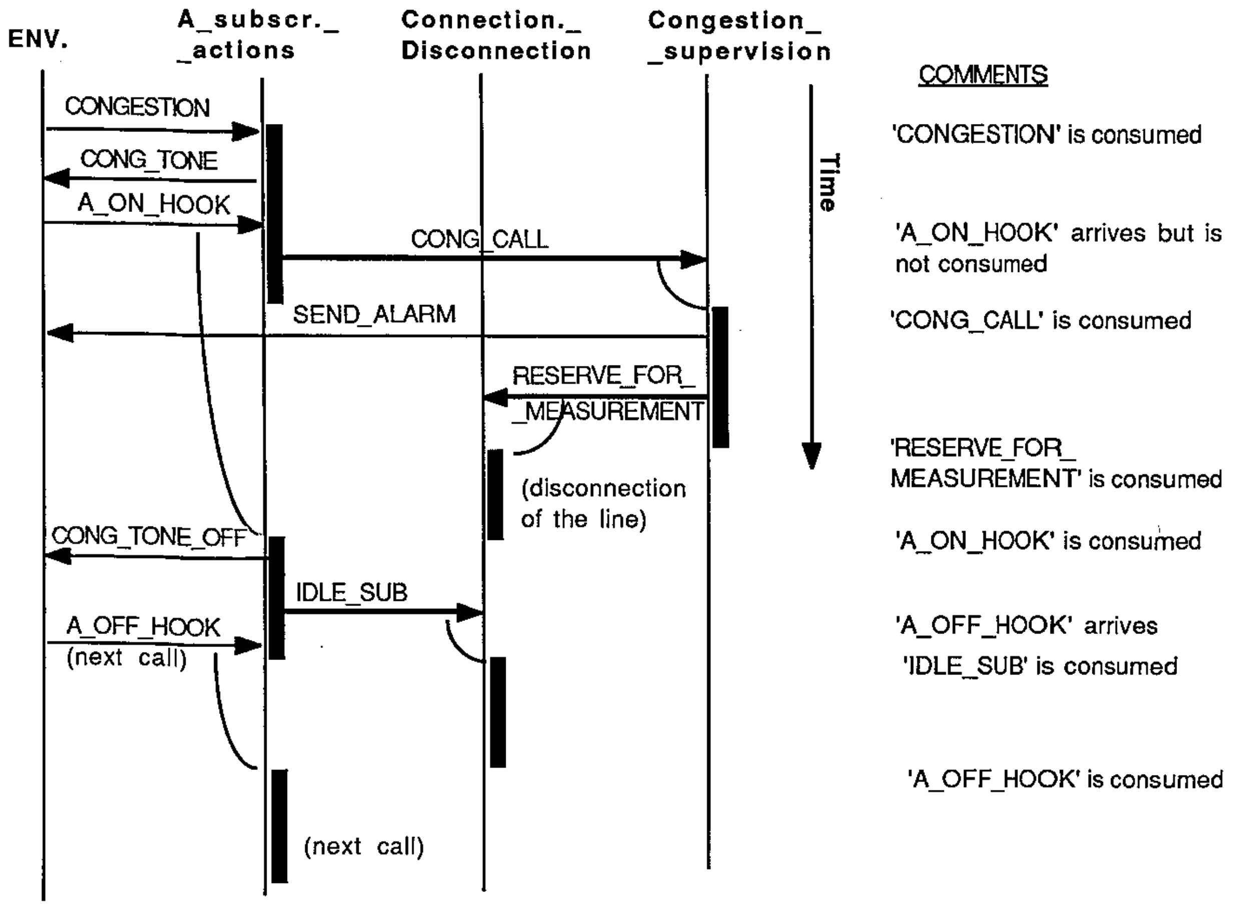


FIGURE D-5.3.5

Sequence chart showing how priority signals can change the order of the transitions. Each thick vertical line indicates execution of one transition

The chart contains both signals to/from the environment of the process and priority signals between the services. The priority signal arrows are thicker. The order of the signal arrows (top-down) indicates how the signals are placed in the queue. The vertical thick lines indicate how execution of the transitions are ordered in time.

The sequence starts with the signal 'CONGESTION' from the register meaning that the call must be rejected. It also means an immediate rejection of the next call.

The figure shows that a transition, activated by a priority signal e.g. 'RESERVE_FOR_MEASUREMENT, starts before a transition activated by an ordinary signal e.g. 'A_ON_HOOK', in spite of opposite queueing order. The transition activated by the signal 'RESERVE _ FOR_ MEASUREMENT' disconnects the subscriber line which means that the next call (signal 'A_OFF_HOOK') will be immediately rejected.

In the next sequence chart we have the same situation but the diagram does not use priority signals. The services interwork with ordinary signals.

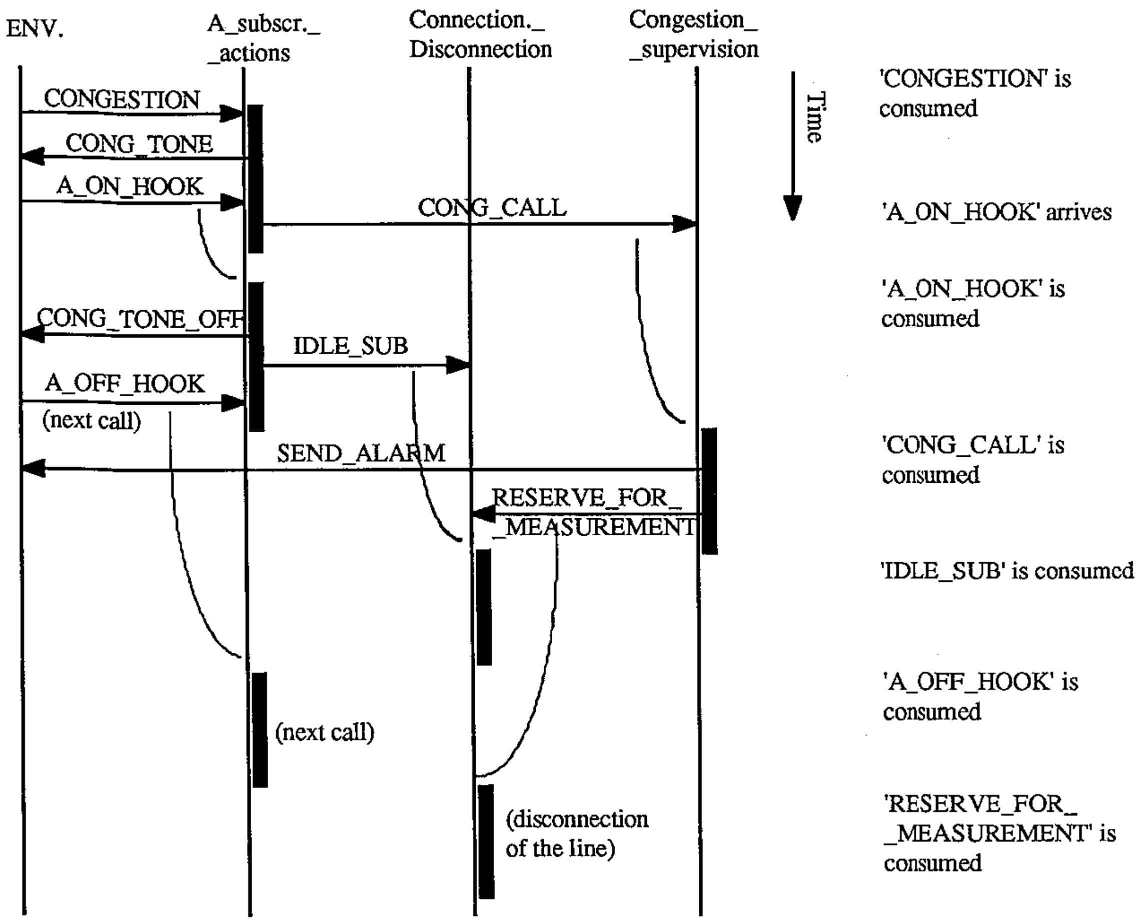


FIGURE D-5.3.6

Sequence chart with ordinary signals and normal order of the transitions

Compared with fig. D-5.3.5 we can see that the order of the transitions has been changed. The next call arrives before the subscriber line is disconnected which means that the call will not be immediately rejected.

D.5.3.3 *Transformation*

The service and the priority signal are additional concepts because they are composed (modelled) from more basic SDL concepts like process and signal. To be able to interpret the service and the priority signal semantically they must be transformed back to these basic concepts. Normally such a transformation need not be performed by the user , at least not manually, but the user must of course be aware of the transformation. In a tool for semantic checking of the service the transformation could be automatically performed.

After the transformation the services have disappeared and they are replaced by a extended process definition which can be semantically interpreted. The transformation has defined the behaviour.

D.5.3.3.1 *Transformation of states*

In the recommendation specific rules for the state transformation are given.
The result of these rules is that the number of states in the process is the product of the number of states in the services.

In addition to this using of priority signals will double this product because every transformed state is split in a preliminary state and a main state. This doubling of the number of states due to priority signals are not dealt with in the following example of state transformation.

In many cases a lot of states in the "transformed" process are not relevant and the state space can be reduced. This is discussed in the following example from § D.10.2.

In the process 'SUBSCRIBER_LINE' we have 4 services, 'A_subscriber_actions', 'B_subscriber_actions, 'Connection.Disconnection' and 'Congestion supervision'. The number of states is 10, 5, 3 and 2 i.e. in total 20.

Applying the rules for state transformation on these services we will have 300 states (10*5*3*2) in the process which means 300 name-tuples. The dimension of the tuple is 4 (4 services). The positions in the tuple are arranged according to the following figure (D-5.3.7).

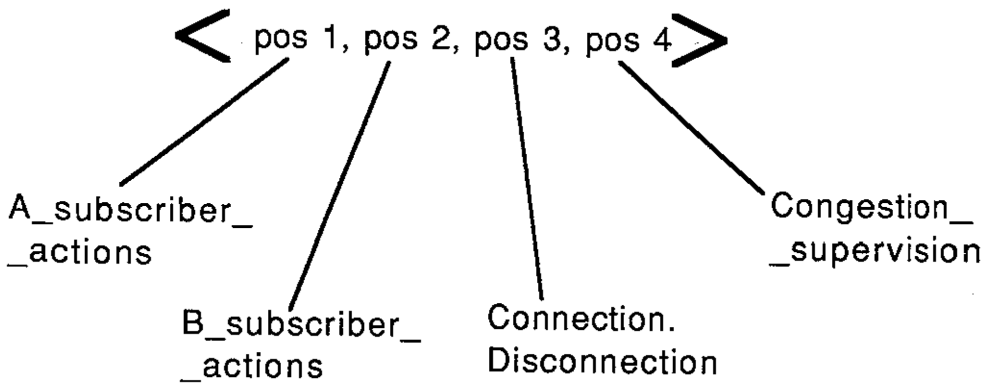


FIGURE D-5.3.7

Arrangement of positions in a name tuple

All possible state names in the different positions give 300 combinations.

Ex. <A IDLE, B_IDLE, CONNECTED, NO_ALARM>
<AWAIT_CONN, B_IDLE, CONNECTED, NO_ALARM>
<AWAIT_FIRST_DIGIT, B_IDLE, CONNECTED, NO_ALARM>
…..
<AWA1T_A_ON_HOOK 2_ , B_IDLE, CONNECTED, NO_ALARM>
<A_IDLE, BRINGING, CONNECTED, NO_ALARM>
<A_IDLE, B_CONVERSATION, CONNECTED, NO_ALARM>
....
<A_IDLE, AWAIT B_ON HOOK, CONNECTED, NO_ALARM>
………
………
<AWAIT A_ON HOOK 2_ , AWAIT_B__ON HOOK, SEIZED, ALARM>

Many of these combinations are not relevant because a subscriber line can never be used as both A-subscriber party and B-subscriber party in the same call. It means that the 10 states in 'A_subscriber_actions' can only be combined with one state in 'B_subscriber_actions' which is 'B_IDLE'. It also means that the 5 states in 'B_subscriber actions' can only be combined with one state in 'A_subscriber_actions' (A_IDLE). The number of relevant states is then (10*1*3*2) + (1*5*3*2) = 90

A reduction of the state space, as examplified above, is dependent on the current example and can not be formalized in general rules applicable for automatic transformation. However, if the transformation is manually performed the state space can probably be reduced. Thus, when comparing the complexity of a process designed without services with the complexity of the same process partitioned into services the process with the reduced state space should be used to get a fair comparison. In most cases using of services will greatly reduce the number of states.

D.5.3.3.2 *Transformation of transitions*

In the recommendation specific rules for transformation of transitions are given. The following example is an illustration of transformation. The process in the example is a specification of a simple protocol. The process is partitioned into two services, Send and Receive.

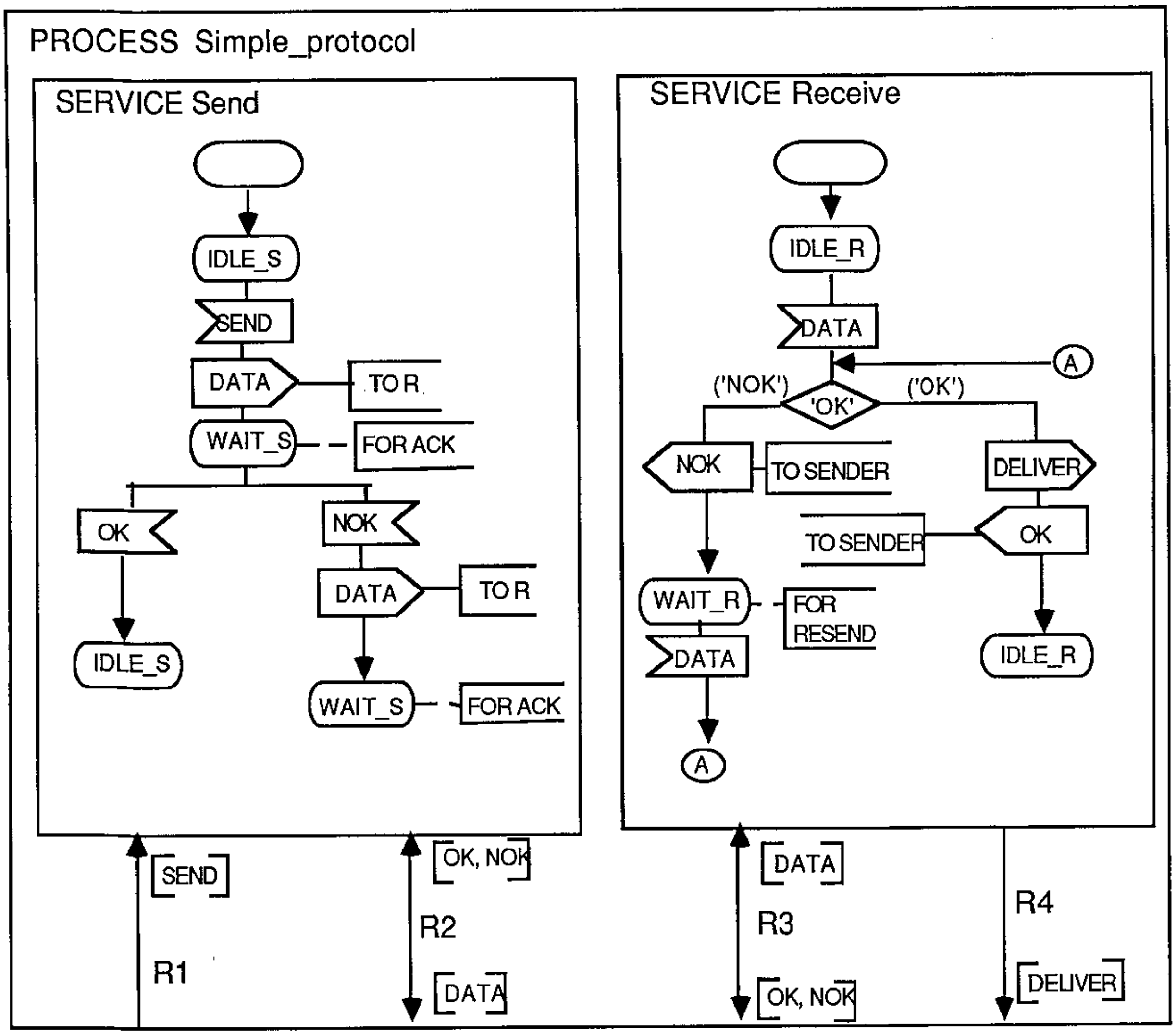
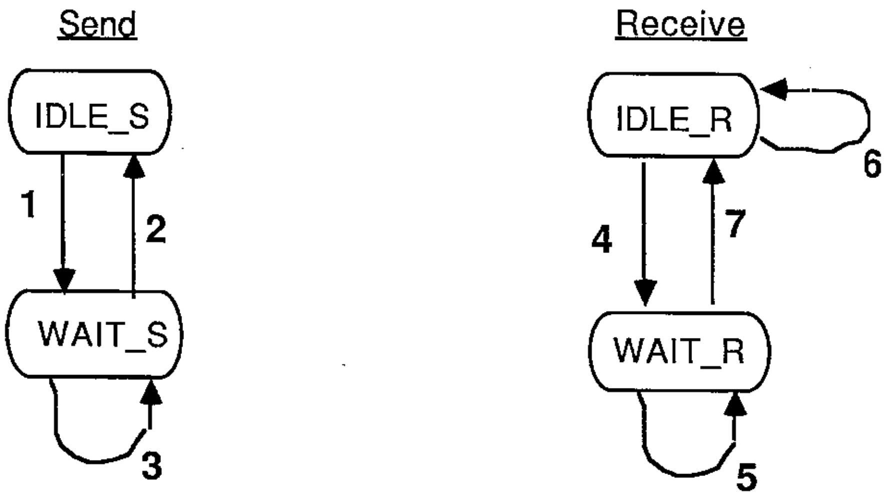


FIGURE D-5.3.8

Process diagram including two service diagrams

The following figure (D-5.3.9) is a state overview diagram for the two services. The transitions are numbered 1-7.



, FIGURE D-5.3.9

State overview diagrams with numbered transitions

Applying the formal rules for transformation of states and transitions the result is the following state overview diagram for the process. No priority signals are used, so the extra splitting of states into preliminary and main is not required and is not shown.

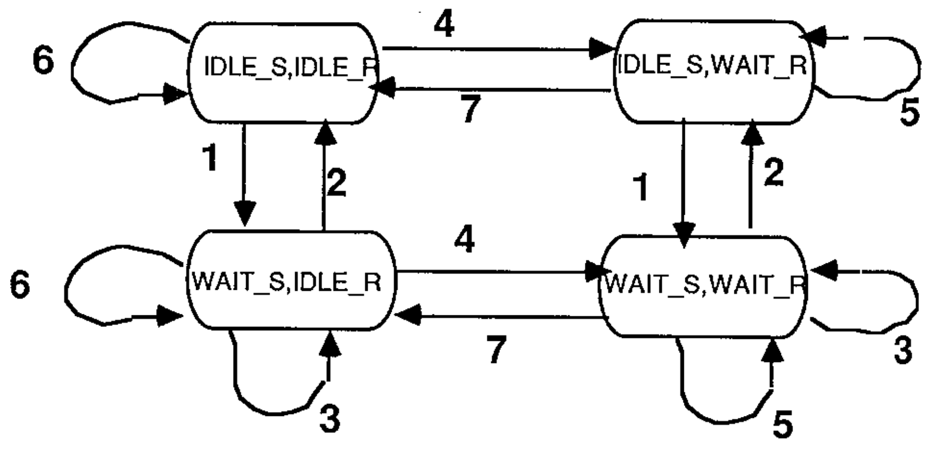


FIGURE D-5.3.10

State overview diagram for the transformed services

The number of states can not be reduced and the process graph is shown in fig. D-5.3.11. 'The state names in the process graph correspond to the name tuples in the figure D-5.3.10. Example: IS.IR corresponds to IDLE_S,IDLE_R.

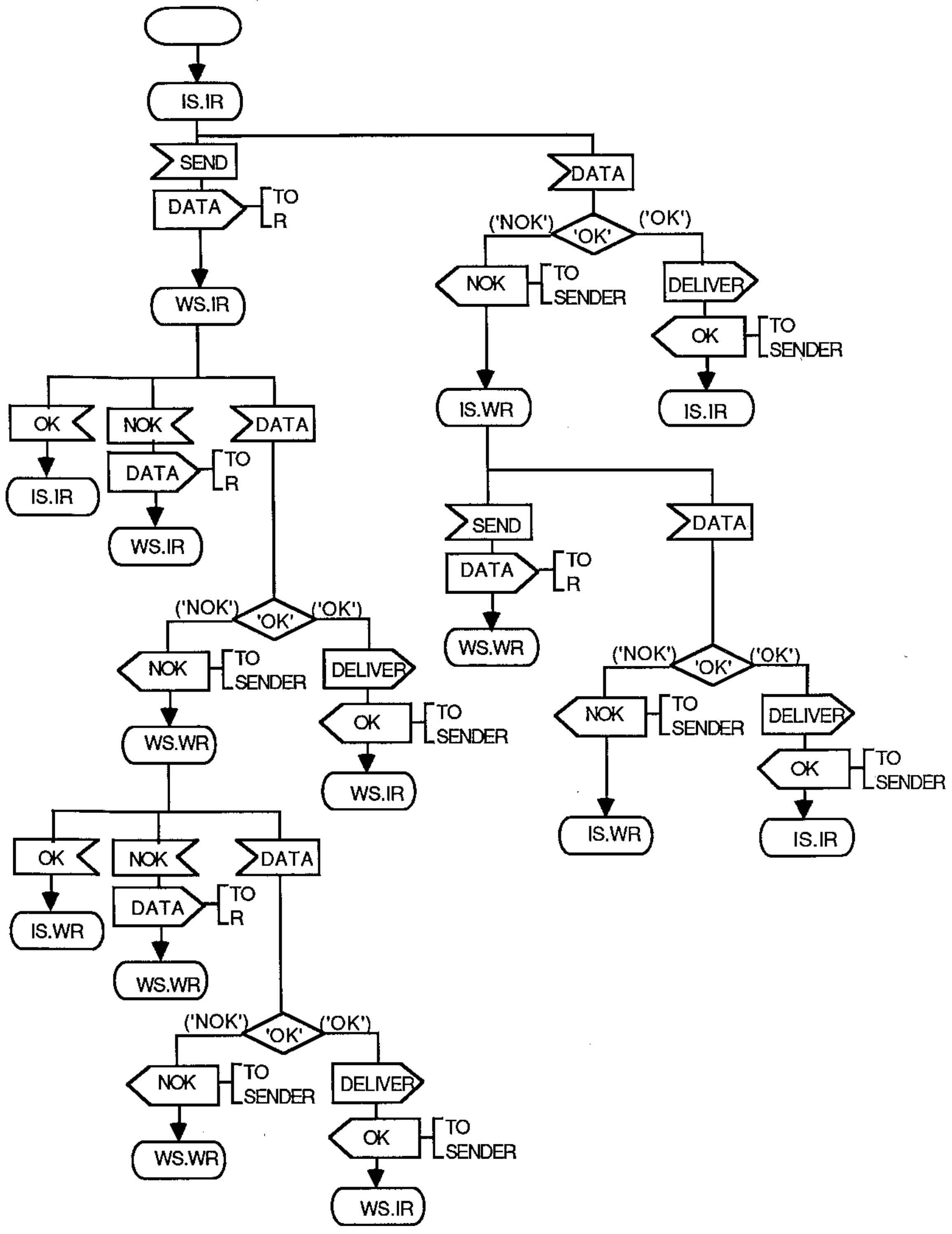


FIGURE D-5.3.11

Process graph transformed from two service graphs

D.5.4 *Guidelines for state-oriented representation and pictorial elements*

This chapter explains the state-oriented representation of SDL/GR and pictorial elements.

The first section (paragraph D.5.4.1) contains general comments on the state-oriented representation ; on its feature, its suitable applications, and its variations.

The second section (paragraph D.5.4.2) explains state description by pictorial elements.

D.5.4.1 *General comments on state-oriented representation*

The SDL/GR gives a choice of three different versions to describe a process diagram.
The first one is called the transition-oriented version of SDL/GR, in which transitions are described exclusively by explicit action symbols.

The second one is called the state-oriented version of SDL/GR or state-oriented pictorial extension of SDL, in which states of a process are described using state picture, and transitions are only implied by the difference between originating and terminating states.

The last one is called the combined version which is a combination of both versions.

Examples of these three versions are given in Annex E of this Fascicle.

The transition-oriented version is suitable when the sequence of actions is important and detailed state descriptions are not important.

The state-oriented version describes the state declaratively and in detail, therefore it is suitable when a process state is more important than the sequence of actions within each transition, when intuitive pictorial explanation is desirable, and when resources and their relations associated to the state are of interest.

The state pictures are usually described by using pictorial elements which indicate relevant resources of the current state of the process. It is available for applications for which suitable pictorial elements are defined, therefore user can apply this representation to any applications by defining appropriate pictorial elements if needed.

The combined version is suitable when both the sequence of actions within each transition and detailed state descriptions are under consideration.

D.5.4.2 *State picture and pictorial element*

D.5.4.2.1 *Pictorial element and qualifying text*

If the state picture option is chosen , a state picture consists of pictorial elements and qualifying text shown in Figure D-5.4.1 a) to d).

This combination makes state pictures understandable. For example, Figure D-5.4.1 a) gives the meaning of a dial receiver manipulated by the process, example b) gives the meaning of a dial tone sender that is sending a permanent signal to the environment.

Note that output signals (non-permanent signals) and the relevant resources are not described in state pictures, output signals may be described in transition diagram.

Example c) shows a timer whose expiration is always represented by an input. Note that the recommended pictorial element for timer incorporates the relevant input signal tl.

The last example d) means that a voice message recorder is now recording.

The resource identity may be heavily abbreviated, and where possible should be placed inside appropriate pictorial element. It is then quite obvious which pictorial elements are being qualified

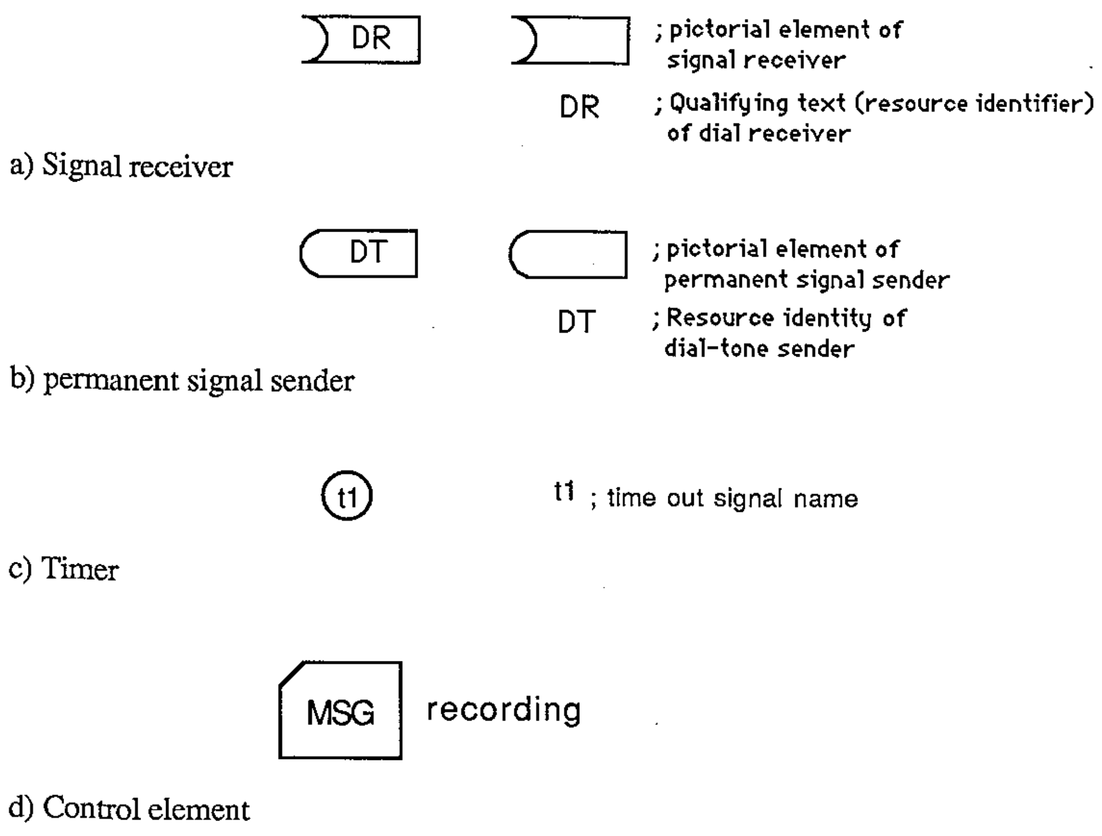

DR
; pictorial element of signal receiver
DR ; Qualifying text (resource identifier) of dial receiver
a) Signal receiver
DT
; pictorial element of permanent signal sender
DT ; Resource identity of dial-tone sender
b) permanent signal sender
t1
t1 ; time out signal name
c) Timer
MSG
recording
d) Control element

FIGURE D-5.4.1

Examples of pictorial element with qualifying text

D.5.4.2.2 *Completeness of state picture*

Sufficient pictorial elements and qualifying text should be assigned to each state picture in order to show:

a) what objects or resources are being considered by this process during the current state. Examples of resources are: switching paths, signalling receivers, senders of permanent signals, and switching modules;

b) whether one or more timers are currently supervising this process;

c) in case that this process concerns call-handling ,whether call charging is currently in progress or not ,and which subscribers are being charged during this state of the call;

d) what objects owned by another process (environment) are being considered to have relations to resources of the process during the current state;

e) the output permanent signals which are output during this state;

f) the relationship between signals and resources in the state;

g) the state of resources relevant to the process current state.

D.5.4.2.3 *Example*

An example of the application of the principle expressed above is illustrated by the state picture in Figure D-5.4.2. It will be seen that during this state:

a) the resources considered by the process consist of a dialled digit receiver ,a dial-tone sender ,a subscriber's handset owned by environment, and switching paths connecting these items;

b) a timer t0 is currently supervising this process;

c) no charging of the call is in progress;

d) the subscriber has been identified as A-party ,but no other category information is taken into account;

e) the following input signals are awaited; A_on_hook (i.e. the handset on-hook signal), digit (i.e. the dialled digit) and t0 (i.e. the supervising timer t0 is running);

f) the output permanent signal DT (i.e. the dial-tone) has been output prior to and during this state.

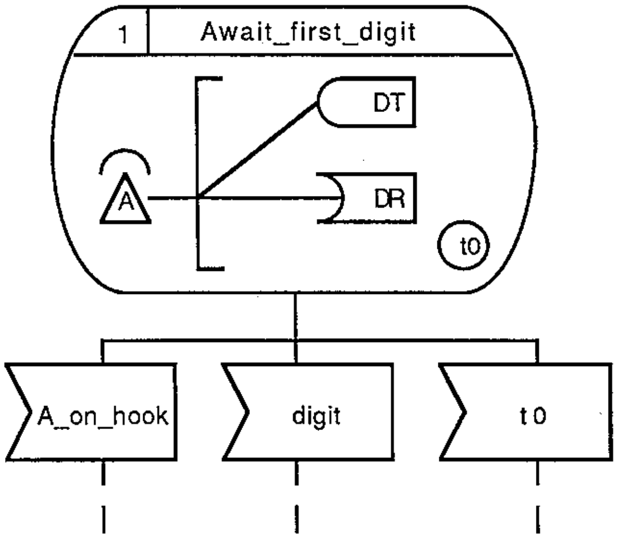


FIGURE D-5.4.2

Example of a state in a call-handling process

D.5.4.2.4 *Consistency checking feature of SDL diagrams with pictorial element*

It is clear that the state picture is more compact and in a certain sense puts more information under the eyes of readers; but at the same time one has to look very closely to the resource to work out the exact set of actions performed in the transition.

In addition it is not possible to tell, looking at the state picture alone, the order of actions in the transition.

Pictorial elements shown outside the block boundary symbol imply elements that are not directly controlled by the given process, and pictorial elements shown within the block boundary symbol imply elements that are directly controlled by the given process. For example, the call process that is partially specified in Figure D-5.4.3 can allocate or deallocate ring-tone sender, ringing sender and speech paths, and start or stop timer t4, but it cannot control directly subscriber's handset condition.

In designing logic from an SDL specification with pictorial elements, only those pictorial elements shown inside the block boundary will affect the processing actions performed during transition sequences. The pictorial elements shown outside the block boundary are normally included in a state picture:

a) because they indicate resources and state of environment which have concern in input signal of the process during the given state;

b) to improve the intelligibility of the diagram.

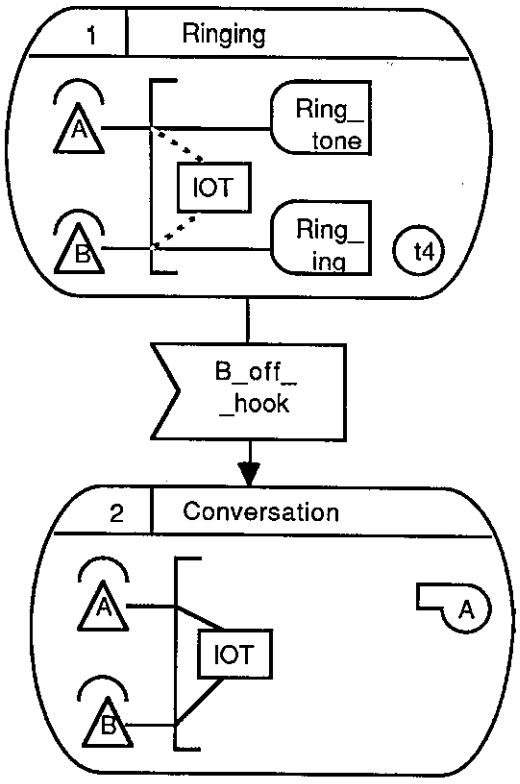


FIGURE D-5.4.3

Example of a transition between two states,where all processing actions are implied by the difference in the state pictures

D.5.4.2.5 *Use of the timer symbol*

Whether or not pictorial elements are used , a timer expiration is always represented by an input.

The presence of a timer symbol in a state picture implies that a timer is running during that state

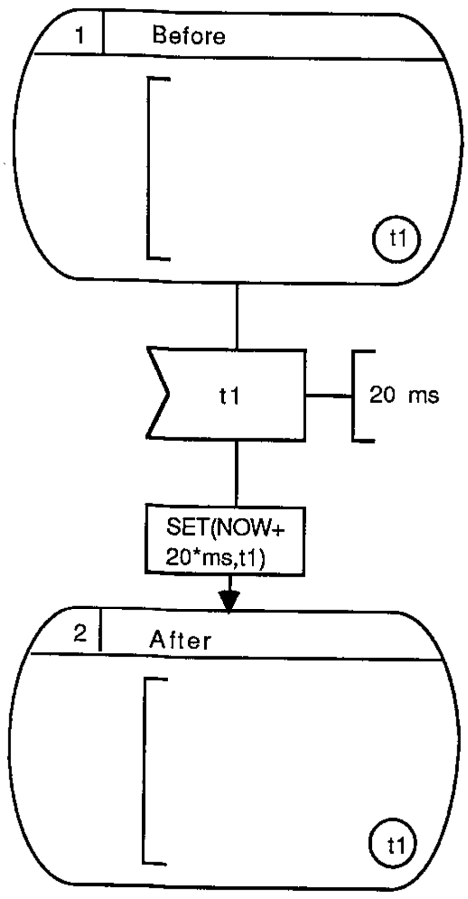
Following the general principle stated in the Recommendations, the starting, stopping restarting and expiration of timers is shown using pictorial elements in the following manner:

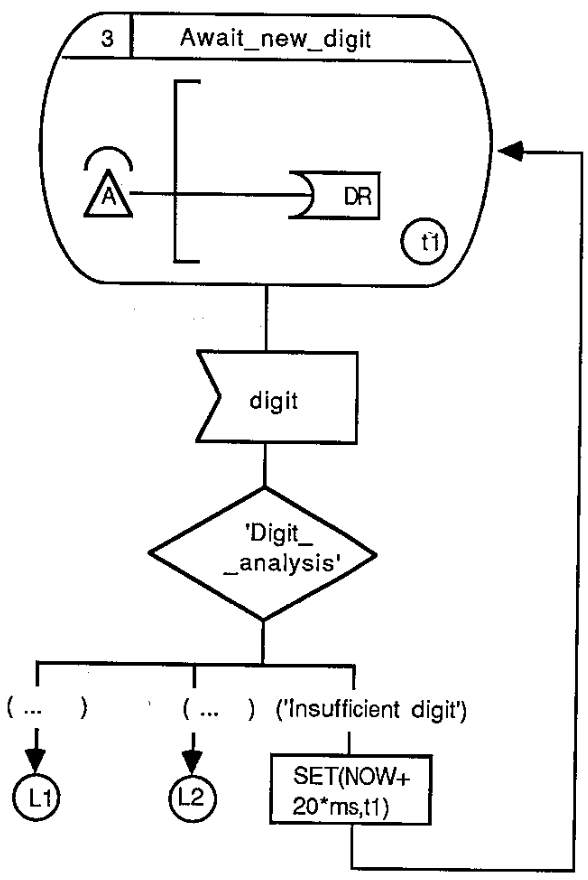
a) To show that a timer is started during a given transition ,the timer symbol should appear in the state picture corresponding to the end of that transition but not in the state picture corresponding to the start of that transition.

b) To show that a timer has been stopped during a transition, the timer symbol should appear in the state picture corresponding to the start of that transition but not in the state picture corresponding to the end of that transition.

c) To show that a timer has been restarted during a transition, an explicit task symbol should be shown in that transition.(Two examples are shown in Figure D-5.4.4)

d) The expiration of a particular timer is shown by an input symbol associated with a state in which the state picture includes the corresponding timer symbol. Of cource more than one timer may concurrently supervise the same process if required, as shown in Figure D-5.4.5.





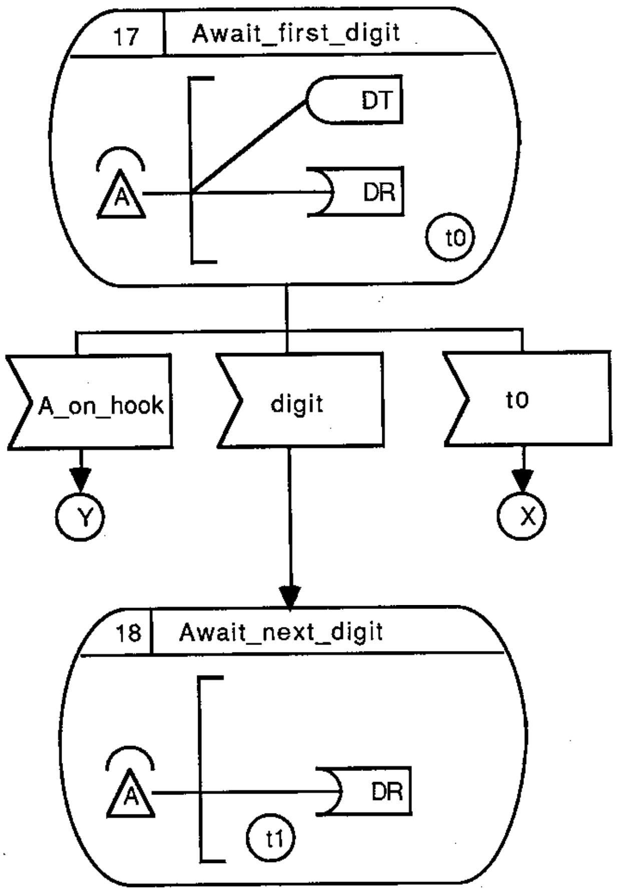
a) Restart of a timer following the timer expiration

b) Restart of a timer when the timer has not yet expired

Note - Each timer t1 has two mutually exclusive conditions: active and inactive.

FIGURE D-5.4.4

Restart of a timer



Note: Timer t0 supervises the arrival of the first digit,whereupon dial-tone is removed from the call and timer t0 is stopped. Timer t1 continues supervising the arrival of sufficient digits to permit routing of the call.

FIGURE D-5.4.5

Example of the use of two supervising timers in the same state

D.5.5 *Auxiliary diagrams*

To aid the reading and understanding of large and/or complex process diagrams, the author may provide informal auxiliary diagrams. The purpose of such documents is to give an overview or simplified description of the process behaviour (or part of it). Auxiliary documents do not replace but give an introduction to SDL documents.

In this section examples of some commonly used auxiliary diagrams are given, i.e. state overview diagram, state/signal matrix, sequence chart. (The block tree diagram described in § D.4.4 is an auxiliary diagram as well).

D.5.5.1 *State overview diagram*

The intention with the state overview diagram is to give an overview of the states of a process and what transitions between them are possible. As the intention is to give an overview, "unimportant" states or transitions may be omitted.

The diagrams are composed of state symbols, directed arcs representing transitions, and optionally start and stop symbols.

The state symbol should contain the name of the referred state. The symbol may contain several state names and an asterisk (*) may be used as notation for all states.

To each of the directed arcs the name of the signal, or set of signals, causing the transition can be associated as well as possible outputs sent during the transition.

An example of a state overview diagram is given in Figure D-5.5.1.

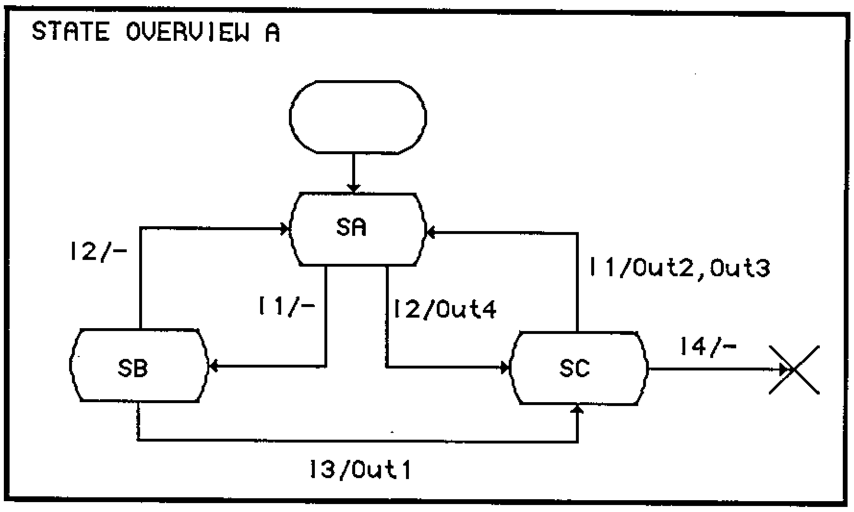


FIGURE D-5.5.1

Example of a state overview diagram

A state overview diagram for a process may be separated into several diagrams each covering different aspects, e.g. "normal case", "fault_handling", etc.

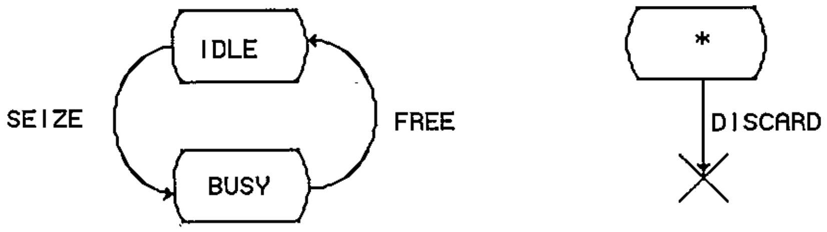


FIGURE D-5.5.2

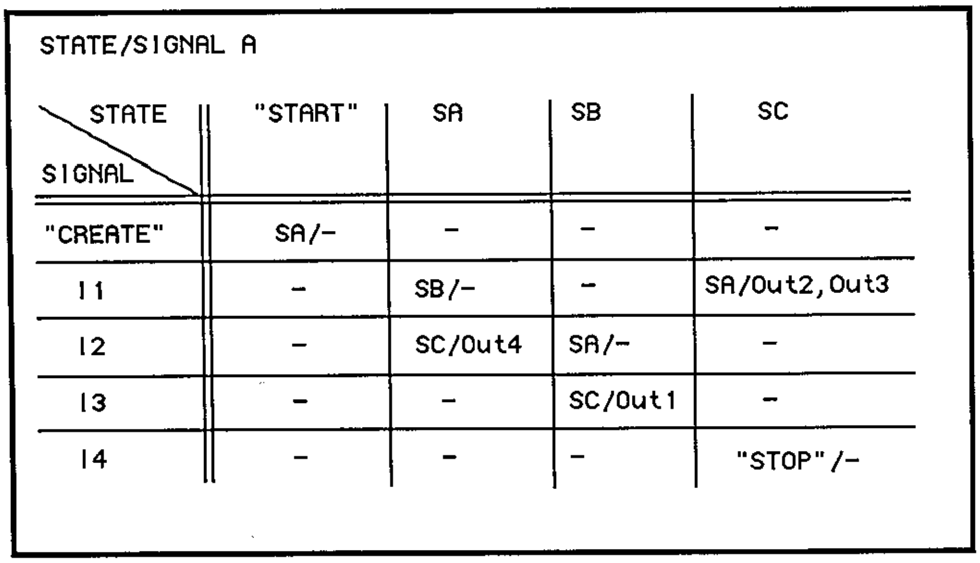
Example of a separated state overview diagram

D.5.5.2 *State/signal matrix*

The state/signal matrix is an alternative to the state overview diagram containing exactly the same information.

The diagram consists of a two-dimensional matrix indexed on one axis by all the states of the process, and on the other by all valid input signals for the process. In each of the matrix elements the next state is given together with possible outputs during the transition. A reference may be given to where to find the combination given by the indices, if it exists.

The element corresponding to the dummy state "START" and the dummy signal "CREATE" is used to show which is the initial state of the process.



STATE/SIGNAL A

STATE / SIGNAL	"START"	SA	SB	SC
"CREATE"	SA/-	-	-	-
I1	-	SB/-	-	SA/Out2, Out3
I2	-	SC/Out4	SA/-	-
I3	-	-	SC/Out1	-
I4	-	-	-	"STOP"/-

FIGURE D-5.5.3

Example of state/signal matrix

The matrix may be separated in sub-parts contained on different pages. The references are the normal references used by the user in the documentation.

Preferably signals and states should be grouped together, so that each part of the matrix covers one aspect of the process behaviour.

D.5.5.3 *Sequence chart*

The sequence chart may be provided to show allowed sequences of signals interchanged between services, processes, blocks and their environment.

The chart is intended to give an overview appreciation of the signal interchange between parts of the system. The complete signal interchange or only parts of it may be shown depending on the aspects that are to be highlighted.

The columns in the chart indicate the communicating entities (services, processes, blocks or the environment).

Their interactions are represented by a set of directed lines, each one representing a signal or a set of signals.

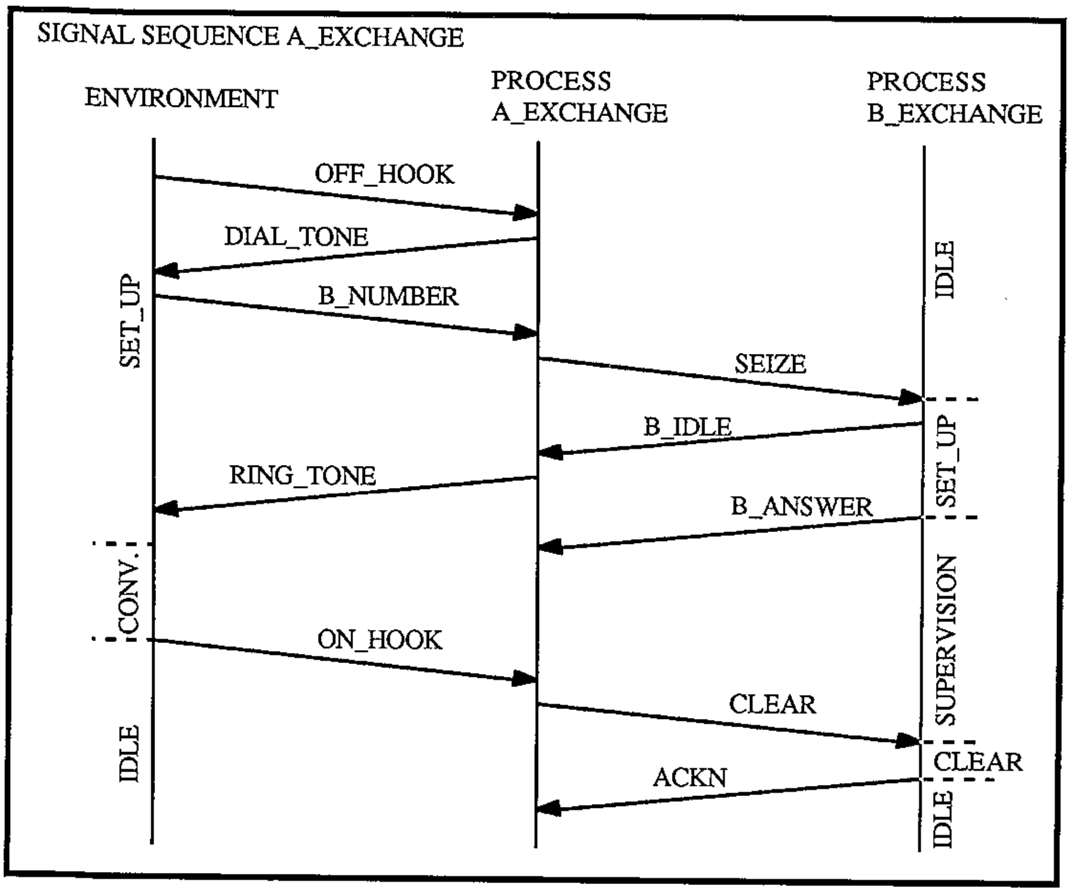


FIGURE D-5.5.4

Example of sequence chart

Each sequence may be annotated so that it is clear what set of information is interchanged. Each line is annotated with the information involved (signal names or procedures).

A decision symbol can be placed on the columns to indicate that the following sequence is valid if the indicated condition is true. Usually in this case the decision symbol appears several times indicating the different sequences resulting from the different values of the condition.

This diagram can show completely all the sequences of signals interchanged or only a meaningfull subset.

A usefull application of this diagram is the representation of the mutual interaction of services resulting from the partitioning of a process.

Sequence charts will usually not cover all possible sequences; they are often preliminary to the full definition.

D.6 *Definition of SDL data*

D.6.1 *Guidelines on data in SDL*

This section gives more information on the concepts defined in § 5 of the SDL Recommendation. The main difference between this section and the former Z.104 is that it is refined for clarification and for harmonisation with ISO. Some of the keywords have been changed and there are some extensions, but the semantics are consistent with the intended meaning of the red book. The use of data in § 2, § 3 and § 4 of the new recommendation (previously Z.101 - Z.103) is unchanged!

D.6.1.1 *General introduction*

The approach the data types of SDL are based on is the 'abstract data type'. This is an approach where one does not describe HOW a type must be implemented, but only tells WHAT will be the result of operators when applied to values.

When abstract data is defined, each segment of the definition, called a "partial type definition" is introduced by the keyword NEWTYPE. Each partial type definition affects the others so that all the partial type definitions at system level constitute a unique data type definition. If more partial type definitions are introduced at a lower level (e.g. at block level) they constitute together with the higher level definitions one data type definition. That is at any point in the specification there is just one data type definition.

Essentially the data type definition consists of three parts:

a) Sort definitions,

b) Operator definitions,

c) Equations.

Each of these parts is explained below. The data type definition is structured in partial data type definitions, each introducing one sort. The operator definitions and equations are spread over the partial data type definitions.

D.6.1.2 *Sorts*

A sort is a set of values. This set can have a finite or infinite number of elements, but cannot be empty.

Examples:

a) The set of values of sort Boolean is {True, False}.

b) The set of values of sort Natural is the infinite set of natural numbers { 0, 1, 2, ... } .

c) The set of values of sort Primary_colour is {Green, Red, Blue}.

Every element of a sort need not directly be provided by the user (it would take an infinite amount of time in case of the natural numbers) but the name of the sort must be given. In the concrete syntax the keyword NEWTYPE is directly followed by the name of the sort (some other possibilities will be seen later). This name is mainly used in operator definitions, as explained in § D.6.1.3 and in variable declarations.

D.6.1.3 *Operators, literals and terms*

Values of a sort can be defined in three ways:

a) by enumeration; the values are defined in the literals-section,

b) by operators; values are produced as results of 'applications' of operators,

c) by a combination of enumeration and operators.

The combination of literals and operators gives terms. Relations between terms are expressed using equations. The following sections deal with literals, operators and terms; § D.6.1.4 deals with the equations.

D.6.1.3.1 *Literals*

Literals are enumerated values of a sort. A partial type definition does not need to contain literals; all elements of the sort could be defined via operators. Literals may be regarded as operators without arguments. A relation between literals may be expressed in equations. In the concrete syntax the literals are introduced after the keyword LITERALS.

Examples:

a) The definition of sort Boolean contains two literals, namely True and False. So the definition of the boolean type looks like:

```
NEWTYPE Boolean
      LITERALS True, False
      . . .
ENDNEWTYPE Boolean;
```

b) The sort Natural may be defined using one literal, the zero. The other values can be generated using this literal and operators.

c) The values of sort Primary_colour can be defined similarly to the boolean literals:

```
NEWTYPE Primary_colour
      LITERALS Red, Green, Blue
      . . .
ENDNEWTYPE Primary_colour,
```

d) In § D.6.1.3.2 the third example (c) shows a partial type definition without literals.

D.6.1.3.2 *Operators*

An operator is a mathematical function mapping one or more values (possibly from different sorts) to a result value. Operators are introduced after the keyword OPERATORS, the sort(s) of their argument(s) and the sort of the result are denoted too (this is called the signature of the operators).

Examples:

a) In the type boolean an operator called "not" can be defined, having one argument of sort Boolean and a result of sort Boolean too. In the definition of the boolean type it appears as follows:

```
NEWTYPE Boolean
LITERALS True, False
        OPERATORS "Not": Boolean -> Boolean ;
        . . .
ENDNEWTYPE Boolean;
```

b) The operator referred to in the previous section needed to construct all natural numbers is Next. This operator takes an argument of sort natural and gives a natural value (the next higher) as a result.

c) A new type for colours can be defined which has no literals but uses literals of sort Primary_colour and some operators:

```
NEWTYPE Colour
        OPERATORS
                Take: Primary_colour -> Colour ;
                Mix : Primary_colour, Colour -> Colour ;
        . . .
ENDNEWTYPE Colour;
```

The intention is to take a primary and regard it as a colour, and then start mixing more primaries into it to get more colours.

D.6.1.3.3 *Terms*

Using literals and operators the set of terms can be constructed for each sort as follows.

1. Collect all literals in a set of the sort they are defined in – each literal is a term.
2. For each operator a new set of terms is created where the operator is applied to all possible combinations of terms of the previously created sets of the correct sort.
   a) For sort Boolean the set of literals is {True, False}. The result of this step is { Not(True), Not(False)) because we only have operator Not.
   b) For sort Natural the result of this step is { Next(O) } .
   c) For sort Colour the set of literals is empty, but the result of this step is {Take(Red), Take(Qreen), Take(Blue) } .
3. The terms of the sets created in the previous step are all of the sort of the result of the operator applied. e.g., all results of the operator Not are of the boolean sort. Now the union is taken of all sets of the same sort, both the original and the newly created sets.
   a) For sort Boolean this union gives the set {True, False, Not(True), Not(False)).
   b) For the natural numbers this step gives the set {0, Next(O)}
4. The last two steps are repeated over and over, in general defining an infinite set of terms.
   a) The set of boolean terms generated by the literals True and False and the operator Not is {True, False, Not(True), Not(False), Not(Not(True)), Not(Not(False)), Not(Not(Not(True))), ...)
   b) The set of natural terms generated by literal 0 and operator Next is {0, Next(0), Next(Next(0)), Next(Next(Next(0))), ... }
   c) The set of colour terms generated by the literals Red, Green and Blue of sort Primary_colour and the operators Take and Mix is (Take(Red), Take(Green), Take(Blue), Mix(Red, Take(Red)), Mix(Red, Take(Green)), Mix(Red, Take(Blue)), Mix(Green, Take(Red)), Mix(Green, Take(Green)), Mix(Green, Take(Blue)), Mix(Blue, Take(Red)), Mix(Blue, Take(Green)), Mix(Blue, Take(Blue)), ... }.

D.6.1.4 *Equations and axioms*

In general the number of terms generated as in the previous section is greater than the number of values in the sort. E.g., there are two boolean values, but the set of boolean terms has an infinite number of elements. There exists, however, a possibility to give rules that state what terms are regarded as denoting the same value. These rules are called equations, and they are explained in the next subsection. Two special kinds of equations, axioms and conditional equations, are treated in § D.6.1.4.2 and § D.6.1.4.3.

Equations, axioms and conditional equations are all given, in the concrete syntax, after the keyword AXIOMS. This keyword is kept for historical reasons.

D.6.1.4.1 *Equations*

An equation states which terms are regarded as denoting the same value. An equation relates two terms which are separated by the equivalence-symbol ==.

For instance, "Not(True) == False" states that the terms Not(True) and False are equivalent; wherever Not(True) is found, False can be substituted without changing the meaning and vice versa.

By giving some equations the set of terms is divided in disjoint subsets of terms which denote the same value. These subsets are called equivalence classes. In daily speaking, equivalence classes are identified with the values.

Examples:

a) The set of terms of sort boolean is divided in two equivalence classes of terms by the following two axioms:

```
Not(True) == False ;
Not(False) == True ;
```

The resulting equivalence classes are:

{True, Not(False), Not(Not(True)), Not(Not(Not(False))), Not(Not(Not(Not(True)))),
Not(Not(Not(Not(Not(False))))), ...}
and

{False, Not(True), Not(Not(False)), Not(Not(Not(True))), Not(Not(Not(Not(False)))),
Not(Not(Not(Not(Not(True))))), ...}.

These two equivalence classes are identified with the values True and False.

b) In the case of colours one wants to specify that mixing a primary with a colour that contains the primary makes no difference. Furthermore, the order in which primaries are mixed is irrelevant. This can be stated in equations as follows:

```
Mix(Red, Take(Red))                 == Take(Red) ;
Mix(Red, Mix(Red, Take(Green)))     == Mix(Red, Take(Green)) ;
Mix(Red, Mix(Red, Take(Blue)))      == Mix(Red, Take(Blue)) ;
Mix(Red, Mix(Green, Take(Red)))     == Mix(Green, Take(Red)) ;
Mix(Red, Mix(Blue, Take(Red)))      == Mix(Blue, Take(Red)) ;        etcetera.
```

This is a lot of work, because for all permutations of Red, Green and Blue similar equations appear. Therefore SDL has the FOR-ALL-construct, which introduces value names which stand for an arbitrary equivalence class (or the value associated with this equivalence class). In the situation above this can be very helpful; all equations stated above and those indicated by etcetera can be written in a few lines as follows:

```
FOR ALL p l, p2 IN Primary_colour
/* 1 */    (    Mix( p1, Take( p 1 )  )  == Take( p1 ) ;
/* 2 */         Mix( pl, Mix( pl, Take( p2 ) ) ) == Mix( pl, Take( p2 ) ) ;
/* 3 */         Mix( pl, Mix( p2, Take( pl ) ) ) _= Mix(p2, Take(pl ) ) ;
/* 4 */         Mix( pl, Take( p 2 )  )  == Mix( p2, Take( pl) );
                FOR ALL c IN Colour
/*5*/           (        Mix( p1,Mix( p 2 , c )  ) = =  Mix( p2, Mix( p1,c) );
/* 6 */                  Mix( pl, Mix( p2, c )   )  == Mix( Mix ( p1, Take( p2 ) ), c ))
           )
```

In the equations above there is overlap but this is not a problem as long as the equations do not contraddict each other.

The equations above create 7 equivalence classes in the set of terms of sort Colour, so with these equations there are 7 colour-values. The following terms are in different equivalence classes:

```
Take( Red ), Take( Green ), Take( Blue ), Mix( Red, Take(Green) ),
Mix( Green, Take(Blue) ),
Mix( Blue, Take(Red) ),
Mix( Blue, Mix( Green, Take(Red) ) ).
```

All other terms of sort Colour are equivalent to one of the terms above.

In the examples of equations with the FOR-ALL-construct, so called explicitly quantified equations, the information that p l and p2 are value identifiers of sort Primary_colour is redundant; the argument of operator Take and the first argument of operator Mix can only be of sort Primary_colour. In general it makes the equations more readable when they are quantified explicitly, but it is allowed to omit the quantification if the sort of the value identifiers can be deduced from the context. In that case the equation is said to be implicitly quantified.

Example:

The equations 4 and 5 above are the same as

```
Mix( p1, Take( p 2 ) )          == Mix( p2, Take( p 1 ) )  ;
Mix(p I, Mix( p2, c ) )         == Mix(p2, Mix( p 1, c ) ) ;
```

D.6.1.4.2 *Axioms*

Axioms are just a special kind of equations, introduced because a lot of equations in practical situations relate to booleans. In that case equations tend to be of the form ... == True ; i.e. they state that some term is equivalent to True.

Example:

Suppose that for type colour an operator is defined: Contains: Colour, Primary_colour -> Boolean ; which is intended to give True if the primary is contained in the colour and False otherwise. Some of the involved equations are:

```
FOR ALL p IN Primary_colour
(       Contains( Take( p ), p)                 == True ;
        FOR ALL c IN Colour
        (        Contains( Mix( p, c ), p )     == True )
)
```

The '== True'-part of these equations may be omitted, and the results are called axioms. Axioms can be recognised by the absence of the equivalence-symbol ==, and they denote terms that are equivalent to the value True of sort Boolean.

The construction of the second equation may look a little forced. A better way to write these equations is shown after the introduction of some helpful constructs.

D.6.1.4.3 *Conditional equations*

Conditional equations are a means to write equations which only hold if some conditions hold. The conditions are denoted with the same syntax as unconditional equations and separated by a ==> symbol from the equation that holds if the condition holds.

Example:

The standard example of a conditional equation is the definition of division in the type real where

```
FOR ALL x, z IN Real
(       z /= 0 == True          ==>             ( x / z ) * z = = x )
```

states that if the condition "z not equals 0" holds then division by z followed by multiplication by z gives the original value. This conditional equation states nothing about what should happen if a value of sort Real is divided by zero. If one wants to specify what happens in case of division by zero a conditional equation of the form

```
FOR ALL x, z IN Real

(       z = 0 == True           ==>             ( x / z ) * z = = . . . )
```

must be given. In these cases, however, a so called 'conditional term' on the right hand side is recommended for readability reasons. In the case above the equation would be:

```
FOR ALL x, z IN Real
(       ( x / z ) * z       ==      I F z / = 0
                                    THEN x
                                    ELSE . . .
                                    FI
```

D.6.1.5 *More on equations and axioms*

The following two sections explain some difficulties one may encounter when operators have results of an already defined sort. § D.6.1.5.3 explains the concept of error as term in an equation.

D.6.1.5.1 *Hierarchy consistency*

At any point in an SDL specification there is one and only one data type definition. This data type definition contains the predefined sorts, operators and equations and all sorts, operators and equations defined by the user in the partial type definitions visible at that point. (This is why a text NEWTYPE...ENDNEWTYPE is called a partial type definition.)

This has some consequences for type definitions on lower levels. They might influence the type in an undesired way. E.g., one could erroneously specify that two terms are equivalent, thereby making them equivalent while they are not equivalent in a surrounding scope.

It is not allowed to give equations in such a way that:

a) values of a sort which are different in a scope at a higher level are made equivalent;

b) new values are added to a sort defined in a scope at a higher level.

This means e.g. that in a block at system level, partial type definitions specified by the user containing an operator with a predefined result sort must relate all terms produced by this operator to values of this result sort.

Examples:

a) If, for some reason, one gives the axiom:

```
FOR ALL n, m IN Integer
(       ( F a c t ( n ) = F a c t ( m ) )     =>      ( n = m ) )
```

with the intention to specify that if the results of operator Fact are the same, then the arguments are the same. (Note that => is the boolean implication; this has little to do with the conditional equation sign ==>.) Then by accident values are unified. From the equations in the previous example it can be derived that Fact(0) = Fact(1), and this last equation states that 0 and 1 are different denotations of the same value. From this last equation it can be proved that the number of elements in the Integer sort is reduced to one.

With the help of a conditional equation it can be stated that, provided n and m are not equal to zero, the same result of operator Fact on n and m implies n=m. In SDL:

```
FOR ALL n, m IN Integer
(       n / = 0 , m / = 0  = > ( F a c t ( n ) = F a c t ( m )  ) = > ( n = m )  )
```

Note that this last equation doesn't add anything to the semantics of Integers; it is a theorem that can be derived from the other equations. On the other hand, the addition of a provable equation does not harm.

a) Suppose one discovers the need of an operator for factorials when specifying some type. In the partial type definition of this type the operator Fact is introduced:

```
Fact: Integer -> Integer ;
```

and the following equations are given to define this operator:

```
Fact(0) == 1;
FOR ALL n IN Integer
(       n > 0 ==> Fact( n ) == n * Fact( n-1 ) )
```

These equations do not define Fact(- 1) and so this is a term of sort Integer which has no relation with other terms of this sort. Therefore Fact(-1) is a new value of sort integer (and the same holds for Fact(-2), Fact(-3), etc.). This is not allowed. The example b) of § D.6.1.5.3 shows a correct definition of fact.

D.6.1.5.2 *Equality and inequality*

In each type the operators for equality and inequality are implied. So, if a partial type definition introduces sort S, then there are implicit operator definitions:

```
"=" : S, S -> Boolean ;
"/_" : S, S -> Boolean ;
```

(Note: The quotes specify that = and 1= are used as infix operators).

The equality operator has the properties one would expect;

```
a=a,
a = b = > b = a ,
a = b AND b = c => a = c ,
a = b => a == b,
a = b => op( a ) = op( b ) for all operators op.
```

The above properties are not written in SDL syntax and must not be stated in axioms or equations because they are implied. The boolean value obtained when this operator is applied is True when the terms on the left hand side and the right hand side are in the same equivalence class, otherwise the value obtained is False. If it is not explicitly specified that the value is either True or False, the specification is incomplete.

For the inequality operator the semantics are best explained by an SDL-equation:

```
FOR ALL a, b IN S
(       a / = b = = N o t ( a = b )  )
```

There is no difference between equality and equivalence. Two terms which are equivalent denote the same value and the equality operator between them gives the result True.

D.6.1.5.3 *Error*

In previous examples the need was felt to specify that application of operators to some values is regarded an error. SDL has a means to specify this formally: the Error. Error should be used to express:

"application of this operator to this value is not allowed and when encountered the future behaviour is undefined."

In the concrete syntax this is denoted by the term Error!, which cannot be used as argument of an operator.

When Error is the result of an operator application, and this application is an argument of another operator, the outer operator application has Error as its result too (error-propagation). In a conditional term the THEN-part or the ELSE-part is evaluated, so one of them can be Error without Error being evaluated (as the other alternative is evaluated).

Examples:

a) In the example of division of values of sort Real the dots can be filled in:

```
FOR ALL x, z IN Real
(       ( x / z ) * z  = =  I F z / = 0
                            THEN   x
                            ELSE   Error! .
                            FI
)
```

For clarity one could add:

```
FOR ALL x IN Real
(       x / 0 == Error! )
```

b) In the example with operator Fact one could specify that application of this operator on negative integers is regarded as an Error. This avoids that Fact(-1), Fact(-2), ... become new values of the Integer sort. A good definition of operator Fact would be:

```
n < 0   ==>     Fact( n ) == Error! ;
                Fact( 0 ) == 1 ;
n > 0   ==>     Fact( n ) == Fact( n-1) * n ;
```

These three lines are much clearer than the programming style equation below. In general the conditional term should be used if there are two complementary cases; nesting of conditional terms ruins the readability, as can be seen from:

```
Fact(n) == I F n > 0
THEN  Fact( n-1 ) * n
ELSE            IF n = 0
                THEN 1
                ELSE Error!
FI
```

D.6.2 *Generators and inheritance*

This section deals with two constructs which can be used to specify types which have common parts. The generator specifies not a type but a schema which becomes a type when the formal sorts, operators, literals and constants are replaced by actual ones.

Inheritance offers the possibility to make a new type starting from an already existing type. Literal and operator names can be renamed and additional literals, operators and equations can be specified.

D.6.2.1 *Generators*

A generator definition defines a schema parameterised by formal names of sorts, literals, constants and operators. Generators are intended for types which are `variations on a theme, such as sets of items, strings of items, files of records, look-up tables, arrays.

This will be explained using an example for which variations can be envisaged. Suppose there is a need for a type that resembles the mathematical notion of a set of integers. The following text is part of the type definition of this integer-set.

```
NEWTYPE Int_set
    LITERALS empty_int_set
    OPERATORS
        Add : Int_set, Integer -> Int_set ;
        Delete : Int_set, Integer -> Int_set ;
        Is_in : Int_set, Integer -> Boolean
    AXIOMS
    /* 1 */    Delete( empty_int_set, int )
                        ==   empty_int_set ;
    /* 2 */    Is_in(set,int1) = false ==>
               Delete( Add( set, int1 ), int2 )
                        ==   IF int1 = int2
                             THEN  set
                             ELSE  Add( Delete( set, int2 ), int1 )
                             FI ;
    /* 3 */    Is_in( empty_int_set, int )
                        ==    False ;
    /* 4 */    Is_in( Add( set, int1 ), int2 )
                        ==   int1 = int2 OR Is_in( set, int2 ) ;
    /* 5 */    Add( Add( set, int1 ), int2 )
                        ==   IF int1 = int2
                             THEN  Add( set, int1 )
                             ELSE  Add( Add( set, int2 ), int1 )
                             FI
 ENDNEWTYPE Int_set;
```

FIGURE D-6.2.1

Newtype Int_set

All equations have implied quantification.The first equation states that deletion of an element from the empty set gives the empty set as result. The second equation tells that deletion after insertion of the same element gives as result the set before the insertion (provided the set does not contain the element), otherwise the order of insertion and deletion can be interchanged. The third equation states that the empty set contains no elements. The fourth equation tells that an element is in a set if it is the last added element or if it was in the set before the last element was added. The last equation states that the order of addition of elements to the set does not make any difference.

In the example of Figure D-6.2.1 Int_set is just an example of a set, and if one needs a PId_set, a Subscriber set and an Exchange_name_set in the same specification as well, no one will be surprised that they all contain the operators Add, Delete and Is_in and a literal for the empty set. The equations given for these operators easyly generalise to the other sets.

This is the place where the generator-concept proves its usefulness, the common text can be given once and can be used several times. Figure D-6.2.2 shows the generator. (Note that formal sort names are introduced by the keyword TYPE. This is for historical reasons only.)

```
GENERATOR Set (TYPE Item, LITERAL empty_set)
    LITERALS empty_set
    OPERATORS
        Add : Set, Item -> Set ;
        Delete : Set, Item -> Set ;
        Is_in : Set, Item -> Boolean
    AXIOMS
    /* 1 */    Delete( empty_set, itm )
                        ==  empty_set ;
    /* 2 */    Is_in(st,itm1) = false ==>
               Delete( Add( st, itm1 ), itm2 )
                        ==  IF itm1 = itm2
                            THEN  st
                            ELSE  Add( Delete( st, itm2 ), itm1 )
                            FI ;
    /* 3 */    Is_in( empty_set, itm )
                        ==  False ;
    /* 4 */    Is_in( Add( st, itm1 ), itm2 )
                        ==  itm1 = itm2 OR Is_in( st, itm2 ) ;
    /* 5 */    Add( Add( st, itm1 ), itm2 )
                        ==  IF itm1 = itm2
                            THEN  Add( st, itm1 )
                            ELSE  Add( Add( st, itm2 ), itm1 )
                            FI
ENDGENERATOR Set;
```

FIGURE D-6.2.2

Generator Set

Instead of using Integer the formal type Item is used, and to be able to give different names to the empty integer-set and the empty sets in other types this literal is made a formal parameter too.

With this generator the type Int set can be constructed as follows:

```
NEWTYPE Int set Set(Integer, empty_mt_set)
ENDNEWTYPE Int_set ;
```

Comparing Figure D-6.2.1 and Figure D-6.2.2 we can see that:

a) GENERATOR and ENDGENERATOR are replaced by NEWTYPE and ENDNEWTYPE respectively,

b) the generator formal parameters (i.e., the text between parentheses after the generator name) are deleted,

c) Set, Item, and empty_set are replaced throughout the generator by Int_set, Integer, and empty_mt_set respectively.

So there is absolutely no difference between this Int_set and the one in Figure D-6.2.1, but ...

– if one needs a set of PId-values the type can be created by

```
NEWTYPE PId set Set(PId, empty_pid_set)
ENDNEWTYPE Pld_set ;
```

– if one needs a set of subscribers, where subscribers are represented by a type that introduces sort Subscr, the set of subscribers can be created by

```
NEWTYPE Subscr_set Set(Subscr, empty_subscr_set)
ENDNEWTYPE Subscr_set ;
```

This does not only save paper, but life becomes easier too because one only has to think once about sets or this work can be delegated to skilled abstract data type specialists.

Example:

This example shows a generator using a formal sort, operator, literal and constant. It describes a row of items with a maximum length max_length. The sort has one literal to denote the empty row and operators for insertion and deletion of items in/from a row, concatenation of rows, selection of a subrow, and determination of the length of a row. This last operator is made formal to be able to rename it.

```
GENERATOR Row (TYPE Item, OPERATOR Length, LITERAL Empty,
                    CONSTANT max_length)
      LITERALS Empty
      OPERATORS
            Length: Row                       -> Integer ;
            Insert: Row, Item, Integer        -> Row ;
            Delete: Row, Integer, Integer      -> Row ;
            "//" : Row, Row                   -> Row ;
            Select: Row, Integer, Integer     -> Row
            /* and other operators relevant for rows of items */
      AXIOMS
            /* The equations for the operators above, among *
            * which the following two (or equivalents)          */
            Length( r) = max_length = = >  Insert( r, itm, int) == Error! ;
            Length( rl ) + Length( r2 ) > max length ==> rl // r2 == Error!
ENDGENERATOR Row ;
```

Note that the formal operator Length and the literal Empty are given once more in the body of the generator because they are renamed when instanciated. In case of the operator, the arguments and the result sort are given in the body only. The generator Row can be used to make lines, pages and books as follows:

```
NEWTYPE Line Row( Character, Width, Empty_line, 80 )
ENDNEWTYPE Line ;

NEWTYPE Page Row( Line, Length, Empty_page, 66 )
ENDNEWTYPE Page ;

NEWTYPE Book Row( Page, Nr_of_pages, Empty_book, 10000 )
ENDNEWTYPE Book ;
```

D.6.2.2 *Inheritance*

Inheritance is a way to get all values of the so called parent sort, some or all operators of the parent type and all the equations of the parent type. For both literals and operators there is a possibility to rename them. In general this is good practice because in that case the reader can deduce from the context that, even if the literals are the same, another type is involved.

If an operator is not inherited, it is systematically renamed to a name unaccessible to the user . The fact the operators are still present means that all equations of the parent type are still present (with renamed operators). This ensures that the parent values are inherited.

Together with the possibility of preventing the use of an operator (by not inheriting it), the possibility of adding new operators is provided. After the keyword ADDING, literals, operators and equations can be given as in an ordinary type. However, one has to be very careful with new literals and the interference between inherited and added operators.

When literals are added, the result of inherited operators when applied to these new literals must be defined (by equations). When operators are added one should remember the invisibly renamed operators and their associated equations.

Equations for definition of the added operators should be consistent with both the equations involving inherited and not inherited operators.

After this list of warnings, let's look at some examples.

a) Suppose the newtype colour is completed and available. This type is based on taking and mixing beams of light of primary colours. It would be a lot of thinking and writing and/or copying to define something similar for taking and mixing paint.

A nice solution to this problem is to make the newtype colour into a generator by just two replacements:

1) the first line

```
NEWTYPE Colour
```

becomes

```
GENERATOR Colour( TYPE Primary_colour )
```

2) the keyword ENDNEWTYPE becomes ENDGENERATOR.

The generator can now be renamed when instantiated. Suppose the former sort Primary_colour is called Light_Primary, and the sort Paint primary is defined as:

```
NEWTYPE Paint_primary
    LITERALS Red, Yellow, Blue
ENDNEWTYPE Paint_primary ;
```

Now its very easy to define two similar types, one for light and one for paint:

```
NEWTYPE Light_colours Colour( Light primary) ENDNEWTYPE ;
NEWTYPE Paint colours Colour( Paint_primary ) ENDNEWTYPE ;
```

So far no problem, but how can one see the difference between the light colour Take(Red) and the paint_colour with the same syntax? If the need to distinguish between the two is felt inheritance can help. Instead of Light_colours and Paint colours the types Light and Palette are defined via inheritance, renaming operator Take:

```
NEWTYPE Light
INHERITS Light_colours
OPERATORS ( Beam = Take, Mix, Contains )
ADDING
    LITERALS White
    AXIOMS
      White == Mix( Red, Mix( Yellow, Beam( Blue)) )
ENDNEWTYPE Light ;
```

Now newtype Light has the literals from Light_colours plus the literal White. Light_colours has no literals of its own (because it uses the literals of Light primary), so White is the only literal of Light. The operators and equations of Light are the same as those of Light colours, except that the operator name Take is replaced by Beam, and the given equation for White is added. The added axiom states that this added literal becomes an element of the set of terms where all three primaries are mixed.

The newtype Palette has the literals from Paint_colours and the Take operator is replaced by Paint:

```
NEWTYPE Palette
    INHERITS Paint_colours
    OPERATORS ( Paint = Take, Mix, Contains )
ENDNEWTYPE Palette ;
```

b) Suppose one wants to extend the set of integers type (sort Int_set), introduced in the previous section, by an operator that finds the smallest integer in the set. First of all one should ask oneself whether this operator can be introduced in the generator definition to make it available for sets of other things as well. While it could be done, it would restrict Item to have > and < defined. This does not suit all items (e.g. PId) and it may be better to make a newtype with sort New_int_set providing an operator Min.

```
NEWTYPE New_int_set
    INHERITS Int_set
    OPERATORS ALL
    ADDING
      OPERATORS
              Min : New_int_set -> Integer
```

```
        AXIOMS
                Min( Empty_int set) = Error! ;
                Min( Add( Empty_int_set, x ) ) == x ;
                Min( Add( Add( nis, x ), y ) ) ==
                                IF y < Min( Add( nis, x) )
                                THEN y
                                ELSE Min( Add( nis, x) )
                                FI
ENDNEWTYPE New_int_set ;
```

Because addition after a generator instantiation is rather common, the text starting with ADDING and ending just before the ENDNEWTYPE can be given inside the generator instantiation. An example is given in § 5.4.1.12 of the Recommendation.

D.6.3 *Views on equations*

When introducing a new data type it is essential to introduce enough equations. In the sequel three views on equations are given which help in constructing equations.

D.6.3.1 *General requirements*

No matter how one constructs the equations, the following facts must hold:

a) Each operator appears at least once in the set of equations (except for pathological cases).

b) All true statements can be derived from the equations. Either they are stated as axioms or they can be derived by substituting equivalent terms in the equations.

c) No inconsistency can be detected; i.e., it cannot be derived from the equations that True=False.

A procedure to find equations can be expressed in informal SDL as in figure D-6.3.1.

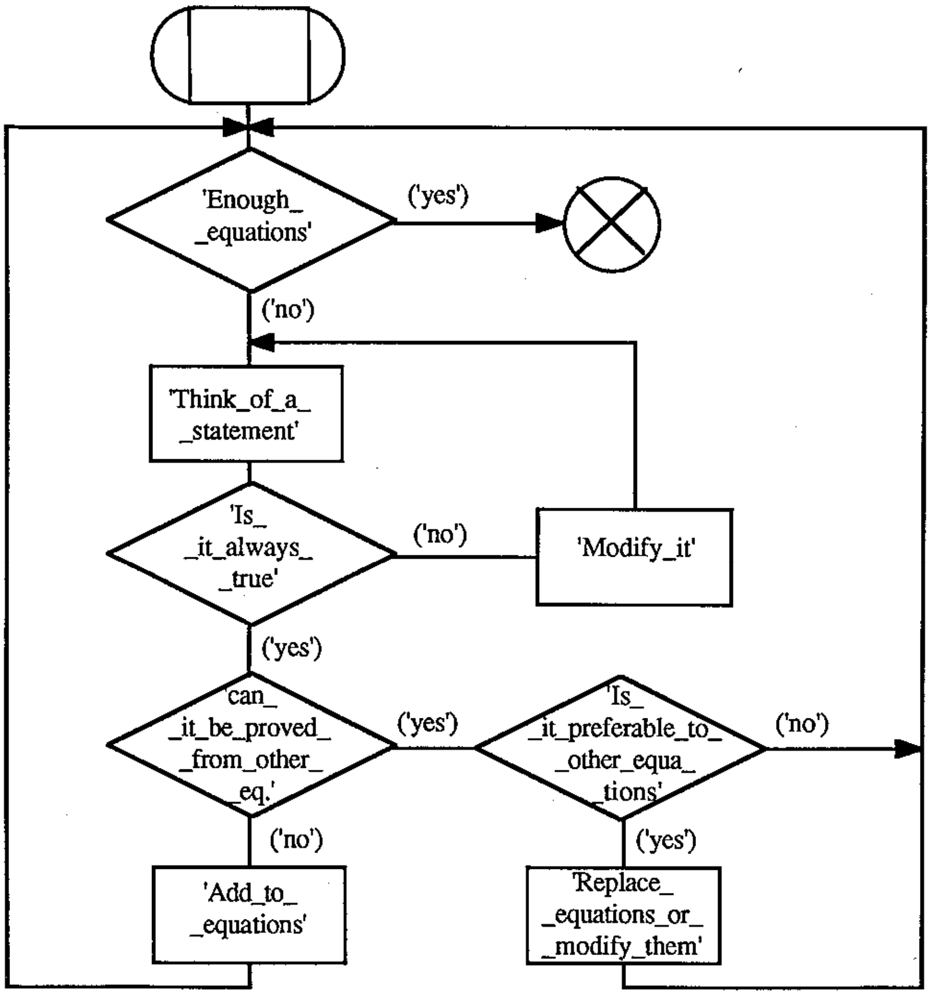


FIGURE D-6.3.1

Procedure to find equations in informal SDL

D.6.3.2 *Function application on constructors*

In general the set of operators has a subset of operators known as 'constructors and functions'.The constructors can be used to generate all the values (equivalence classes) of the sort. In this approach literals are regarded as operators without arguments.

Examples:

a) The boolean type has its literals as constructors.

b) The natural type has the literals 0 and the operator Next as its constructors; every natural can be constructed with 0 and Next only.

c) The generator for sets has the literal empty_set and the add-operator as its constructors; every set can be constructed using empty_set and add only.

d) The integer type can be constructed by the literals 0 and 1, the operators + and unary minus.

It should be noted that sometimes there are several, possible choices for the set of constructors. Any choice will do in the remainder of this section, but mall sets are in general the best.

Now all functions are treated one by one. For each argument of a function all possible terms consisting of constructors only are listed. To avoid problems with infinite numbers of terms quantification should be used.

Examples:

a) For the natural numbers this list can be reduced to

0
Next (n) where n is any natural number.

b) For the sets a possible list is

```
empty_set
Add( s, i ) where s is any set and i is any item.
```

If in the right hand side term of an equation having (s,i) in the left side, there is a difference between s being empty or not empty, the list can be rewritten as follows:

```
empty_set
Add( empty_set, i ) where i is any item,
Add( Add( s, i ), j ) where s is any set and i, j any item.
```

After creation of this list the left hand sides of the equations are obtained by applying each function to any combination of arguments from the list. Value identifiers in different arguments are given different names. The same procedure as given above for functions can be followed for constructors; in that case it gives relationships between terms where constructors are used in different orders.

Examples:

a) For the multiplication operator for natural numbers with signature

```
"*" : Natural, Natural -> Natural
```

this procedure gives the left hand side of the following (incomplete) equations. The user should provide the right hand side.

```
             0 * 0            == . . .;
             0 * Next(  n  )  == . . .;
Next (  n  )  * 0             == . . .;
Next (  n  )  * Next (  m  )  == . . . ;
```

b) For the operators Is_in and Delete, in generator Set (§ D.6.2.1), this approach is already used.

c) For sort Colour the constructors are Take and Mix. An operator similar to Contains in § D.6.1.4.2 must be defined for the arguments

```
Take( p ) where p is any primary,
Mix( p, c ) where p is any primary and c is any colour
```

Because it is promised in § D.6.1.4.2 to give neat equations for this operator they are given in full:

```
Contains( Take( p ), q) == p = q ;
Contains( Mix( p, c ), q) == (p = q) OR Contains( c, q  ) ;
```

This procedure for construction of equations may produce more equations than necessary, but it is very safe. E.g., in the example of the multiplication of natural numbers above it is likely that the commutative property of multiplication will be stated and therefore only the last (or the second) equation of the first three will be needed.

The procedure described in this section can be used in combination with the procedure described in the previous section where it is helpful in the task "Think_of a _statement".

D.6.3.3 *Test-set specification*

Another way of looking to the equations is from the implementation point of view. If the operators are implemented as functions in a programming language, the equations, describe how these functions have to be tested.

Evaluate the expression corresponding to the left hand side of an equation, do the same for the right hand side of that equation and look whether they are equivalent. What might cause problems is the FOR ALL construct. Often a very pragmatic approach solves the problem:

```
Instead of          FOR ALL i IN Integer
the tester can use  FOR ALL i IN (-10, -1, 0, 1, 10 }
and this will do in most cases.
```

Thinking of equations as requirements for the implementation can be helpful in the task "Think_ofa_statement" in the procedure of § D.6.3.1.

D.6.4 *Features*

This section describes some of the whistles and bells of SDL, i.e. features that are rarely needed or one can mostly do without, but sometimes they make life much easier.

D.6.4.1 *Hidden operators*

Sometimes the set of equations can be simplified or made more readable if an extra operator was introduced, but this operator should not be used in the processes. This means that the operator is visible inside, but hidden outside the type definition.

This result can be reached by defining a 'hidden type', i.e., a type that the user should not use. From this hidden type the user can inherit all operators he is allowed to access; it is the inherited type that should be used. Checking the correct use can be done by inspection of all variable declarations (no variables of the sort introduced by the hidden type should appear).

The feature of hidden operators means that the same can be reached by restriction of the visibility of these operators to the equations only. This is done by putting an exclamation mark after the operator.

Example:

The proper way of making a one element set in the generator Set is

```
Add( empty_set, x )
```

and this is the way each user should do it. In the equations the specifier can use a special operator, e.g.:

```
Mk_set! : Item -> Set ;
```

defined by the equation:

```
Mk_setl ( itm ) == Add( empty_set, itm )
```

which can be used in partial type definitions, but not in SDL process body.

D.6.4.2 *Ordering*

When ordering of elements of a sort has to be specified this means in general that four operators must be defined (<, <=, >, >=), and the standard mathematical properties (transitivity, etcetera). If there are a lot of literals, a lot of equations must be given as well. For instance, the predefined data type Character is defined this way.

SDL provides a feature that overcomes these lengthy, unreadable and boring type definitions: the shorthand ORDERING.

ORDERING is given in the list of operators, preferably at the beginning or at the end of this list. This introduces the ordering operators and the standard equations. When ORDERING is specified the literals, if any, have to be given in ascending order.

Example:

```
NEWTYPE Even_decimal_digit
    LITERALS 0, 2, 4, 6, 8
    OPERATORS
              ORDERING
ENDNEWTYPE Even_decimal_digit ;
```

Now the ordering 0 < 2 < 4 < 6 < 8 is implied.

In § D.6.2.2 (Inheritance) a warning is given to be careful if literals are added to an inherited sort. This is an excellent place to show why.

Suppose an extension of sort Even_decimal_digit is wanted as follows:

```
NEWTYPE Decimal_digit
    INHERITS Even_decimal_digit
    OPERATORS ALL
    ADDING
    LITERALS 1, 3, 5, 7, 9
    AXIOMS
        0 < 1 ;         1 < 2 ;
        2 < 3 ;         3 < 4 ;
        4 < 5 ;         5 < 6 ;
        6 < 7 ;         7 < 8 ;
        8 < 9
ENDNEWTYPE Decimal_digit ;
```

The axioms given here cannot be omitted. Without these axioms there only is a so called partial ordering:

```
                0 < 2 < 4 < 6 < 8
and             1 < 3 < 5 < 7 < 9 .
```

With the axioms above there is a complete ordering:

```
0 < 1 < 2 < 3 < 4 < 5 < 6 < 7 < 8 < 9
```

but with the axiom "9 < 0" instead of the set of axioms above, the complete ordering would be :

```
1 < 3 < 5 < 7 < 9 < 0 < 2 < 4 < 6 < 8 .
```

D.6.4.3 *Sorts with fields*

As shown in § 5.4.1.10 of the Recommendation one can define a structured sort without special constructs, but structured sorts are both common and useful, which justifies some extra constructs in the language.

A structured sort should be used when an object value is formed by the association of values of a number of sorts. Each value of this association is characterised by a name, called the field name. The sort of a field is fixed.

Examples:

```
NEWTYPE Subscriber
    STRUCT  numbers Number_key ;
            name    Name_key ;
            admin   Administrative ;
ENDNEWTYPE Subscriber ;

NEWTYPE Name_key
    STRUCT  name,
            street  Charstring ;
            number  Integer ;
            city    Charstring;
ENDNEWTYPE Name_key ;
```

With the structured sort some operators are implicitly defined:

a) the constructor operator, "(." before, and ".)" after the field values;

b) the field selection operators, the variable of the structured sort followed by an ! and the field name, or followed by the field name in parentheses. The variable followed by an ! should not be confused with the hidden operator (§ D.6.4.1)

An example is given in Figure D-6.4.1

```
PROCESS Some_process;
    DCL  na, st, where    Charstring    ,
         nu               Integer       ,
         nk               Name_key      ;
     /* . . .    Text where values are assigned to the variables na and st */
     TASK nk := (. na, st, 5, 'London' .) ;
     /* . . .    Text where no assignments to nk take place */
     TASK where := nk!city ;
     /* Now where='London' holds */
ENDPROCESS Some_process;
```

FIGURE D-6.4.1

Example of use of implicit structured sort operators

D.6.4.4 *Indexed sorts*

An indexed sort is a sort for which the type has Extract! as an operator name. In the predefined data types the generator Array is such a type. The array is one of the most common examples of an indexed type.

For the hidden operator Extract! there is a special concrete syntax that should be used outside the type definitions.

It may be thought that type Index in the predefined generator Array should be a 'simple' type, like Integer, Natural or Character. There is, however, no reason why a structure like Name_key cannot be used as Index.

Example:

```
NEWTYPE Subsc_data_base
    Array( Name key, Subscriber )
ENDNEWTYPE Subsc_data_base ;
```

The sorts Name_key and Subscriber are those defined in the previous section. Suppose there is a procedure Bill with one parameter of sort Subscriber, and this procedure is defined in a process that also has a variable Sub_db of sort Subsc_data_base. In this process the following call could appear.

```
CALL Bill ( Sub_db ((. 'P.M.', 'Downingstreet', 10, 'London .))) ;
```

D.6.4.5 *Default value of variables*

As explained in the section on declaration of variables (§ D.3.10.1) it is possible to assign values to a variable directly after the declaration. Some types, however, have a value that (almost) always will be the initial value of a variable. There is a feature that avoids having to write down the initial value at every declaration: the DEFAULT-clause.

As an example one can consider the set. It is very likely that almost all variables, of whatever set one can think of, will be initialized to the empty_set.

The notation:

```
DEFAULT empty_set
```

after the list of equations ensures that every variable of every instantiation of the generator will be initialised to the empty_set of that instantiation, except if there is an explicit initialisation. (See § D.3.10.1).

If it is not trivial that the initial value of all variables of a sort should be the same, then don't use the DEFAULT clause. Otherwise it is hard to avoid surprises.

D.6.4.6 *Active operators*

Users familiar with the SDL'84 Z.104 might wonder what happened to the so called active operators. Well, this feature was deleted because

a) it is not needed because ordinary operators together with procedures and/or macros provide the same expressive power,

b) it destroyed the readability of the equations,

c) a lot of users had problems in using it correctly,

d) it does not really fit in with the abstract data type model based on initial algebras which is the model chosen as the mathematical foundation of this part of SDL.

D.7 *Additional guidelines for drawing and writing*

For each concept of SDL the Recommendation specifies the main way to represent the concept. For example the task concept is represented in SDLIGR by means of a rectangle and by means of the keyword TASK in SDL/PR.

The User Guidelines supplement the Recommendation in specifying rules for thawing and writing, applicable to all concepts, in order to emphasize what may be considered appropriate, wrong or bad-looking drawing/writing.

D.7.1 *SDL/GR guidelines*

D.7.1.1 *General*

General guidelines for drafting diagrams are:

- The sequence of reading should be top-down and left-right.
- Diagrams should not contain too many information at the same level. It is often suitable to partition large diagrams into sub-diagrams covering different parts or aspects; for instance by using the referencing mechanism.

D.7.1.2 *Inlets and outlets*

Inlets and outlets to/from any symbol should be drawn vertically (corresponding to the top-down reading), and only where this is not practical, horizontal inlets and outlets should be used.

Exceptions to this general rule are:

- Decision with two or three outlets are usually drawn with two horizontal outlets (plus a vertical one).
- Macro calls use horizontal and vertical connections.
- Both in and out connectors have usually horizontal inlets and outlets.

Diagonal lines should only occur in exceptional cases (e.g. for channels and signal routes).

Vertical inlets and outlets should divide the symbol into two parts having the same horizontal lenght.

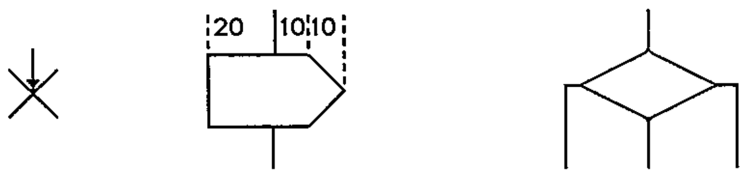


FIGURE D-7.1.1

Properly drawn inlets and outlets

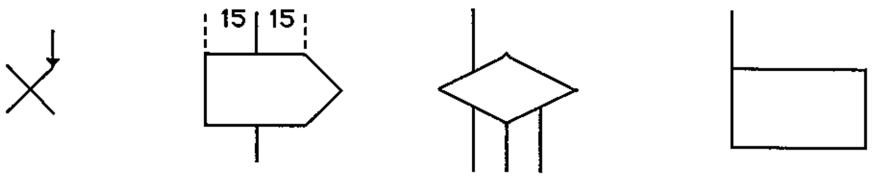


FIGURE D-7.1.2

Inproperly drawn inlets and outlets

D.7.1.3 *Symbols*

a) Symbols should be drawn so that the vertical and horizontal axis coincide with the two axis of the paper.

b) Vertical mirroring of symbols is just allowed for input, output, text extension and comment symbols. (See Figure D-7.1.3).

c) The general ratio between height versus lenght for all symbols in graphs and also for referencing symbols is 1:2.

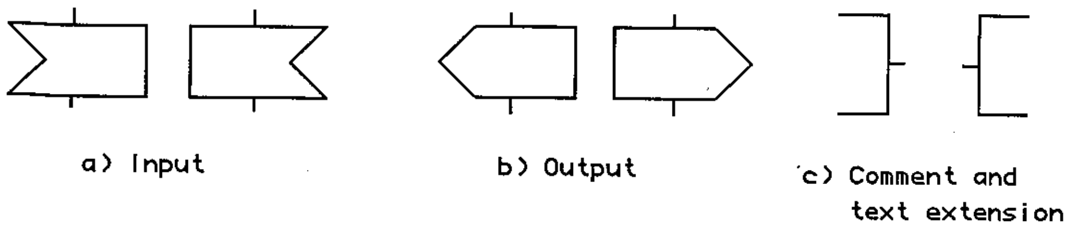


FIGURE D-7.1.3

The four allowed mirrored symbols

D.7.1.4 *Template*

A template for the symbols of SDL/GR has been provided in the inside back cover of this Fascicle. A schematic draft for the template is given in Figure D-7.1.4.

In the figure the symbols for Input, Output, Decision, Alternative, Process, Start, Task, State, Save, Service reference, Connectors and Stop are shown direcly, in three sizes, i.e. 20 x 40 mm, $20 / \sqrt{2}$ x $40 / \sqrt{2}$ mm and 10 x 20 mm. The internal ratio is only shown for the largest size.

The symbols for Procedure Call, Macro Call and Create can be constructed from the task symbol by drawing the extra horizontal or vertical line(s) indicated in the Figure.

The Procedure Start symbol can be constructed from the Process Start symbol by drawing the indicated extra vertical lines.

The Return symbol is the combination of the Connector and the Stop symbol.

The Comment, the Text Extension and the Signal List symbols are drawn using the Task symbol.

The Priority Input and Priority Output symbols can be constructed from the Input and Output symbols by drawing the indicated extra line.

The Procedure Reference symbol can be constructed from the Process Reference symbol by drawing the two extra vertical lines as indicated.

The Enabling Condition and the Continuous Signal symbols can be drawn using the Stop symbol.

Channels, Signal Routes and the line to the Text Extension symbol are solid lines. The line to a comment symbol is a dashed line with the ratio 1:1.

The text symbol is drawn using the Task symbol and folding the rightmost upper corner as indicated.

All the recommended symbols are defined in the Recommendation. An overview of the recommended symbols can be found in Annex C – Graphical Syntax Summary.
The three shown sizes are the preferred ones. If other sizes are used the ratio should still be same (i.e. 1:2). The sizes shown, i.e. lenght 40 mm, 28 mm and 20 mm permits photo reduction from A3 to A4 paper with compatible symbol sizes (as 40 mm / $\sqrt{2}$ = 28 mm and 28 mm / $\sqrt{2}$ = 20 mm).

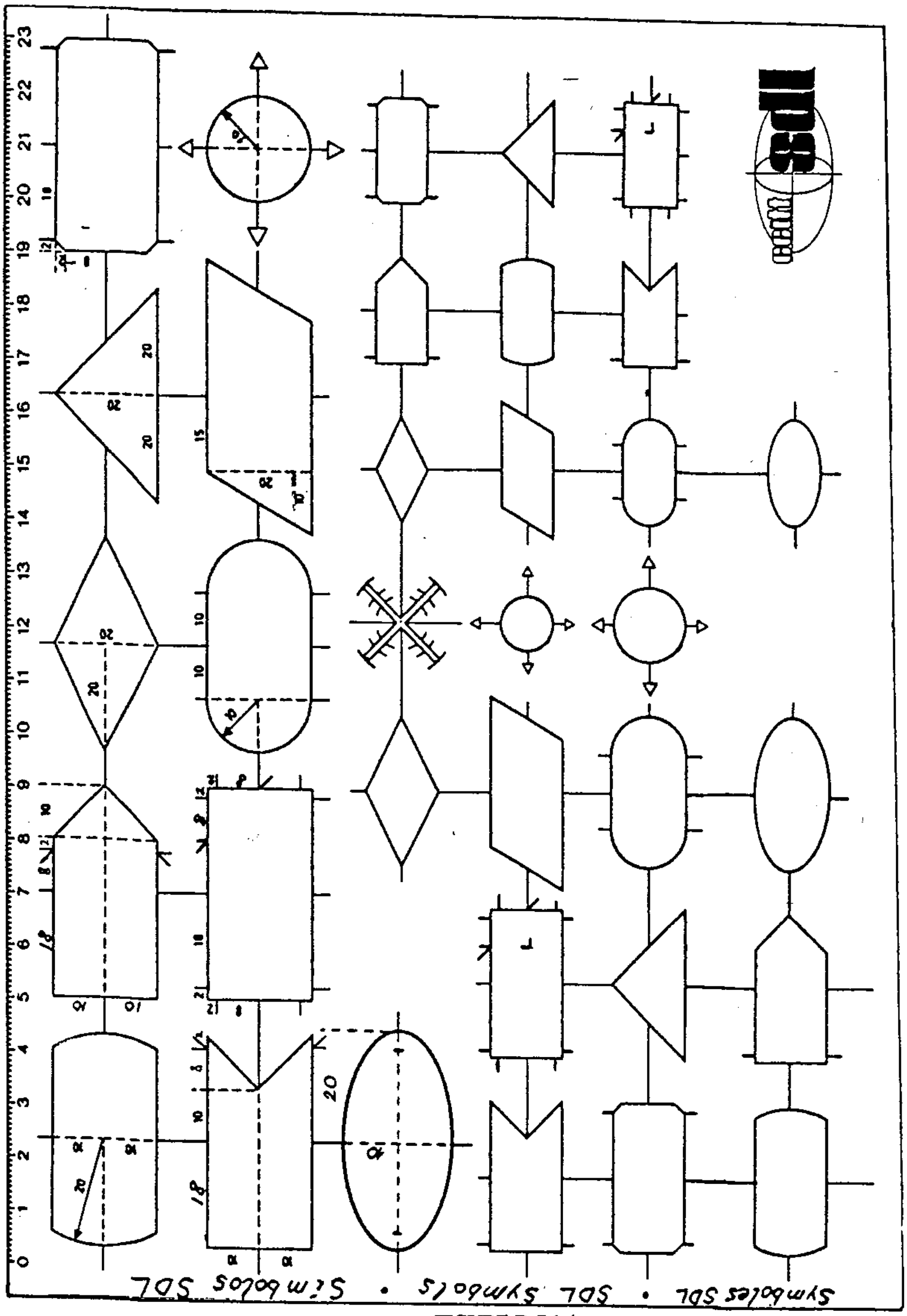


FIGURE D-7.1.4

Schematic draft of the template

D.7.2 *SDL/PR guidelines*

General guidelines for drafting textual SDL are:

- The sequence of reading should be top-down and left-fight.
- The text should be divided into parts covering different aspects.
- Comments on the statements level should begin at the same column.
- The lines should be indentated. Indentation can follow the usual hierarchy of SDL concepts as shown in the example of Figure D-7.1.5.

```
SYSTEM A;
    SIGNAL S1,S2;
    BLOCK B;
        PROCESS P;
            START;
            NEXSTATE ST1;
            STATE F;
                INPUT G: . . .
                INPUT F: . . .
                . . .
            ENDSTATE F;
        ENDPROCESS P;
    ENDBLOCK B;
ENDSYSTEM A;
```

FIGURE D-7.1.5

Example of textual indentation in SDL

D.8 *Documentation*

D.8.1 *Introduction*

A document is defined by ISO as "a limited and coherent amount of information stored on a medium in a retrievable form". It should therefore be considered as a logical unit which is strictly delimited. Documents are used for conveying all information related to a system which is specified using SDL.

When paper is used as the physical medium for storing a document, the term document is often incorrectly applied to the sheets of paper rather than their logical contents. With the growing use of magnetic storage media, the term is returned to its original meaning.

This chapter is concerned with the logical organization of documents rather than their physical organization. This is left to the users discretion. The similarity in requirements of both the logical and physical organization of documents means that some useful hints may be offered in the following text to aid a user in setting up a physical organization for documents.

By splitting the information into a suitable number of documents, the system can be made more readable and manageable.

The language does not recommend certain documents or document structures. However, some proposals are given to aid the user in handling the documents.

D.8.2 *Types of system representation*

When specifying a system using SDL the result is a set of definitions in SDL/GR and/or SDL/PR.

These definitions may be nested or sequential depending on type of system representation, being either hierachical or flat. In fig. D-8.2.1 and D-8.2.2 a system is described with the two alternative representations in SDL/GR. The two figures are not complete system specifications as, for simplicity, only inputs and outputs are shown and possible signal and data definitions are missing.

Of course it is allowed to use a mixture of the representations when a system is specified.

When the definitions are sequential as in fig. D-8.2.1 they are 'referenced', a mechanism which is possible for definitions both in SDL/PR and SDL/GR.

Looking at a system specification as a set of definitions a document can be seen as a container for these definitions.

If the system is small and hierarchical as in fig. D-8.2.2 one single document is enough. If a flat representation is used as in fig. D-8.2.1 more documents can be used e.g. one document per definition.

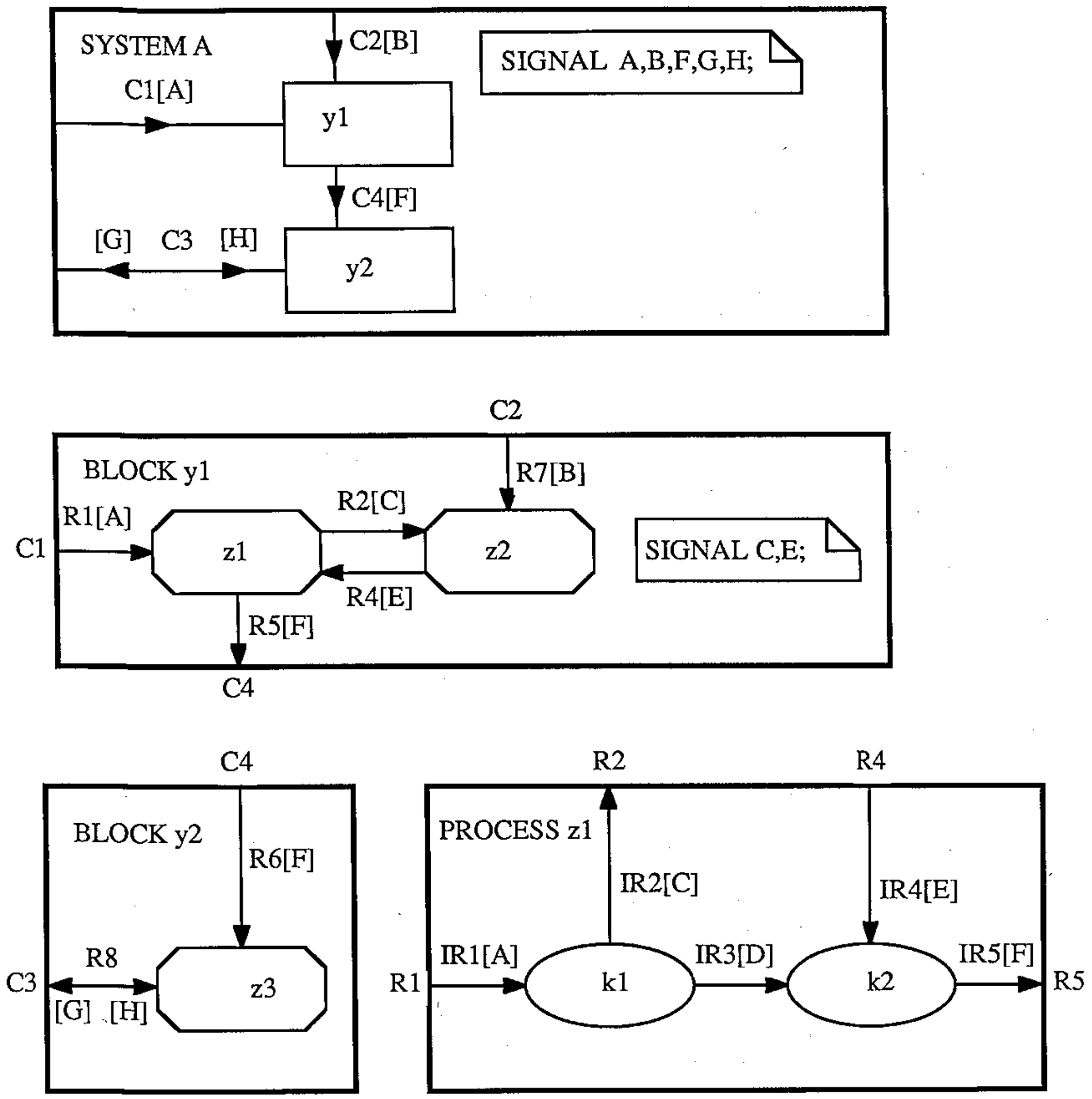


FIGURE-D-8.2.1a

Example of sequential (referenced) diagrams used for a flat system representation

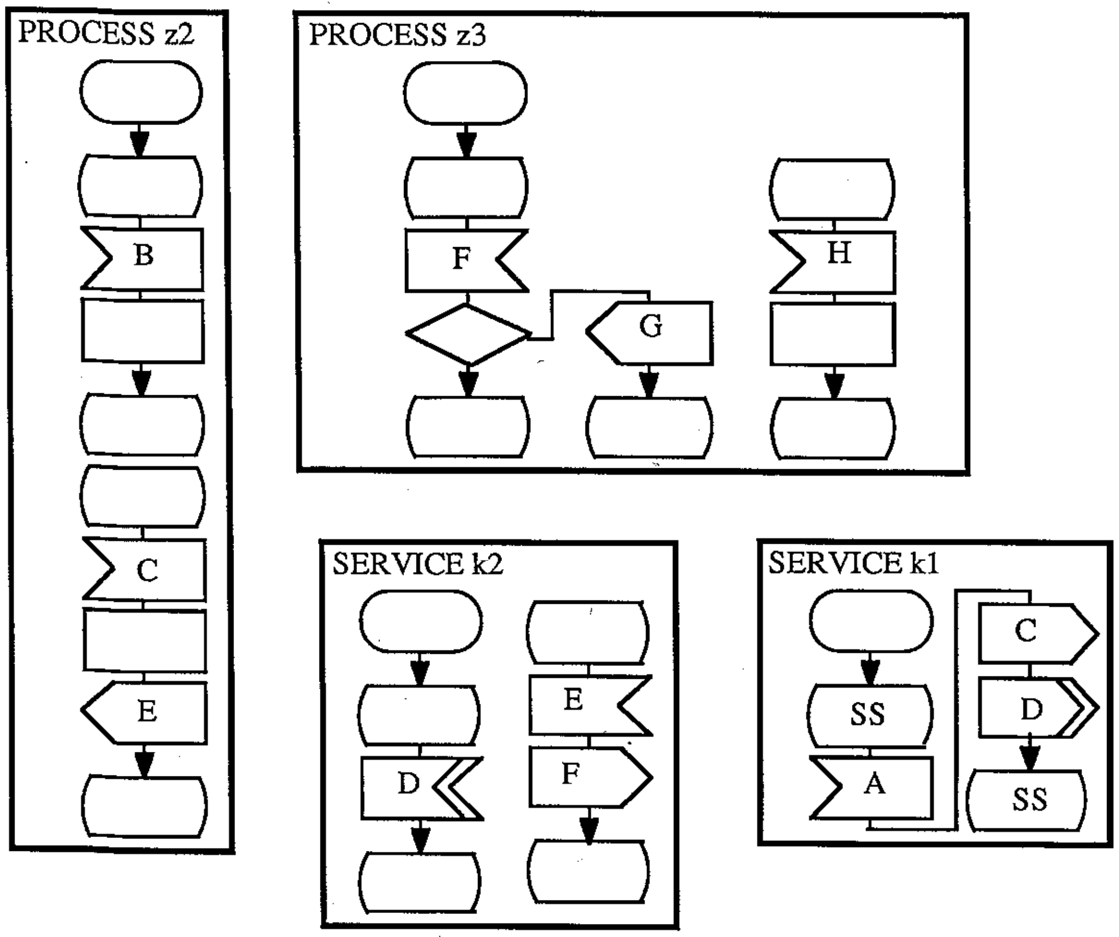


FIGURE-D-8.2.1b

Example of sequential (referenced) diagrams used for a flat system representation

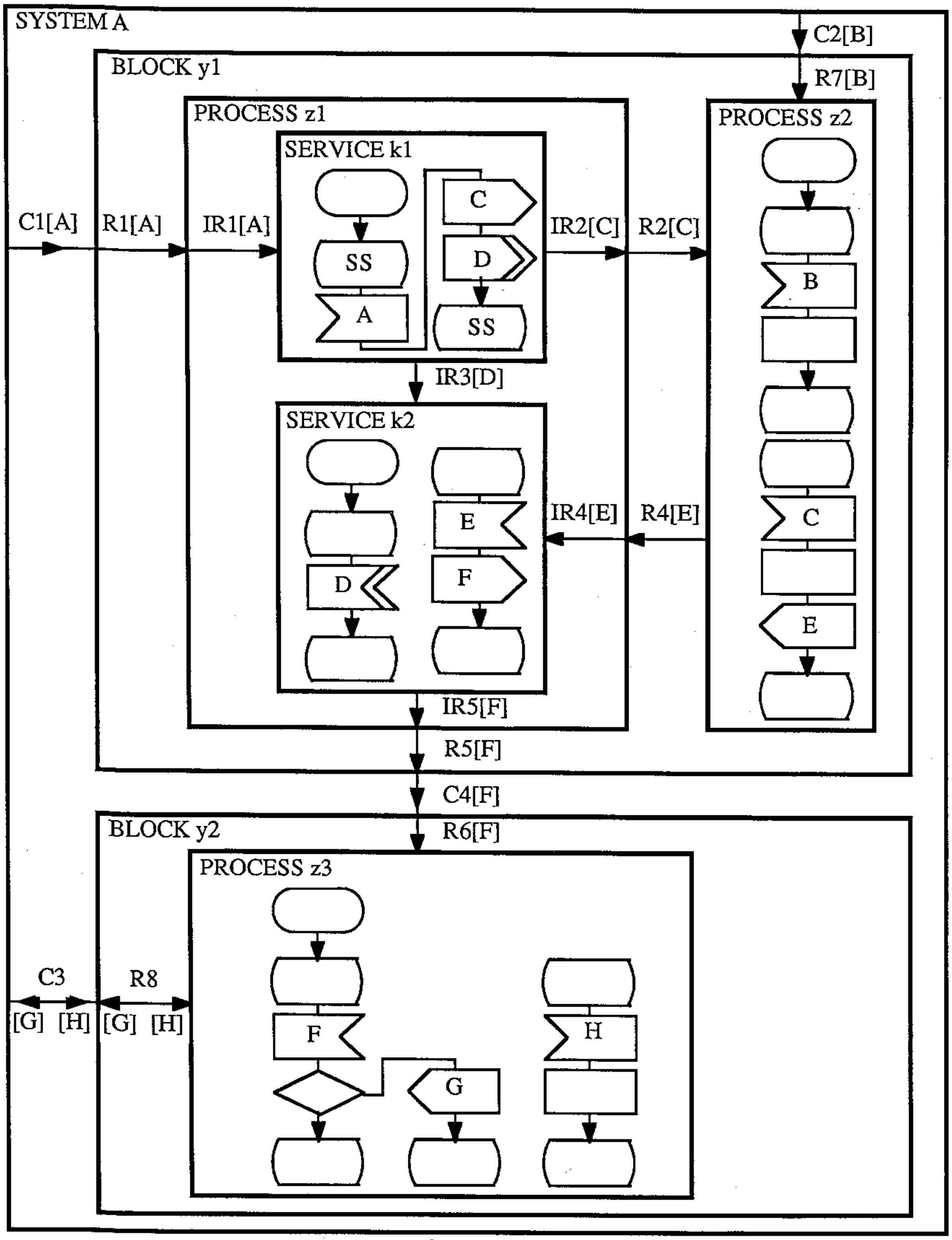


FIGURE D-8.2.2

Example of nested definitions used for a hierarchical system representation

When choosing the type of system representation it is necessary to consider the type of documents desired. In order to have one document per definition, a flat representation is necessary. If one single document for the whole system specification is desired, a hierarchical representation is necessary.

The normal case is probably a mixture of these representations.When deciding this mixture the following rules are applicable:

1) a definition should not be divided between several documents.

2) if a definition is wanted in a separate document it must be referenced, not nested.

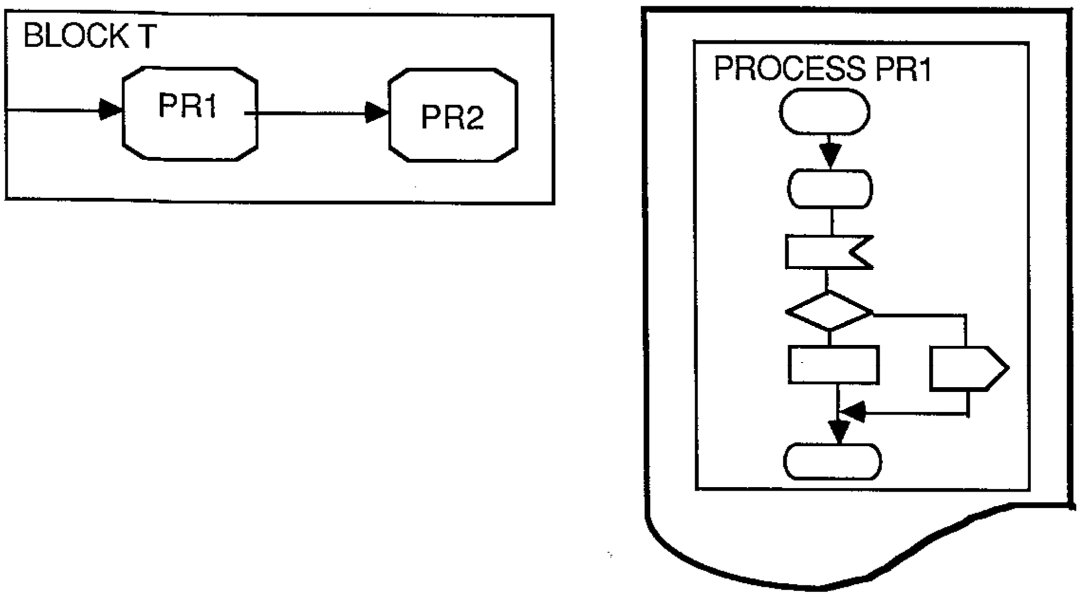


FIGURE D-8.2.3

A referenced definition can be placed in a separate document

3) when using the logical page concept to split a diagram in several diagram pages, the diagram pages should coincide with the phisical pages of the document (see fig. D-8.2.4).

4) if a diagram is more than one page it must be referenced, not nested.

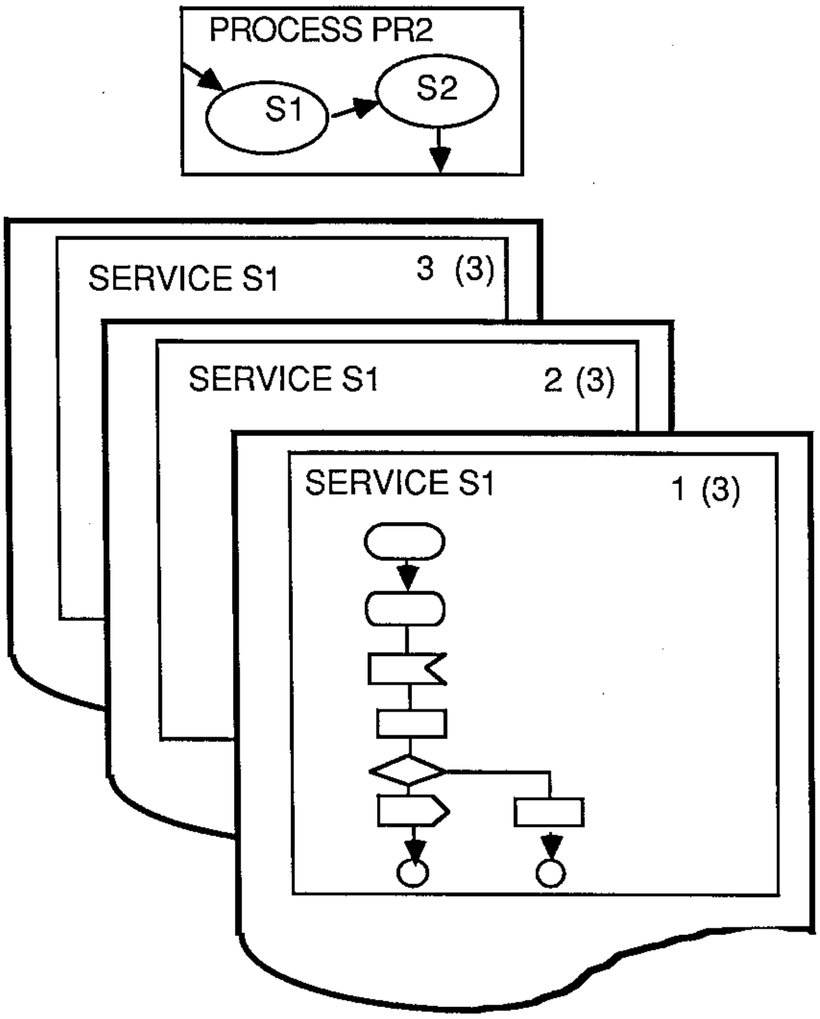


FIGURE D-8.2.4

A referenced service diagram partitioned into 3 pages

D.8.3 *Document structure*

The set of documents covering the whole system definition can be structured.
A document structure , where documents refer to sub-documents, can be attached to any entity in the system such as system, block and processes. See Figure D-8.3.1.

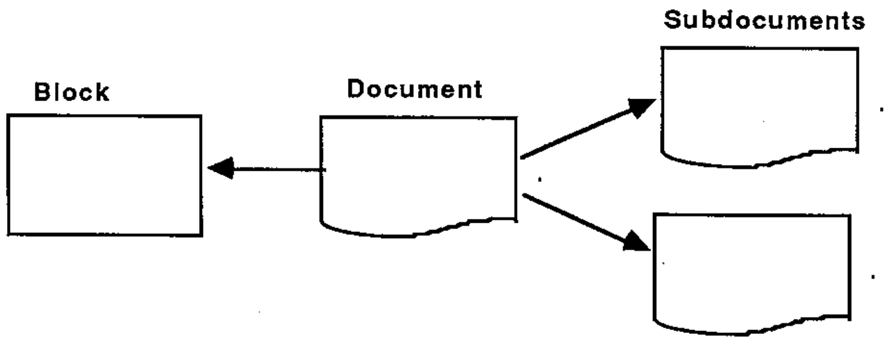


FIGURE D-8.3.1

A document structure

D.8.4 *Referencing mechanism*

The reference mechanism-in the language, where concept names are used for references between concepts, can also be used for references between documents. This is a natural approach when a document coincides with a definition.

D.8.5 *Classification of documents*

Documents can be classified in accordance with the types of definitions they contain.

In this classification at least the process or service definitions, describing the dynamic behaviour of the system, should be placed in separate documents. These documents can also include variable definitions.

A possible document structure for a system is given in Figure D-8.5.1.

In this example the channel and signal route definitions are included in the documents for system, block and process definitions. The signal definitions, the data definitions and the signal list definitions are placed in separate documents and it has been assumed that all data definitions are at system level.

The procedure definitions, the macro definitions and the service definitions form sub-documents to the process document.

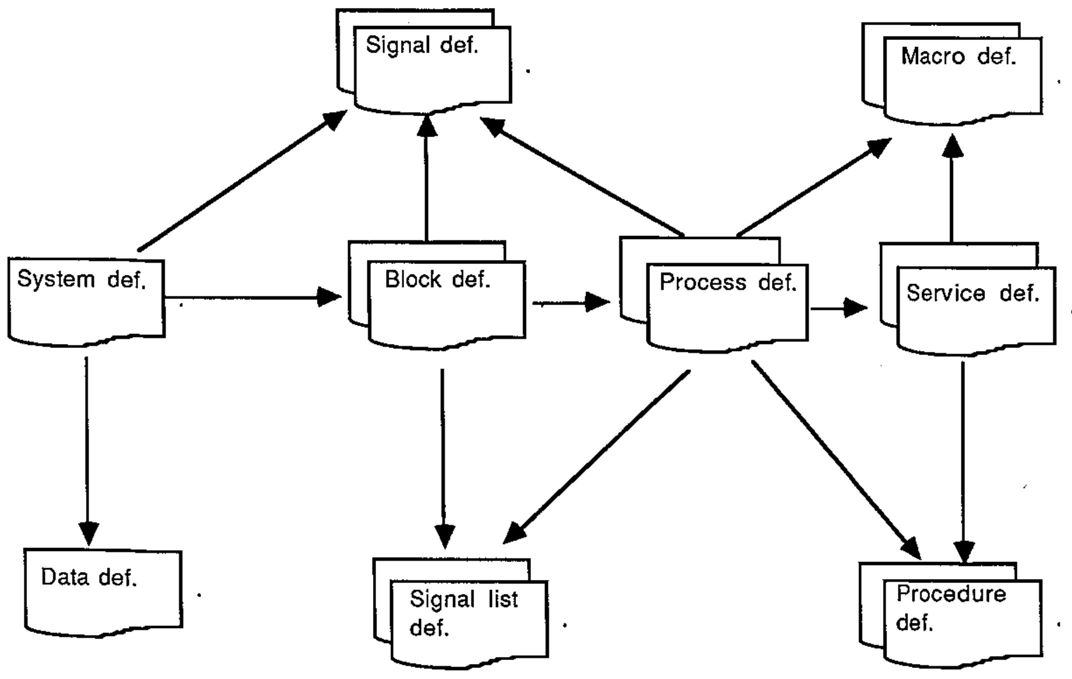


FIGURE D-8.5.1

A possible document structure for a system

If different services together form a system function, the services can be described in a common document.

The different service definitions can be placed after each other in a service document but it is also possible to have them side by side on the same document page. The latter document layout gives a good understanding of the interaction between the services. Figure D-8.5.2 is an example of a document page in a service document.

For large system specifications "tables of content" for the system should be provided to indicate where to find the states, the inputs, the outputs etc. In addition the tables of content should also contain all the concepts i.e. where to find the definitions and where they are used. Examples are system, blocks, channels, signals, processes, services, macros, procedures.

Such tables of content for the system may be formed as separate documents

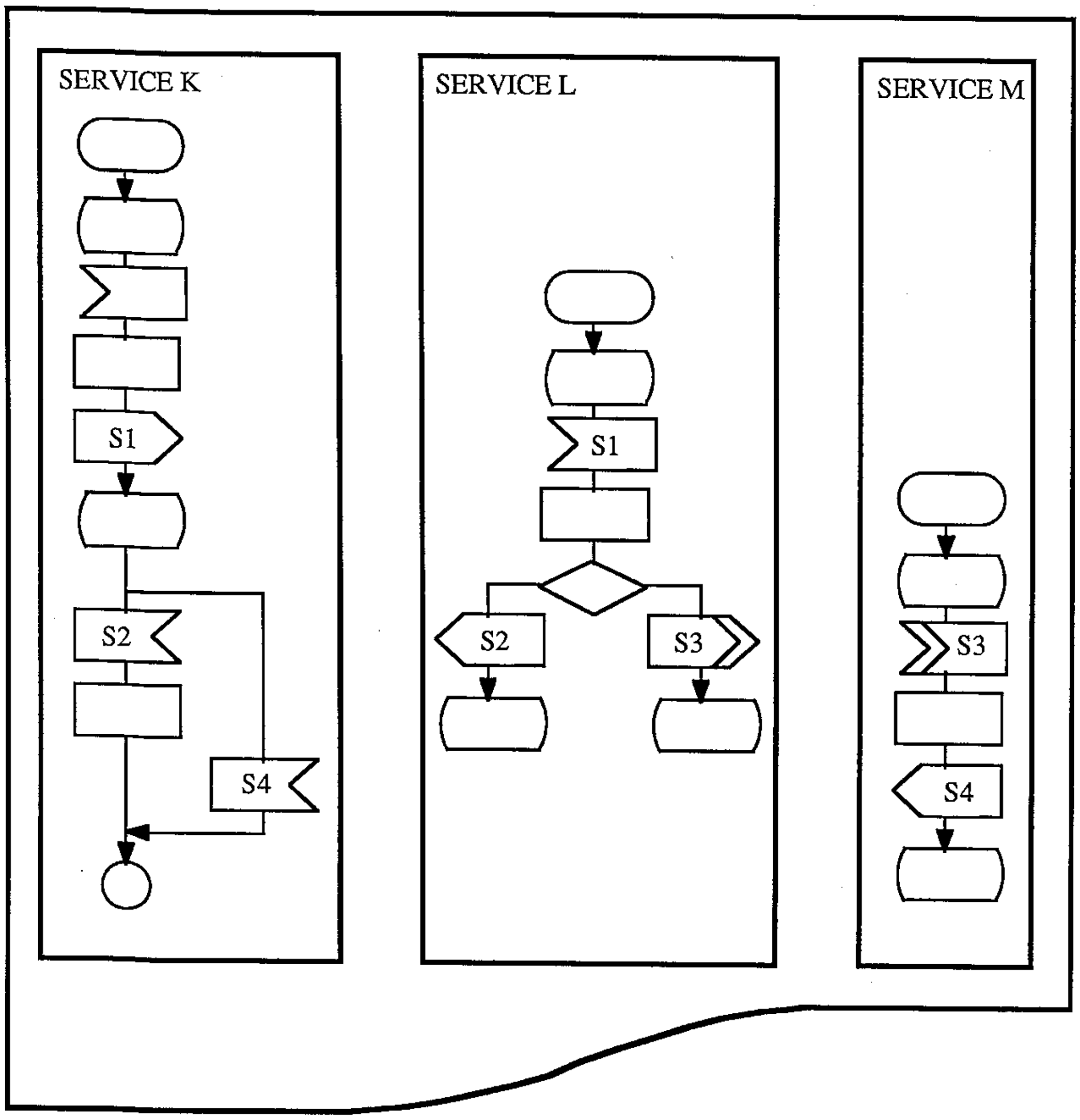


FIGURE D-8.5.2

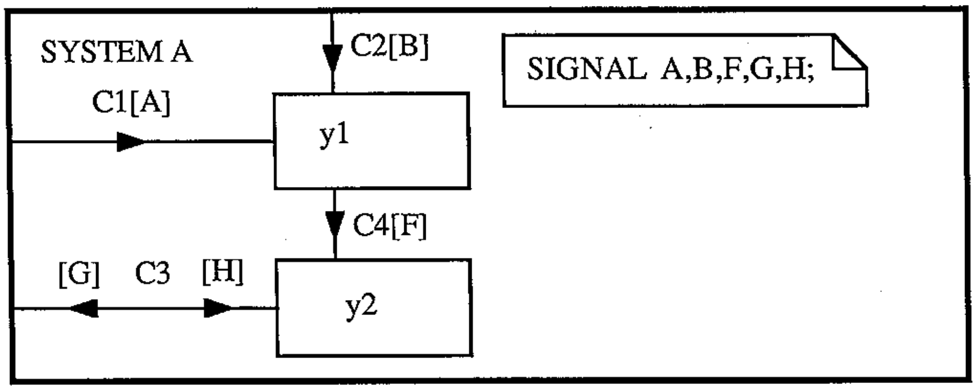
A page in a service document

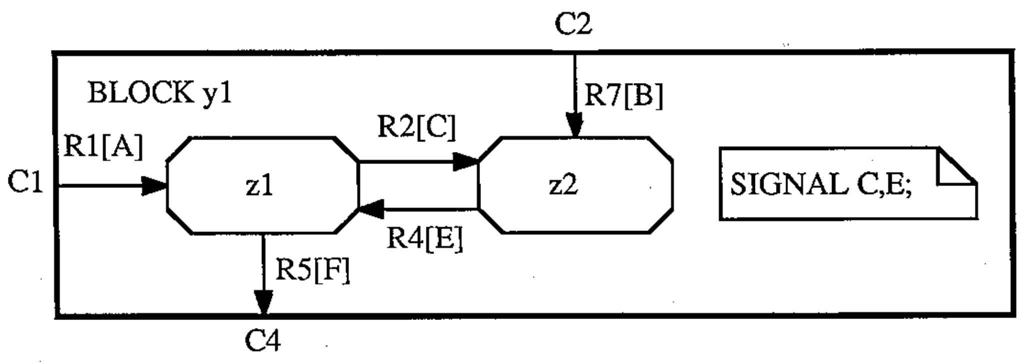
D.8.6 *Mixing of SDL/GR and SDL/PR*

When specifying a system SDL/GR and SDL/PR can be used together.

The concepts of system, block, process, service, procedure, macro, channel etc can either be specified in SDL/GR or SDL/PR.
Change from SDL/PR to SDL/GR or from SDL/GR to SDLIPR is performed using the reference mechanism in the language. A concept which is referenced in SDL/PR can be specified in SDL/GR and a concept, referenced in SDL/GR , can be specified in SDL/PR.

Figure D-8.6.1 is a "complete" system specification using a mixture of SDL/PR and SDL/GR. It is the same system as in fig. D-8.2.1 and D-8.2.2. Each definition in the example can be located in a separate document.





```
BLOCK y2;
    SIGNALROUTE R8FROM ENV TO z3 WITH H;
                  FROM z3 TO ENV WITH G;
    SIGNALROUTE R6FROM ENV TO z3 WITH F;
    CONNECT C3 AND R8;
    CONNECT C4 AND R6;
    PROCESS z3 REFERENCED;
ENDBLOCK y2;
```

```
PROCESS z1;
    SIGNALROUTE IR1 FROM ENV TO k1 WITH A;
    SIGNALROUTE IR2 FROM k1 TO ENV WITH C;
    SIGNALROUTE IR3 FROM k1 TO k2 WITH D;
    SIGNALROUTE IR4 FROM ENV TO k2 WITH E;
    SIGNALROUTE IR5 FROM k2 TO ENV WITH F;
    CONNECT R1 AND IR1;
    CONNECT R2 AND IR2;
    CONNECT R4 AND IR4;
    CONNECT R5 AND IR5;
    SERVICE k1;
        START;
        STATE SS;
            INPUT A;
            OUTPUT C;
            PRIORITY OUTPUT D;
            NEXSTATE SS;
    ENDSERVICE k1;
    SERVICE k2 REFERENCED;
ENDPROCESS z1;
```

FIGURE D-8.6.1a

A system specification with a possible mixture of SDL/GR and SDL/PR

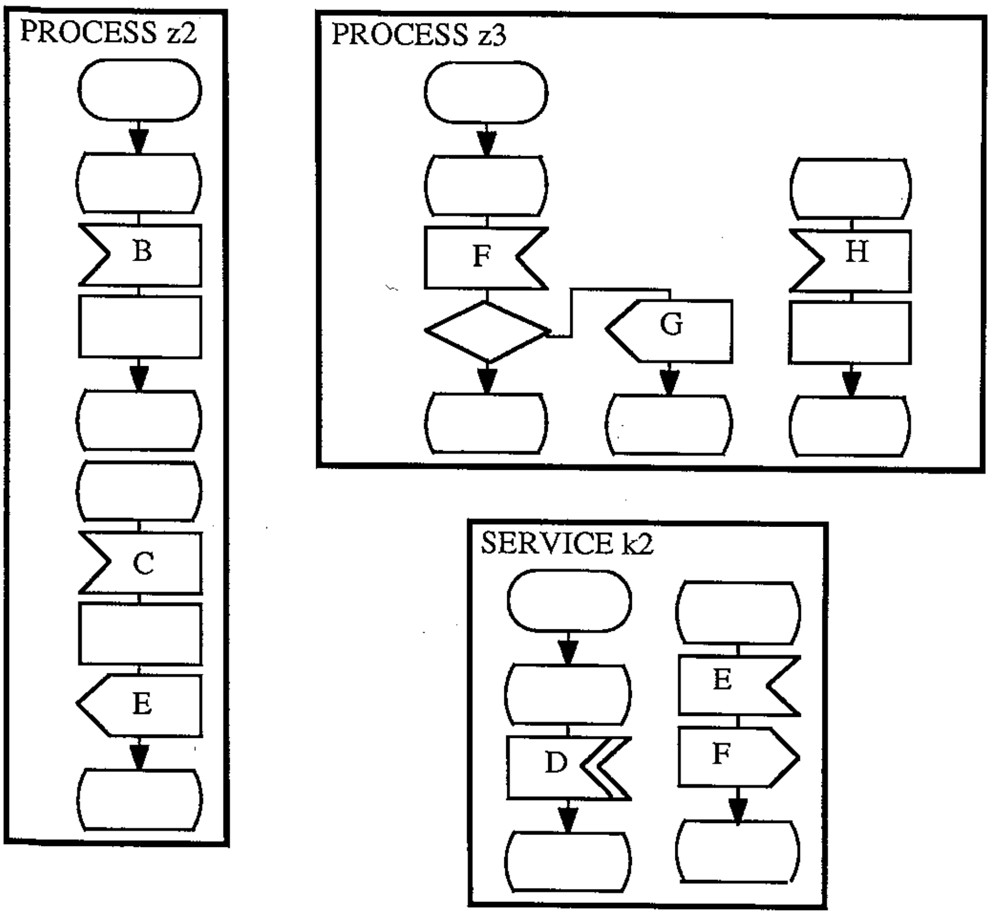


FIGURE D-8.6.1b

A system specification with a possible mixture of SDL/GR and SDL/PR

D.9 *Mappings*

This describes some aspects of the mapping between SDL and CHILL (§ D.9.1), and the mapping between SDL/GR and SDL/PR (§ D.9.2).

D.9.1 *Mapping between SDL and CHILL*

In the following some possible ways of mapping SDL to CHILL are illustrated. This is done briefly and is not intended to be either exhaustive or to suggest that any of these ways should be used in practice.

The discussion on the mapping should not only consider the available CHILL compiler and the target machine but be more general. The mapping is a very complex intellectual activity and it is only through experience that designers/programmers can decide on a particular CHILL program structure to be used to implement a particular SDL representation. This also applies to the representation of the functions implemented by a CHILL program. A one-to-one mapping (if achievable) is not necessarily the best way of using SDL to represent the functions implemented in CHILL.

Using this approach, the overall structure of a CHILL program derived from an SDL diagram appears as shown in Figure D.9.1.1.

Some examples of a mapping between constructs of the two languages are given in Figures D-9.1.2 to D-9.1.5. They cover the following SDL constructs:

- state and reception/save of signals; selection of a nextstate;
- output;
- join;
- decision.

The declaring module contains both the definition and the declaration of all the signals used in the transformed SDL diagram and all of the variables associated with these signals. All these variables are granted to the module representing the functional block of the SDL diagram.

```
Declaring: MODULE
          /*  CHILL module containing the signals and associated variables, used in the SDL
              diagram                                                                    */

          GRANT
          /*  granting of signals and variables                                          */

          SIGNALS
          /*  signals definition                                                         */

          SYNMODE (OR NEWMODE)
          /*  type definition                                                            */

END Declaring;

Functional_Block: MODULE
          /*  module containing the procedural part of the SDL diagram                   */

          SEIZE

          /*  seizing of all the signals and variables that can be received and sent from (to) this
              functional block                                                           */

          /*  data definition and declaration; such data is global to all the processes belonging to this
              module                                                                     */

          Process name:PROCESS ( );

              /*    local data definition and declaration                                */
              nextstate:=...;
              join:=none;
              DO FOR EVER;
                     state_loop: CASE nextstate OF
                     /*    loop on the variable nextstate indicating the SDL state       */

                            (state_label1): RECEIVE CASE
                                                (signal name1):
                                                . . .
                                                (signal namen):
                     ESAC state_loop;
                     DO WHILE join/=none;
                            CASE join OF
                                   (join - lab1): join: = none;
                                   . . .
                                   (join -labm): join: = none;
                            ESAC;
                     OD;
              OD;
          END process - name;
END Functional - Block;
```

FIGURE D-9.1.1

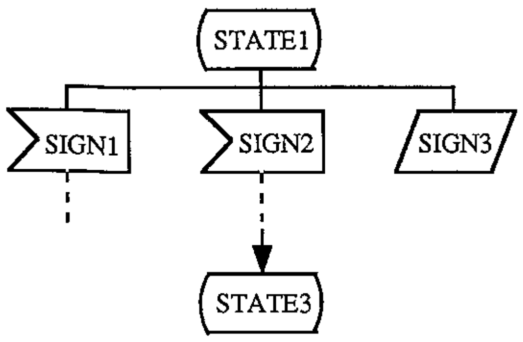
Overall structure of a CHILL program derived from an SDL diagram

The functional block module represents the behaviour (procedural part) of the *SDL* processes.

In this translation schema, each SDL process is represented as an infinite loop; a variable named "nextstate" indicates the state to be examined, and a variable named "join" indicates possible join points determining common sets of statements.

A selection, by means of the case construct of CHILL, is made upon the value of nextstate; each entry of the case identifies an SDL state. In each entry a selection among the possible input signals is made. Each input signal determines the set of actions to be performed (the "transition path").

Each transition path ends with an assignment either to the variable "nextstate", directly determining the next state to be examined, or to the variable "join". A subsequent selection loop on the current value of the variable "join" allows every transition to terminate, in an SDL sense, and at the end assigns a value to the variable "nextstate".



a) SDL

```
STATE1:
   RECEIVE CASE
      (SIGN1): ...
                 ...
      (SIGN2): ....
                 NEXTSTATE := STATE3;
     ELSE GETOUT (LIST1)
   ESAC STATE1;
```

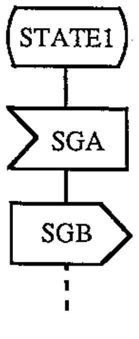
b) CHILL

FIGURE D-9.1.2

Example of mapping STATE/INPUT/SAVE/NEXTSTATE

One of the major problems in relating SDL and CHILL is the different semantics of signal reception; in fact while CHILL doesn't consume (and therefore doesn't destroy) any signal but the expected ones (persistent signals), SDL process consumes all the signals received and destroys the signals not expected for that state. The semantics mismatch has been resolved by introducing the built-in-routine GETOUT, as an alternative (ELSE path) in the CHILL RECEIVE CASE construct, as shown in Figure D-9.1.2. The CHILL built-in-routine GETOUT, which knows (by parameters) the list of input and save signals, destroys the other signals available to the process when it is called.

After executing the GETOUT routine the state selector is set to repeat the loop for that state until a valid input signal is selected (or arrives if not already present).



a) SDL

```
STATE1:
   RECEIVE CASE
     (SGA): pi := get_instance_value();
                send SGB to pi;
                nextstate := ... ;
     ELSE nextstate := state1;
   ESAC STATE1;
```

b) CHILL

FIGURE D-9.1.3

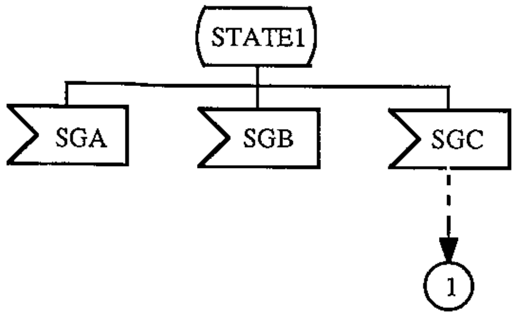
Example of mapping OUTPUT

For example, once the input signal SGA, in Figure D-9.1.3 has been recognized, the appropriate destination process instance for the signal SGB is selected and the signal SGB is sent.

Before sending the signal SGB it may be necessary to fill some information fields which are to be carried by the signal. This can either be done immediately before, or well in advance of sending the signal.

When a join point is met in the diagram (see Figure D-9.1.4), the appropriate value is assigned to the variable "JOIN". As explained in Figure D-9.1.1, a loop on the value of the variable "join" is performed to determine the next state to be examined. A join point can be seen, from the programming language point of view, as a "goto" construct; collecting all the join points together so that they can be examined which allows the entire skeleton program to be written without using gotos, thus making it easier to read.

An SDL decision has a direct translation into the case construct of CHILL, as shown in Figure D-9.1.5.



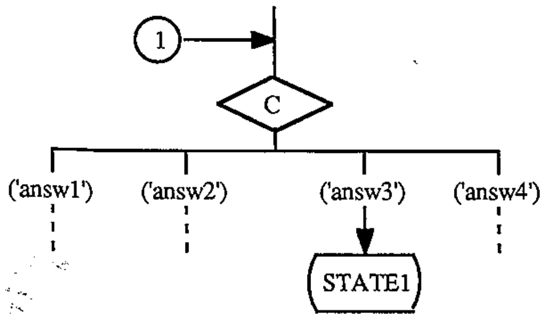
```
STATE1:
   RECEIVE CASE
      (S1 in m): case m.id of
                      (SGA): ... ;
                      (SGB): ... ;
                      (SGC): ... ; JOIN:=1;
                      ELSE nextstate := state1;
                 esac;
   ESAC STATE1;
```

a) SDL

b) CHILL

FIGURE D-9.1.4

Example of mapping join



```
...
1: CASE C OF
     ('answ1'): ... ;
     ('answ2'): ... ;
     ('answ3'): nextstate := state1;
     ('answ4'): ... ;
   ESAC 1;
```

a) SDL

b) CHILL

FIGURE D-9.1.5

Example of mapping DECISION

D.9.2 *Mapping between GR and PR*

Taking into account the restrictions mentioned about macros (§ D.5.1), the GR form can always be mapped to PR and viceversa.

Examples of equivalent specifications in both forms can be found troughout all of this text.

D.10 *Application examples*

D.10.1 *Introduction*

Paragraph D.10 contains some examples of using SDL. The examples are taken, from the telecommunications application area, an attempt was made to take realistic examples and cover as many SDL concepts as possible.

The examples are intended to show the use of SDL, and are not international specifications.

D.10.2 *The service concept*

The following system is an illustration of the service concept. In the system three "system functions" are of special interest for this example. It is a traffic handling function, a maintenance function and an alarm function. The following figure, D-10.2.1, shows how the system functions are composed of subordinate services in five different processes in five blocks.

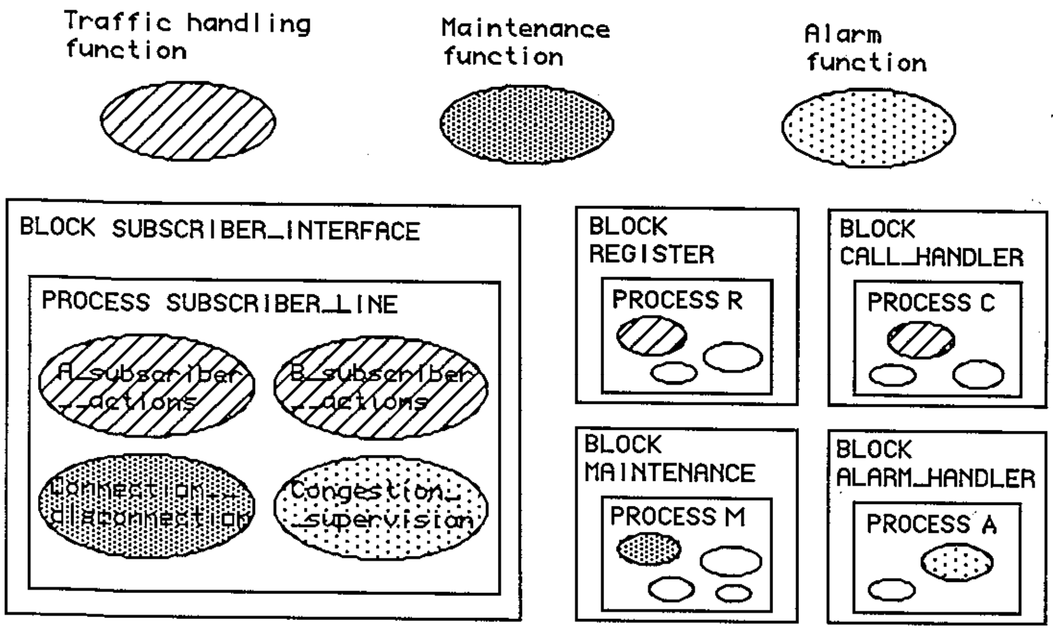


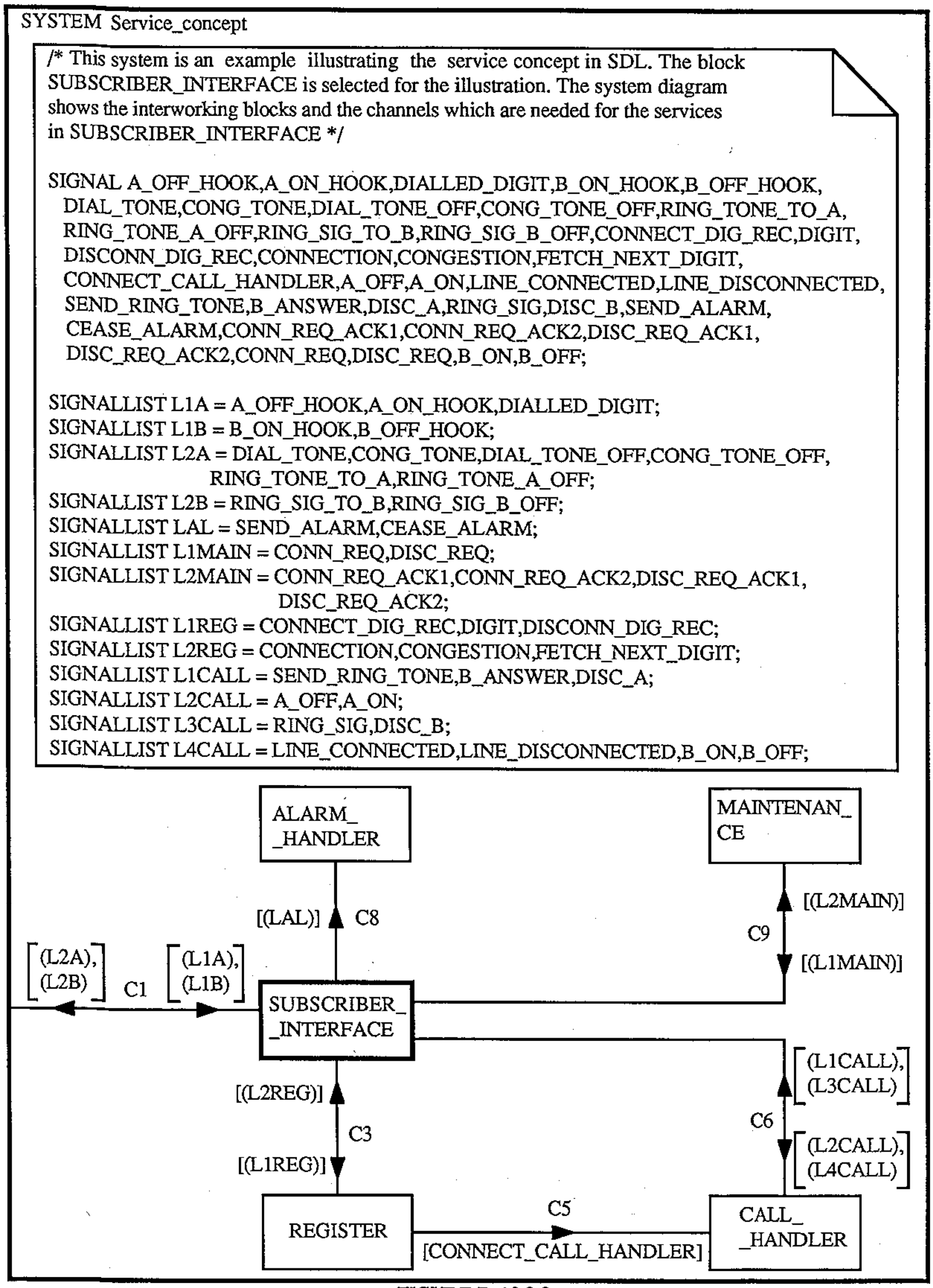
FIGURE D-10.2.1

Function composition

The traffic handling function includes set-up and termination of a telephone call.

This example is not a complete documentation of the functions on system level, just a descnption of the four services in the' SUBSCRIBER_LINE' process.

The following three figures are the system diagram (fig. D-10.2.2), the block diagram for the 'SUBSCRIBER_INTERFACE' (fig. D-10.2.3) and the process diagram for 'SUBSCRIBER_LINE' (fig. D-10.2.4). The necessary channels, signal routes and signals can be seen in these diagrams



```
SYSTEM Service_concept

/* This system is an example illustrating the service concept in SDL. The block
SUBSCRIBER_INTERFACE is selected for the illustration. The system diagram
shows the interworking blocks and the channels which are needed for the services
in SUBSCRIBER_INTERFACE */

SIGNAL A_OFF_HOOK,A_ON_HOOK,DIALLED_DIGIT,B_ON_HOOK,B_OFF_HOOK,
  DIAL_TONE,CONG_TONE,DIAL_TONE_OFF,CONG_TONE_OFF,RING_TONE_TO_A,
  RING_TONE_A_OFF,RING_SIG_TO_B,RING_SIG_B_OFF,CONNECT_DIG_REC,DIGIT,
  DISCONN_DIG_REC,CONNECTION,CONGESTION,FETCH_NEXT_DIGIT,
  CONNECT_CALL_HANDLER,A_OFF,A_ON,LINE_CONNECTED,LINE_DISCONNECTED,
  SEND_RING_TONE,B_ANSWER,DISC_A,RING_SIG,DISC_B,SEND_ALARM,
  CEASE_ALARM,CONN_REQ_ACK1,CONN_REQ_ACK2,DISC_REQ_ACK1,
  DISC_REQ_ACK2,CONN_REQ,DISC_REQ,B_ON,B_OFF;

SIGNALLIST L1A = A_OFF_HOOK,A_ON_HOOK,DIALLED_DIGIT;
SIGNALLIST L1B = B_ON_HOOK,B_OFF_HOOK;
SIGNALLIST L2A = DIAL_TONE,CONG_TONE,DIAL_TONE_OFF,CONG_TONE_OFF,
                 RING_TONE_TO_A,RING_TONE_A_OFF;
SIGNALLIST L2B = RING_SIG_TO_B,RING_SIG_B_OFF;
SIGNALLIST LAL = SEND_ALARM,CEASE_ALARM;
SIGNALLIST L1MAIN = CONN_REQ,DISC_REQ;
SIGNALLIST L2MAIN = CONN_REQ_ACK1,CONN_REQ_ACK2,DISC_REQ_ACK1,
                    DISC_REQ_ACK2;
SIGNALLIST L1REG = CONNECT_DIG_REC,DIGIT,DISCONN_DIG_REC;
SIGNALLIST L2REG = CONNECTION,CONGESTION,FETCH_NEXT_DIGIT;
SIGNALLIST L1CALL = SEND_RING_TONE,B_ANSWER,DISC_A;
SIGNALLIST L2CALL = A_OFF,A_ON;
SIGNALLIST L3CALL = RING_SIG,DISC_B;
SIGNALLIST L4CALL = LINE_CONNECTED,LINE_DISCONNECTED,B_ON,B_OFF;
```

FIGURE D-10.2.2

System diagram

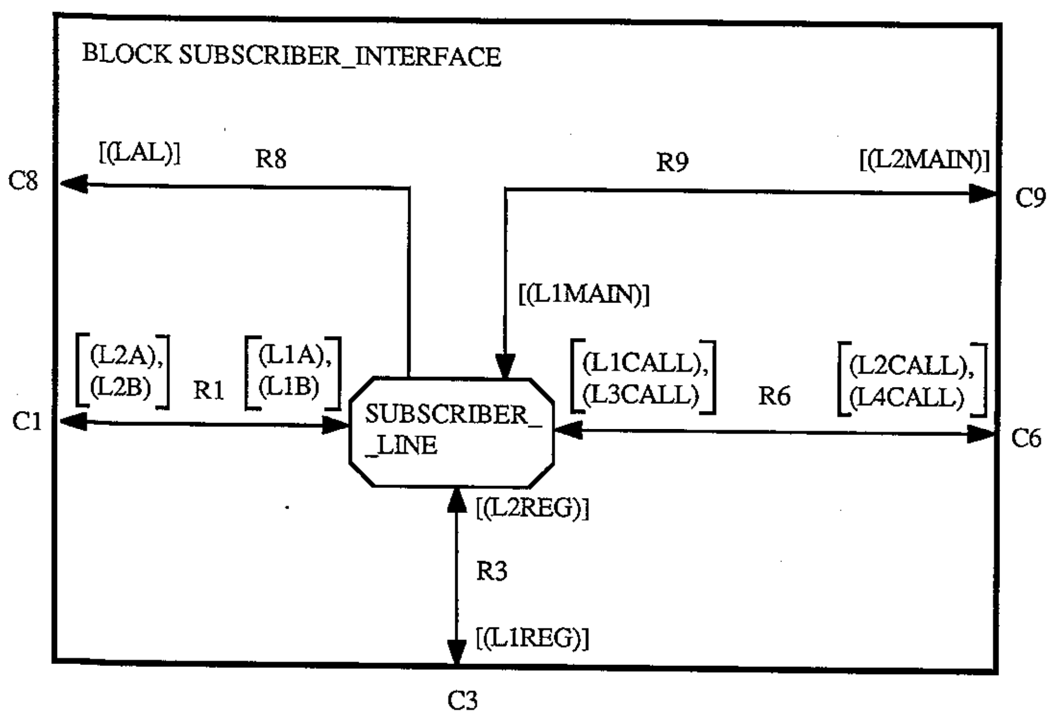
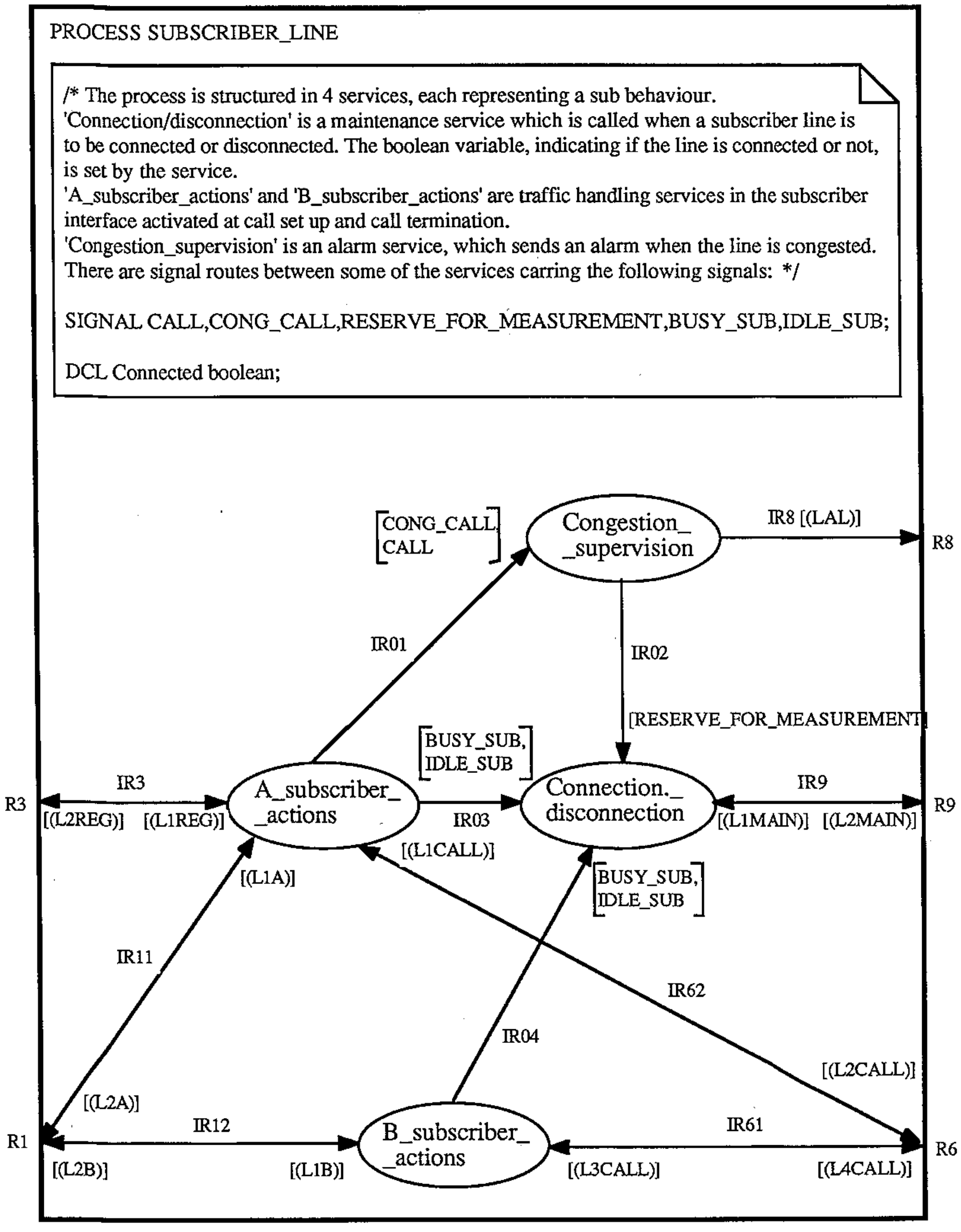


FIGURE D-10.2.3

Block diagram

As can be seen in the process diagram (fig. D-10.2.4) the services interwork with priority signals via the signal routes IRO1,IR02, IR03 and IR04. But the services also interact and affect each other by means of the 'global' variable 'Connected' declared in the process:



PROCESS SUBSCRIBER_LINE

/* The process is structured in 4 services, each representing a sub behaviour.
'Connection/disconnection' is a maintenance service which is called when a subscriber line is to be connected or disconnected. The boolean variable, indicating if the line is connected or not, is set by the service.
'A_subscriber_actions' and 'B_subscriber_actions' are traffic handling services in the subscriber interface activated at call set up and call termination.
'Congestion_supervision' is an alarm service, which sends an alarm when the line is congested.
There are signal routes between some of the services carring the following signals: */

SIGNAL CALL,CONG_CALL,RESERVE_FOR_MEASUREMENT,BUSY_SUB,IDLE_SUB;

DCL Connected boolean;

FIGURE D-10.2.4

Process diagram

To get a better understanding of the services and the interaction between the blocks a number of sequence charts are included in the example.

The first two sequence charts show the normal case for interaction between the blocks during a call. The interaction at register congestion is showed on the third chart. To simplify the charts no delay is assumed between sending and receiving a signal.

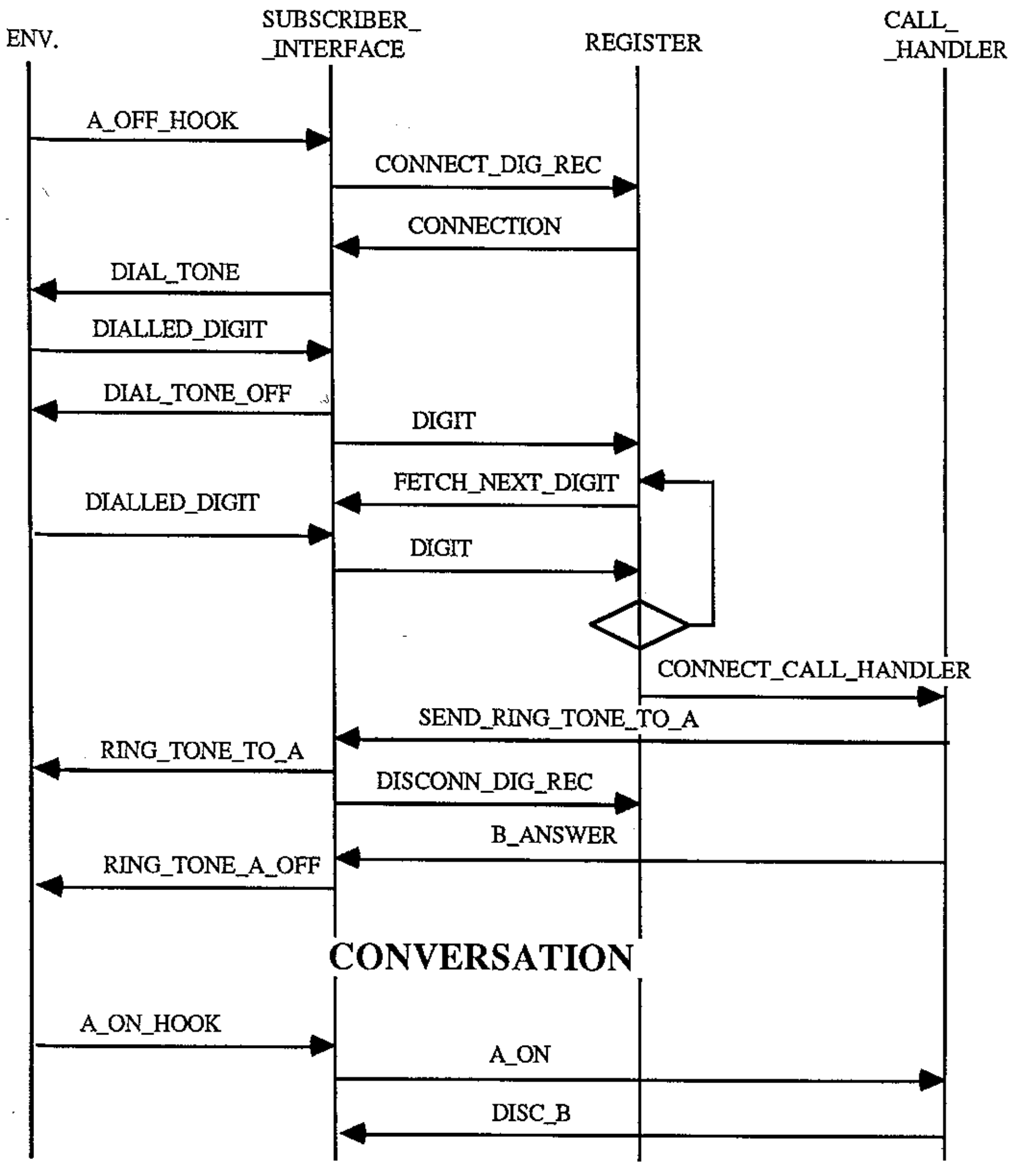
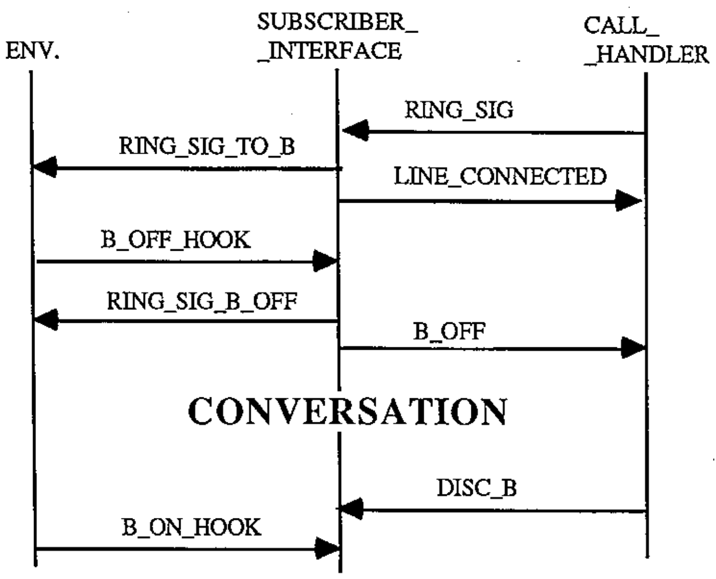


FIGURE D-10.2.5

Sequence chart. Block interaction in the normal case. Necessary signals for the A-subscriber party



Note: Sequence chart. Block interaction in the normal case.
Necessary signals for the B-subscriber party.

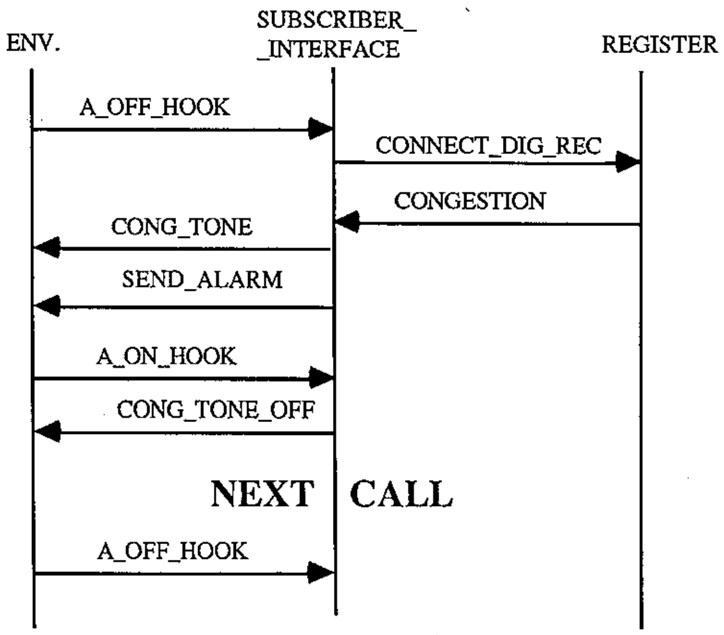
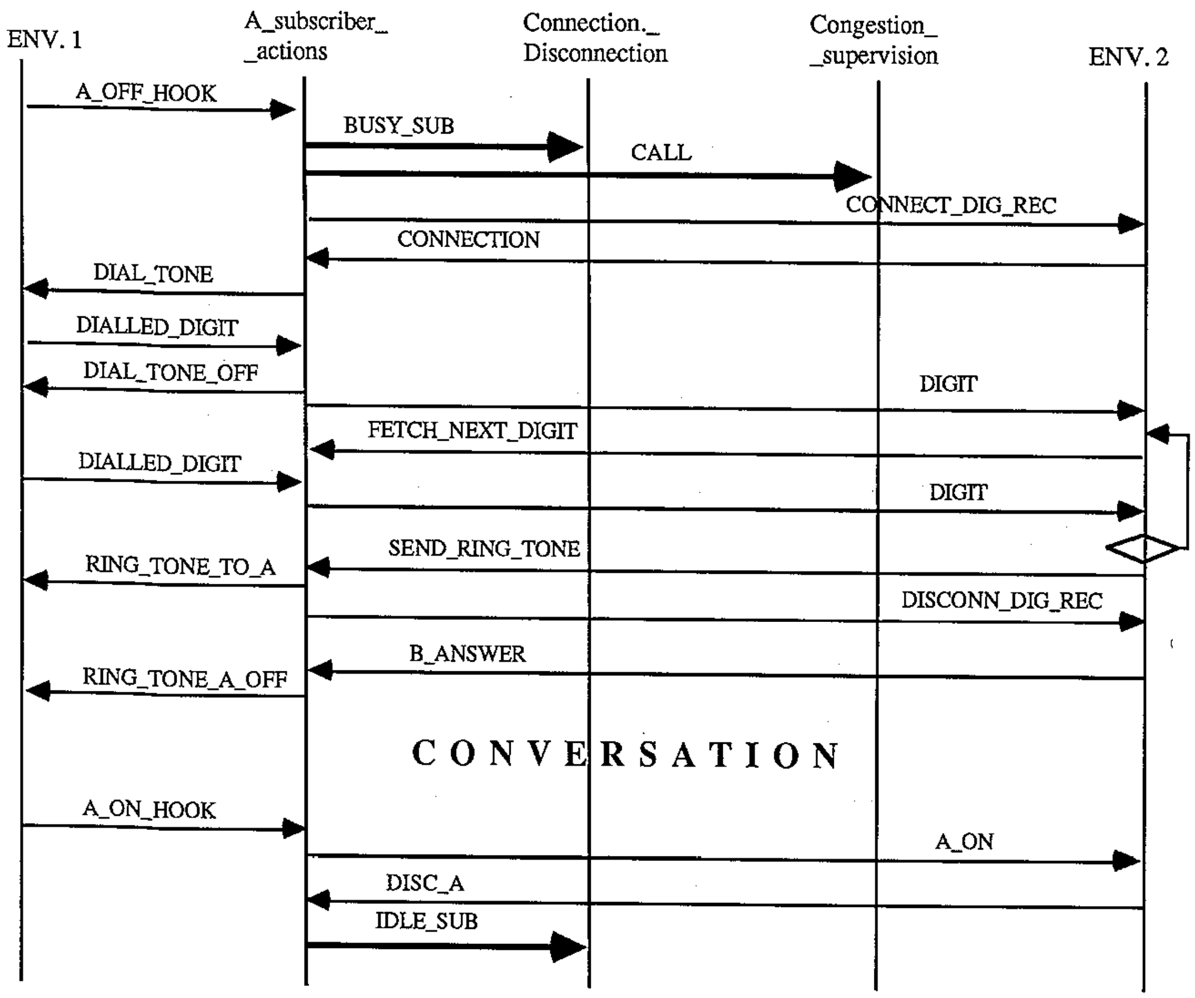


FIGURE D-10.2.6

Sequence chart. Block interaction at register congestion

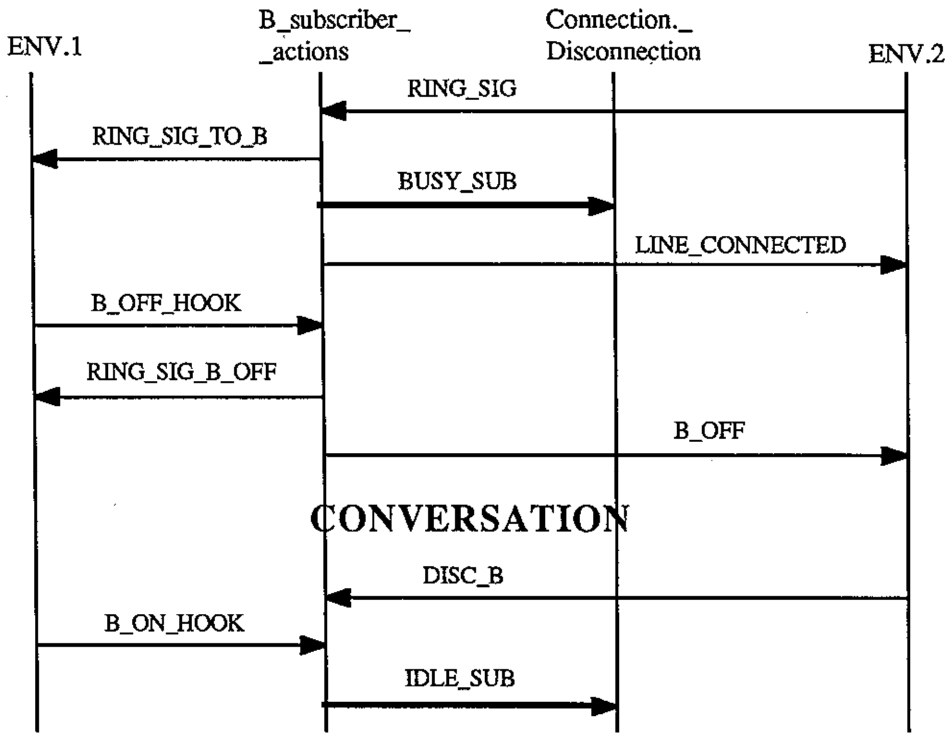
The interaction between the services in the process 'SUBSCRIBER_LINE' is described in the following sequence charts.



Note: The priority signals are indicated with thiker lines.

FIGURE D-10.2.7

Sequence chart. Service interaction in the normal case. Necessary signals for the A-subscriber party



Note: The priority signals are indicated with thiker lines.

FIGURE D-10.2.8

Sequence chart. Service interaction in the normal case. Necessary signals for the B-subscriber party

The behaviour of each service in the process 'SUBSCRIBER_LINE' is described in the four service diagrams (fig. D-10.2.9÷15).

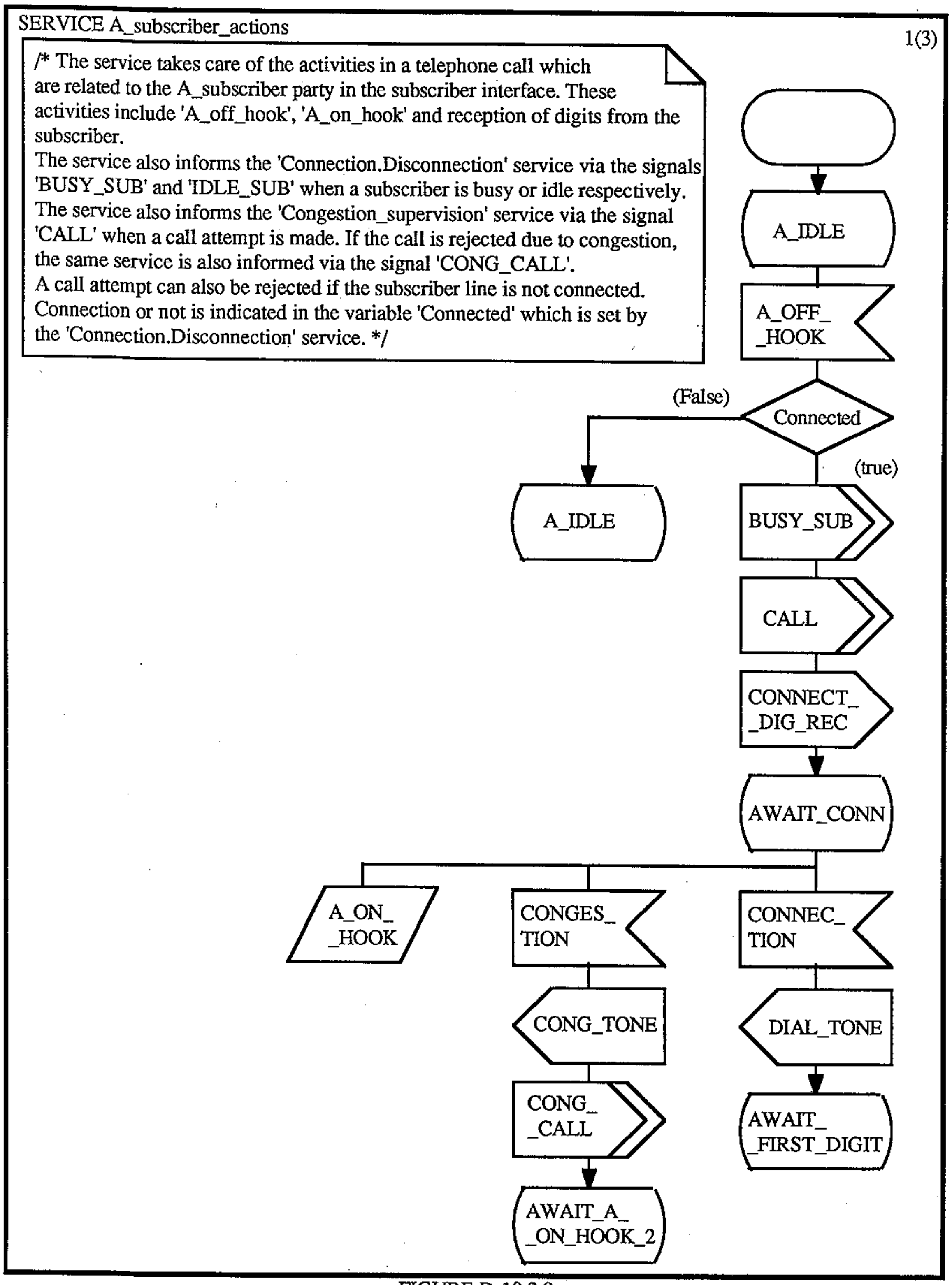

SERVICE A_subscriber_actions
1(3)
/* The service takes care of the activities in a telephone call which are related to the A_subscriber party in the subscriber interface. These activities include 'A_off_hook', 'A_on_hook' and reception of digits from the subscriber.
The service also informs the 'Connection.Disconnection' service via the signals 'BUSY_SUB' and 'IDLE_SUB' when a subscriber is busy or idle respectively.
The service also informs the 'Congestion_supervision' service via the signal 'CALL' when a call attempt is made. If the call is rejected due to congestion, the same service is also informed via the signal 'CONG_CALL'.
A call attempt can also be rejected if the subscriber line is not connected.
Connection or not is indicated in the variable 'Connected' which is set by the 'Connection.Disconnection' service. */
A_IDLE
A_OFF_ _HOOK
(False)
Connected
(true)
A_IDLE
BUSY_SUB
CALL
CONNECT_ _DIG_REC
AWAIT_CONN
A_ON_ _HOOK
CONGES_ TION
CONNEC_ TION
CONG_TONE
DIAL_TONE
CONG_ _CALL
AWAIT_ _FIRST_DIGIT
AWAIT_A_ _ON_HOOK_2

FIGURE D-10.2.9

Service diagram

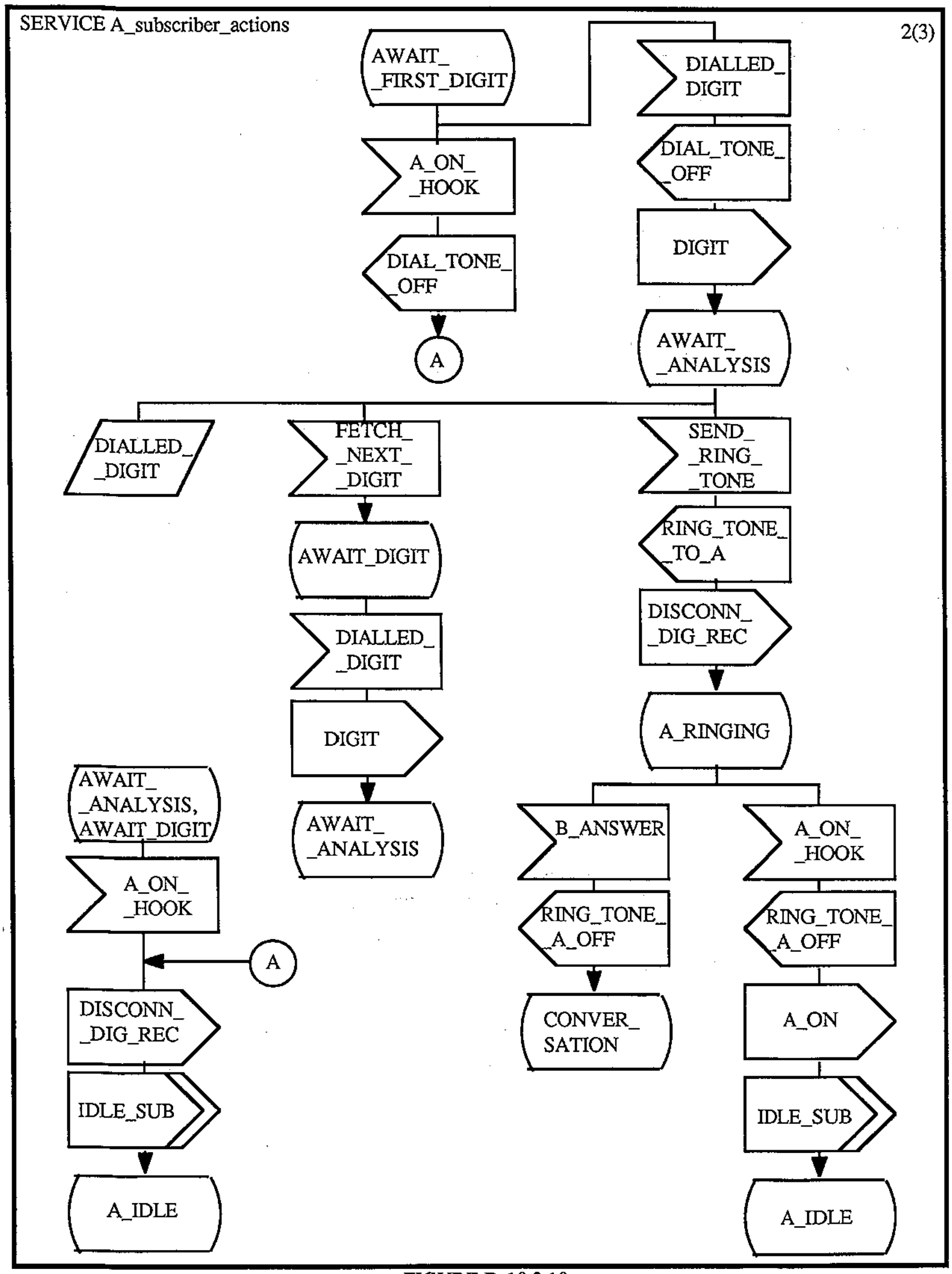


FIGURE D-10.2.10

Service diagram

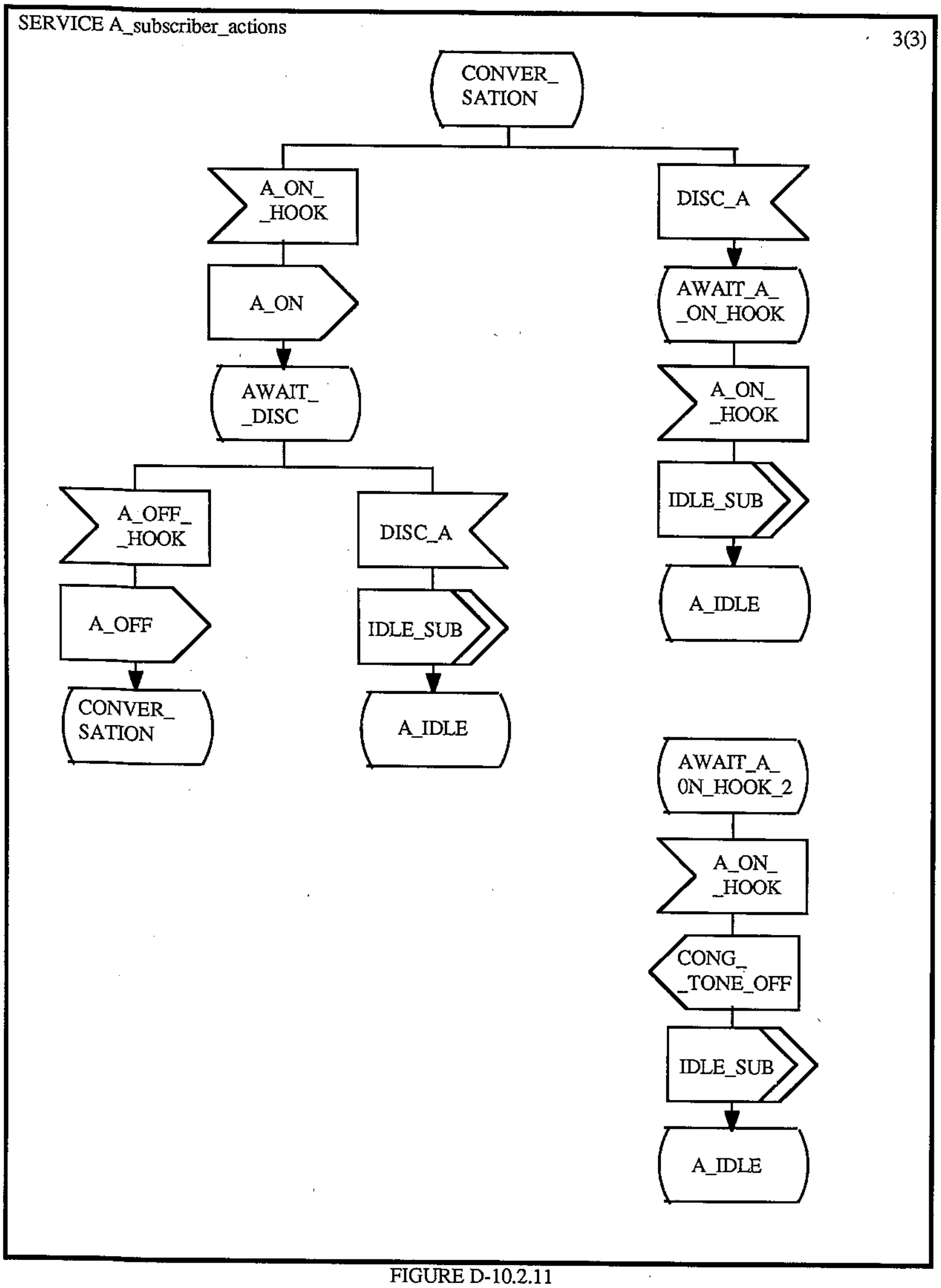


FIGURE D-10.2.11

Service diagram

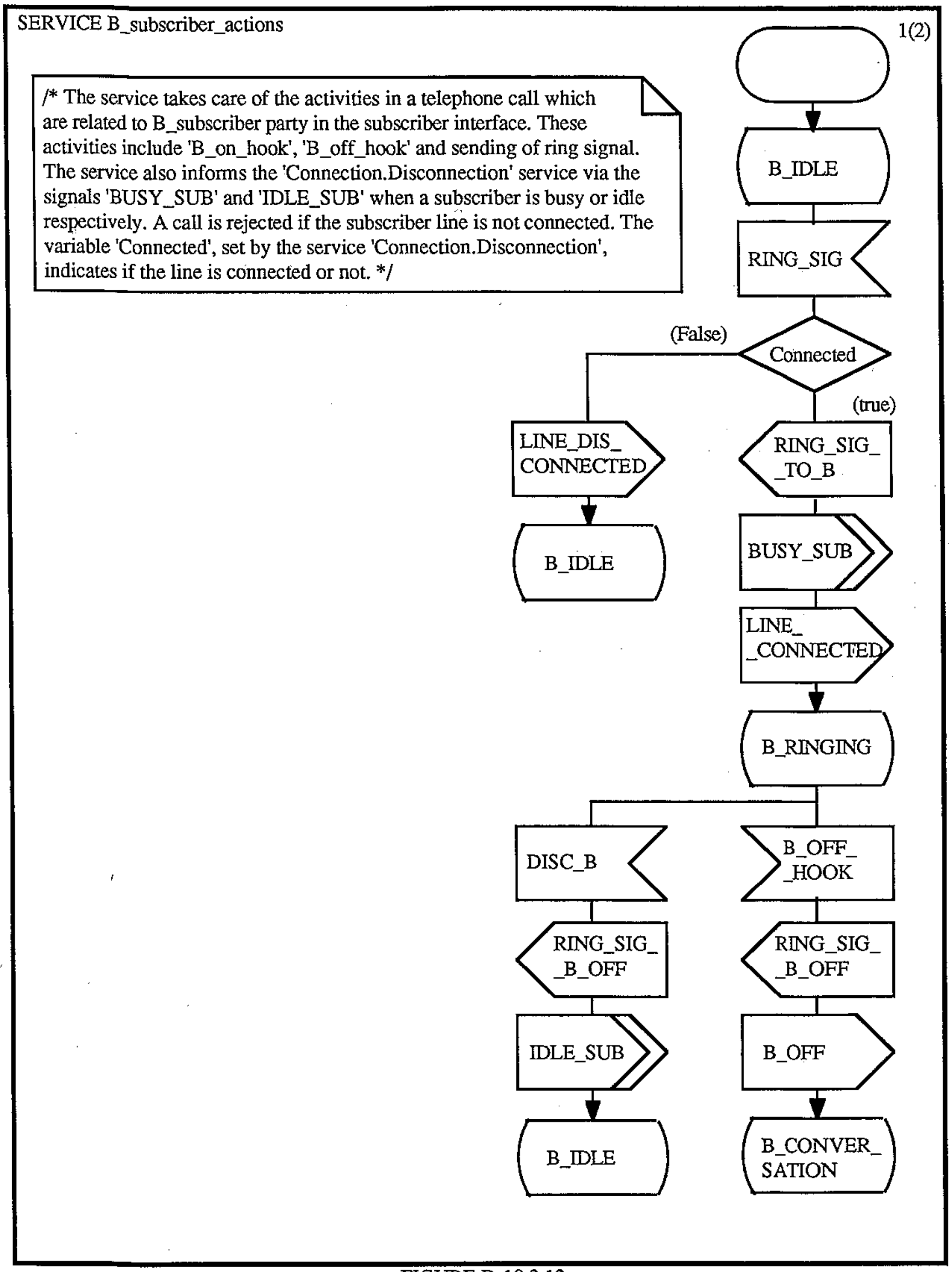

SERVICE B_subscriber_actions
1(2)
/* The service takes care of the activities in a telephone call which are related to B_subscriber party in the subscriber interface. These activities include 'B_on_hook', 'B_off_hook' and sending of ring signal. The service also informs the 'Connection.Disconnection' service via the signals 'BUSY_SUB' and 'IDLE_SUB' when a subscriber is busy or idle respectively. A call is rejected if the subscriber line is not connected. The variable 'Connected', set by the service 'Connection.Disconnection', indicates if the line is connected or not. */
B_IDLE
RING_SIG
Connected
(False)
(true)
LINE_DIS_ CONNECTED
B_IDLE
RING_SIG_ _TO_B
BUSY_SUB
LINE_ _CONNECTED
B_RINGING
DISC_B
B_OFF_ _HOOK
RING_SIG_ _B_OFF
RING_SIG_ _B_OFF
IDLE_SUB
B_OFF
B_IDLE
B_CONVER_ SATION

FIGURE D-10.2.12

Service diagram

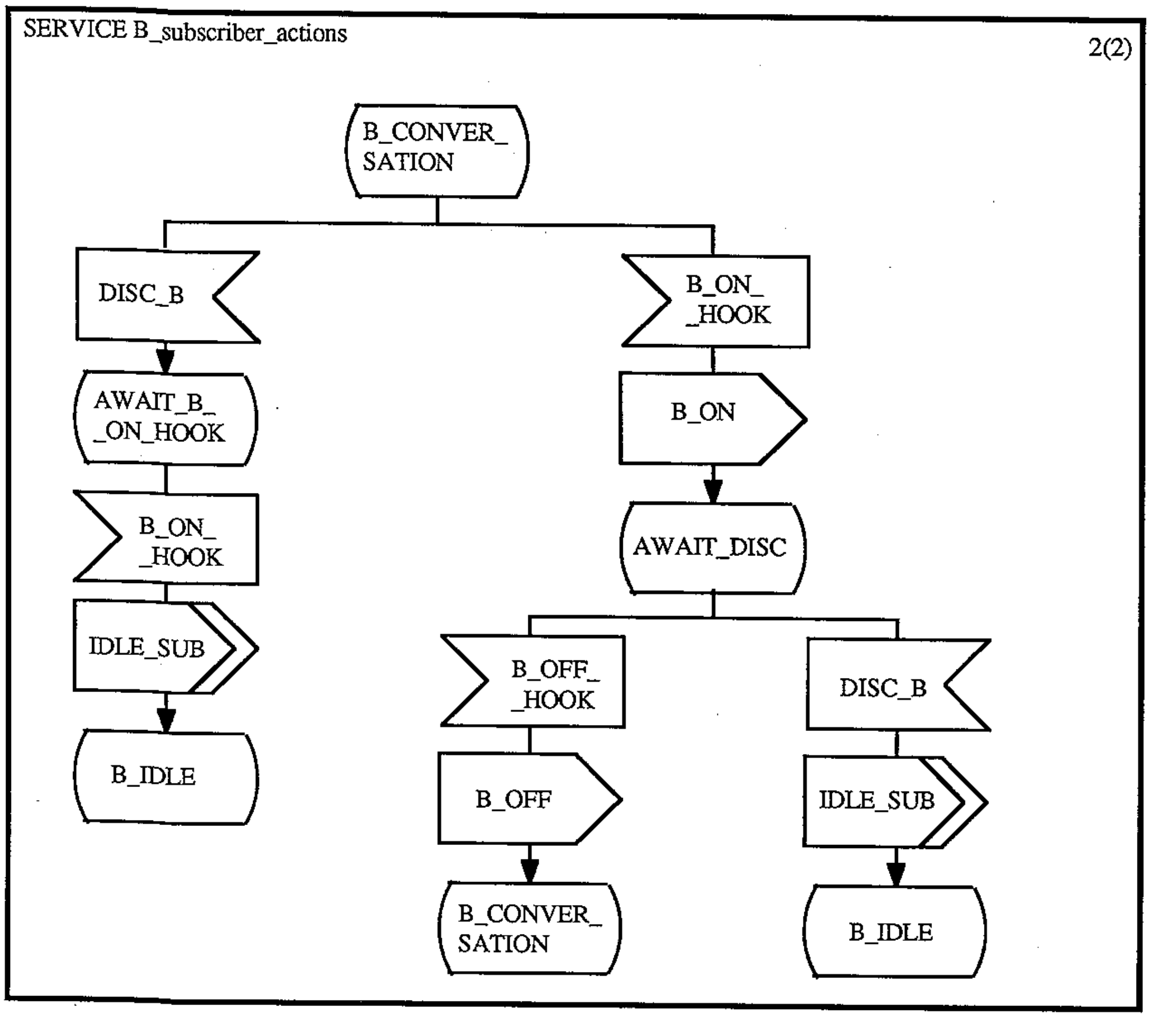
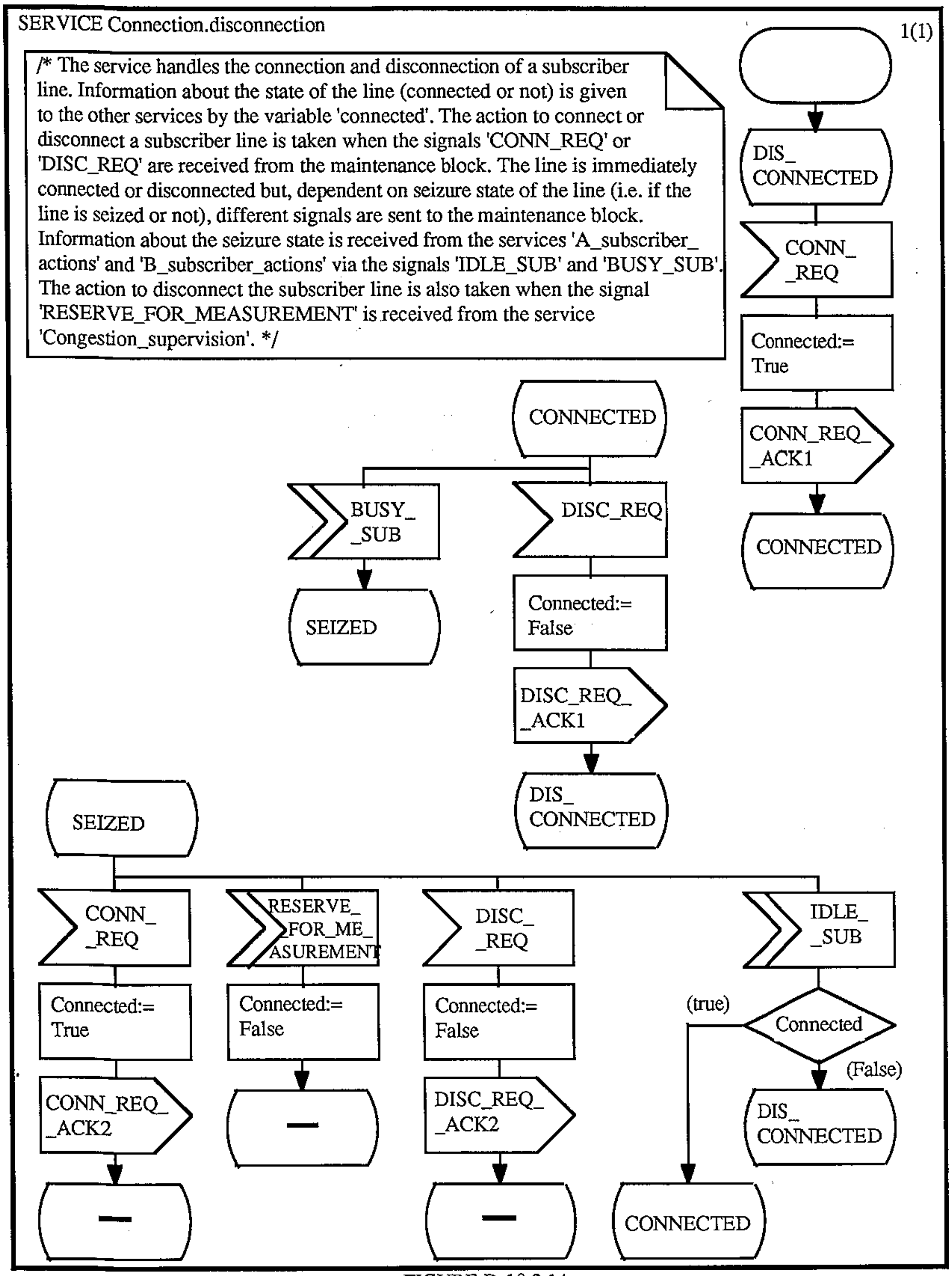


FIGURE D-10.2.13

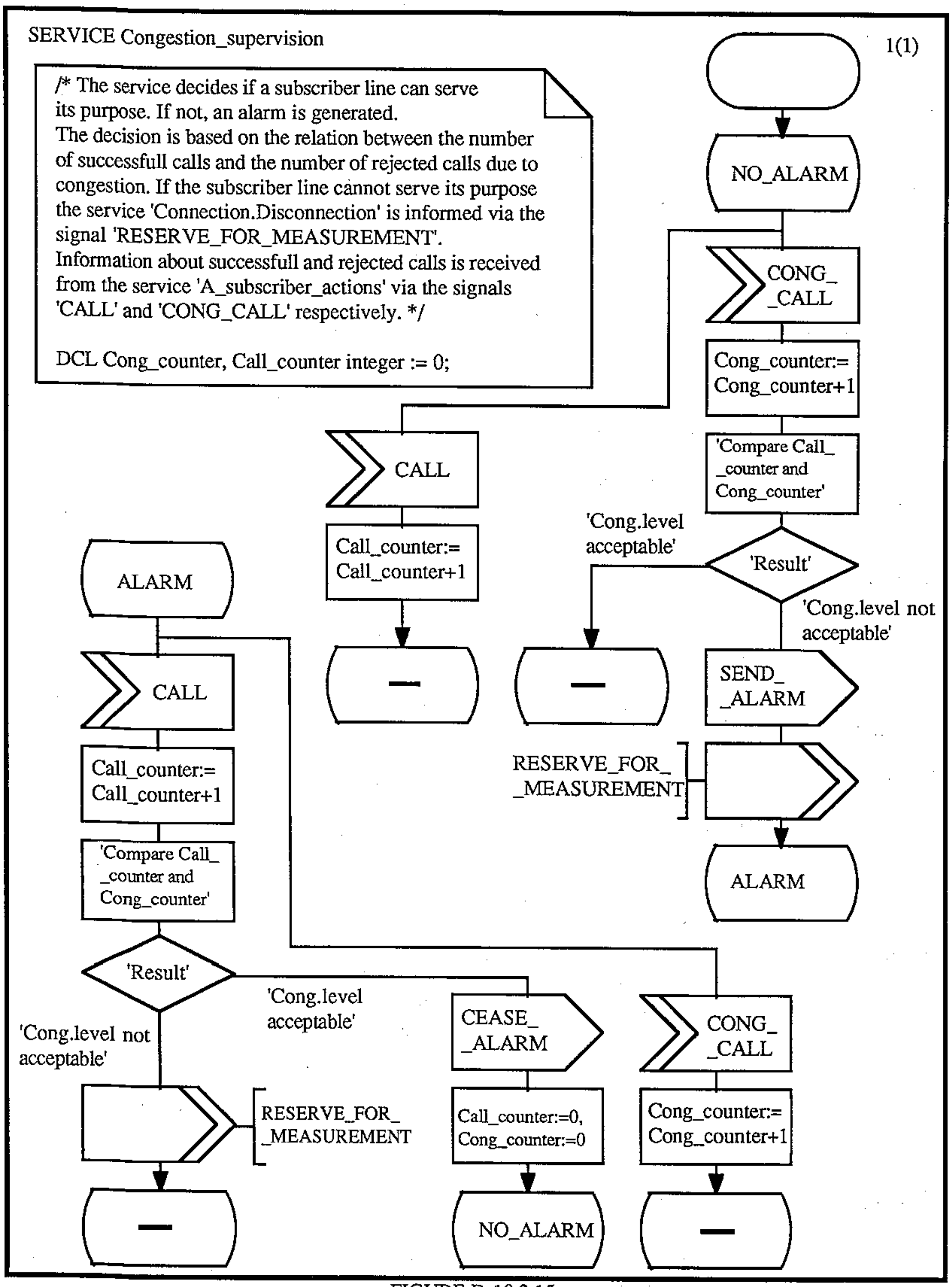
Service diagram


SERVICE Connection.disconnection

/* The service handles the connection and disconnection of a subscriber line. Information about the state of the line (connected or not) is given to the other services by the variable 'connected'. The action to connect or disconnect a subscriber line is taken when the signals 'CONN_REQ' or 'DISC_REQ' are received from the maintenance block. The line is immediately connected or disconnected but, dependent on seizure state of the line (i.e. if the line is seized or not), different signals are sent to the maintenance block. Information about the seizure state is received from the services 'A_subscriber_actions' and 'B_subscriber_actions' via the signals 'IDLE_SUB' and 'BUSY_SUB'. The action to disconnect the subscriber line is also taken when the signal 'RESERVE_FOR_MEASUREMENT' is received from the service 'Congestion_supervision'. */

FIGURE D-10.2.14

Service diagram



SERVICE Congestion_supervision
1(1)
/* The service decides if a subscriber line can serve its purpose. If not, an alarm is generated.
The decision is based on the relation between the number of successfull calls and the number of rejected calls due to congestion. If the subscriber line cannot serve its purpose the service 'Connection.Disconnection' is informed via the signal 'RESERVE_FOR_MEASUREMENT'.
Information about successfull and rejected calls is received from the service 'A_subscriber_actions' via the signals 'CALL' and 'CONG_CALL' respectively. */
DCL Cong_counter, Call_counter integer := 0;
NO_ALARM
CONG_ _CALL
Cong_counter:= Cong_counter+1
'Compare Call_ _counter and Cong_counter'
'Result'
'Cong.level acceptable'
'Cong.level not acceptable'
SEND_ _ALARM
RESERVE_FOR_ _MEASUREMENT
ALARM
CALL
Call_counter:= Call_counter+1
—
ALARM
CALL
Call_counter:= Call_counter+1
'Compare Call_ _counter and Cong_counter'
'Result'
'Cong.level not acceptable'
'Cong.level acceptable'
RESERVE_FOR_ _MEASUREMENT
—
CEASE_ _ALARM
Call_counter:=0, Cong_counter:=0
NO_ALARM
CONG_ _CALL
Cong_counter:= Cong_counter+1
—

FIGURE D-10.2.15
Service diagram

D.11 *Tools for SDL*

D.11.1 *Introduction*

This paragraph outlines a set of tools which may support SDL. These tools may support the generation of documents, SDL diagram (SDL/GR) or statements (SDL/PR), and/or the validation of SDL specifications.

An exhaustive list of possible tools is not listed in these guidelines. The tools required are dependant on the methodology chosen by the user.

In principle SDL can be used without any tools. However, the inherent complexity of modem systems is such that the SDL specifications are often complex. Therefore, automatic tools are needed to help manage the specification, design, and documentation of many systems. For instance, the complexity and cost of manually drawing, and possibly updating, the graphic documentation of a switching exchange would be significantly reduced by using suitable aids.

Due to the above considerations SDL has been designed so that tools can be effectively used to support its use.

D.11.2 *Classes of tools*

SDL tools may be classified by the activities carried out during the production of SDL documentation, for instance:

- Tools for input: According to the syntactic forms we have input aids for the SDL graphic, textual phrase, and pictorial.
- Tools for syntactic checking: These include syntax analyzers for each of the two syntaxes.
- Tools for document generation: Once SDL documents are stored within a machine, tools can access and reproduce them, possibly using several peripherals. These may use a syntactic form different from that used to input the document. In addition tools may be able to generate new types of documents derived from those originally entered.
- Tools for system modelling and analysis: The SDL documents representing a system can be used for abstracting a model of the system. Checks can be performed on that model. They can search for deadlocks, make a comparison between various models of the same system (e.g. between a specification and a description), or run a simulation of the system behaviour etc.
- Tools supporting code generation: Very detailed SDL specifications may be used to help in the programming of software. Tools may be developed in such a way that a guided, semi-automatic, code sequence may be performed.

A specific, but useful, class of tools is represented by:

- SDL training tools: These may either stand alone or be integrated with other tools. Integration allows users of other functions to be given aid when required.

As SDL is used in many of the various phases of a systems life cycle, one can easily see a use of all types of tools in an integrated project environment.

D.11.3 *Document input*

No special requirements are needed for entering the textual phrase form of SDL, as the SDL/PR is equivalent, from the input viewpoint, to any character string input. Therefore it may benefit by the same tools (string editors). The other syntax however assumes graphic handling capability.

It is obvious that whilst support for SDL/PR input can be beneficial, SDL/GR input support is essential, if we plan to use these syntaxes as input.

A graphic editor is always required for functions such as the connection of two symbols, displacement of a set of symbols to another area of the page or to other pages, and concatenate deletion (deletion of a symbol implies the deletion of the connection to that symbol). As for the SDL/PR tools, the SDL/GR input tools should be modelled on the SDL semantics/syntax. Therefore it should highlight invalid connections, and prompt the user to fill up all of the uncompleted parts etc.

The tools are faced with several problems which derive from the physical constraints of graphics devices, such as "resolution". It is almost impossible to have a sufficient number of characters, which are readable, while also allowing the display of a reasonable number of symbols on the screen.

Solutions such as a zooming window, or scrolling, have to be considered, but they are not completely satisfactory. High resolution may not be considered a "must" if diagrams are copied by the user but it is very desirable if diagrams are produced directly by the user. For the same reason (requirement of an overview and of a certain amount of details) high resolution is desirable in the display of diagrams.

Tools for aiding the SDL/PR input can be helpful: they may prompt the user with expected SDL/PR keywords.

They can immediately format the SDL/PR according to the keywords received, insert delimiters automatically, and present to the user SDL/PR oriented function keys, provide a clear layout etc.

The implementation of such tools may be based on existing character string editors which can be extended to include the features mentioned above.

D.11.4 *Document verification*

Once documents are stored in a machine the next step is to verify them. They should first be verified individually, and then related diagrams should be combined and verified until the total system is verified.

If the input has been made through an SDL oriented tool a good amount of checking on each individual document could have already been done.

Errors deriving from "not possible" operations (e.g. inputs or saves following anything but a state) should all be detected and corrected during the input phase. The detection of some errors is however only possible at the completion of the input phase, both on a single document and, of course, in case of inconsistencies between documents.

Several SDL rules can be automatically checked. For instance the requirement that all outputs should have a corresponding input.

In the case of a multi-level specification, to a certain extent, consistency between levels can be checked.

The formal SDL model may be used to derive a collection of verification procedures.

D.11.5 *Document reproduction*

SDL documents stored in a *machine* must be able to be retrieved, displayed, and reproduced. Tools for all these activities are required. It may be useful to be able to retrieve only part, or subsets, of documents. The retrieval can be SDL oriented, e.g. "find all the processes sending" a given signal, or "in which states" is a certain action performed, and so on. Tools for displaying information are of particular interest when the information is to be displayed using the graphic syntax. The same observations as made for the input of documents in SDL/GR syntaxes apply. The reproduction of documents is dependent on the type of document to be reproduced, on how this document has been stored and on the characteristics of the output peripheral. It may also be dependent on how it was entered. Users may wish for an output in a different syntax from the one used to enter the document.

The reproduction of documents is affected by the output peripheral constraints. For example a diagram can be too large to fit in a given space of paper, and therefore has to be chopped into pieces. Connectors have to be added and cross references should be inserted. It may be desirable to distinguish between an "addition" made by the tool and original input characteristics. Tools allowing flexibility in output format are desirable: these features include different sizes of symbol, different output formats, vertical or horizontal display etc.

A document should always be able to be reproduced in exactly the same way as it was entered.

D.11.6 *Document generation*

Starting from the SDL documents entered by the users and stored in the machine, several other documents can be generated automatically, among these:

- the signal lists, organized per process, per block, or per system;
- the state overview diagrams, showing the process graph as a set of states connected by arcs representing the transitions;
- the cross reference table, organized per process, per block, or per system;
- the block tree diagram, showing the structure of the blocks and levels;
- the system behaviour, as a response to sequences of environment actions;
- indexes: Once generated documents will have to be reproduced and the same considerations presented above are valid.

SDL documents entered in SDL/GR can automatically be translated to the equivalent SDLIPR and viceversa.

The following considerations apply:

- the SDL/GR contains visual information which cannot be translated into the SDL/PR (it does not exist in SDL/PR). For example, the coordinates of the symbols are meaningless in SDL/PR;
- connectors linking flow lines on different pages can be eliminated.

The inverse translation, from SDL/PR to SDL/GR, is however much more complex and is unlikely to completely satisfy all the potential readers.

Due to the two dimensional representation of the SDL/GR some labels inserted to cope with the sequential structure of the SDL/PR, can be deleted as a connection line is sufficient.

Usually the translation generates a model of the SDL/GR diagrams. This model contains all the information necessary for a tool to format and reproduce the diagram on a graphic device.

Note that two different tools translating some SDL/PR into SDL/GR may obtain two SDL/GR specifications with different layouts. The SDL/GR specifications so obtained are both correct provided they preserve the semantics expressed in the originating specification.

D.11.7 *System modelling and analysis*

SDL documents, independent of whether they specify or describe a system, are basically a model of that system.

This model, which is primarily intended to transfer information from one person to another, can also be interpreted by tools, checked for consistency, for completeness (this aspect may not be satisfied in cases of specification aiming at specifying only some parts of a system), and for correctness and compliance with the SDL rules (as described in the paragraph on document verification).

In addition, tools may be developed to use the model to simulate the systems functional behaviour. The simulator can interact with the environment and conclusions on the adherence of the model to the users expectations can be drawn.

If additional information is added to indicate the time consumed in executing each action, and to dimension the resources available (queues, instances etc.) the simulation can also study the capacity of the system.

Tools may be developed to create a model of the environment, starting from the system model, to create meaningful sequences to check the real system. Path analysis may detect deadlocks in the model.

The system model can also be used as on-line documentation. If appropriate links exist between the actual system and the documentation storage, a tool may be developed to trace system real time events on the model.

To achieve this, correlation should be provided between the physical events, as seen by the system, and the logical events dealt with by the SDL documentation. If the documentation is organized into several levels of abstraction the user can choose the level to be traced. This can be very useful as it allows users with different education and training to inspect the system activities.

Tools interpreting the SDL model can also be used to single out differences in behaviour of different models of the same system. It may also be used to compare different system descriptions (systems produced by different companies), or to compare the system specification with the description. In this way it is possible to see if a system description complies with the original specification.

D.11.8 *Code generation*

The provisions of a formally defined syntax and of a formal mathematical definition of SDL, make it possible to realize tools able to map the semantics of SDL specifications to the semantics of programming languages. Such tools are possibly unable to provide complete implementation programs, but they may be very useful in providing at least a frame-work for the actual program.

Paragraph D.9.1 of this U.G. outlines an example of how the mapping between SDL and CHILL may be achieved.

D.11.9 *Training*

A complete training course on SDL has been developed; it covers all the aspects of the language providing at the same time examples and a few suggestions on the use of SDL.

The SDL Course is obtainable from the International Telecommunication Union, General _Secretariat - Sales Section, Place des Nations, CH-1211 Geneve 20 (Switzerland)

ITU-T RECOMMENDATIONS SERIES

Series A	Organization of the work of the ITU-T
Series B	Means of expression: definitions, symbols, classification
Series C	General telecommunication statistics
Series D	General tariff principles
Series E	Overall network operation, telephone service, service operation and human factors
Series F	Non-telephone telecommunication services
Series G	Transmission systems and media, digital systems and networks
Series H	Audiovisual and multimedia systems
Series I	Integrated services digital network
Series J	Transmission of television, sound programme and other multimedia signals
Series K	Protection against interference
Series L	Construction, installation and protection of cables and other elements of outside plant
Series M	TMN and network maintenance: international transmission systems, telephone circuits, telegraphy, facsimile and leased circuits
Series N	Maintenance: international sound programme and television transmission circuits
Series O	Specifications of measuring equipment
Series P	Telephone transmission quality, telephone installations, local line networks
Series Q	Switching and signalling
Series R	Telegraph transmission
Series S	Telegraph services terminal equipment
Series T	Terminals for telematic services
Series U	Telegraph switching
Series V	Data communication over the telephone network
Series X	Data networks and open system communications
Series Y	Global information infrastructure and Internet protocol aspects
Series Z	**Languages and general software aspects for telecommunication systems**

Printed in Switzerland

Geneva, 2008