INTERNATIONAL TELECOMMUNICATION UNION

ITU-T Z.100 Appendices I and II

(03/93)

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

PROGRAMMING LANGUAGES

SDL METHODOLOGY GUIDELINES

SDL BIBLIOGRAPHY

ITU-T Recommendation Z.100 – Appendices I and II

(Previously "CCITT Recommendation")

FOREWORD

The ITU Telecommunication Standardization Sector (ITU-T) is a permanent organ of the International Telecommunication Union. The ITU-T is responsible for studying technical, operating and tariff questions and issuing Recommendations on them with a view to standardizing telecommunications on a worldwide basis.

The World Telecommunication Standardization Conference (WTSC), which meets every four years, established the topics for study by the ITU-T Study Groups which, in their turn, produce Recommendations on these topics.

Appendices I and II to ITU-T Recommendation Z.100 were revised by the ITU-T Study Group X (1988-1993) and were approved by the WTSC (Helsinki, March 1-12, 1993).

NOTES

1 As a consequence of a reform process within the International Telecommunication Union (ITU), the CCITT ceased to exist as of 28 February 1993. In its place, the ITU Telecommunication Standardization Sector (ITU-T) was created as of 1 March 1993. Similarly, in this reform process, the CCIR and the IFRB have been replaced by the Radiocommunication Sector.

In order not to delay publication of this Recommendation, no change has been made in the text to references containing the acronyms "CCITT, CCIR or IFRB" or their associated entities such as Plenary Assembly, Secretariat, etc. Future editions of this Recommendation will contain the proper terminology related to the new ITU structure.

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

© ITU 1994

All rights reserved. No part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from the ITU.

CONTENTS

			Page
Appendix I – SDL methodology guidelines			1
I.1	Introduction		1
	I.1.1	Objective	1
	I.1.2	Scope	1
	I.1.3	Conventions	1
	I.1.4	Framework	1
	I.1.5	The implications of using SDL	3
	I.1.6	The application area of SDL	4
I.2	Modelling of applications		7
	I.2.1	OSI services and protocols	7
	I.2.2	ISDN according to I.130 and Q.65	13
I.3	Stepwise production of an SDL specification		18
	I.3.1	Introduction	19
	I.3.2	The steps	19
	I.3.3	Example: The lift	29
I.4	Object-Orientation and SDL		35
	I.4.1	Object-oriented analysis	35
	I.4.2	Applying object-oriented SDL concepts	41
I.5	Stepwise production of a complete ADT specification		46
	I.5.1	Completeness of an ADT specification	46
	I.5.2	The basic constructor function method	48
	I.5.3	Four additional steps	52
	I.5.4	Equations for constructors	53
	I.5.5	Limitations	54
I.6	Using message sequence charts		54
	I.6.1	Introduction	54
	I.6.2	System specification using MSC-composition mechanisms	55
	I.6.3	A method for system development based on MSC, SDL and functional decomposition	65
I.7	Derivation of implementations from SDL specifications		69
	I.7.1	Introduction	71
	I.7.2	Differences between real systems and SDL systems	73
	I.7.3	Implementation specifications	78
	I.7.4	Trade-off between hardware and software	84
	I.7.5	Software architecture design	86
	I.7.6	Hardware architecture design	105
	I.7.7	Stepwise guidelines to implementation design	105
I.8	Formal approaches to validation, verification and testing		107
	I.8.1	Introduction	107
	I.8.2	Validation and verification	108
	I.8.3	Conformance testing	109

Page

I.9 Auxiliary documents 110
I.9.1 Communication and interface specification 110
I.9.2 Tree diagram 115
I.9.3 State-overview diagram 119
I.9.4 Signal-state matrix 120
I.10 Documentation 120
I.10.1 Introduction 120
I.10.2 Language support for documentation 122
I.10.3 Mapping of specifications to documents 124
I.10.4 Documentation issues 124

Bibliography 126

Appendix II – SDL Bibliography 129

Appendix I to Recommendation Z.100

SDL METHODOLOGY GUIDELINES

(Helsinki, 1993)

I.1 Introduction

I.1.1 Objective

The objective of this appendix is to provide guidelines for the effective use of SDL within some overall methodology.

It is recognized that there are many ways of using SDL, depending on the user's preference, on the target application, on the in-house rules of the organization, etc. The guidelines published in this appendix are therefore not normative, it is left for the discretion of the users of SDL if and to what extent they should be followed. The reason for publishing them is the belief that they are useful, and also important for the promotion of SDL, and for the unified usage and tool support of the language.

I.1.2 Scope

This appendix provides guidelines for the following topics:

- the use of SDL for different application areas;
- stepwise production of an SDL specification;
- using message sequence charts;
- derivation of implementations from an SDL specification;
- formal approaches to validation, verification and testing;
- auxiliary diagrams;
- documentation aspects.

The guidelines do not form a single, coherent and complete methodology. They are not intended to be exhaustive, nor specific, nor on a detailed level, nor to unduly restrict the freedom of the SDL users. They are intended to be selected and incorporated by the SDL users in their overall methodologies, and tailored for their application systems and specific needs.

I.1.3 Conventions

Formal specifications should not be confused with natural language text or figures. This is achieved in this appendix normally either by treating formal specifications as Examples or by including them in separate subclauses. Examples are numbered if appropriate, e.g. when they are referenced. For textual formal specifications the "`courier`" font is used.

An example may cover only part of a complete SDL specification. Missing parts are indicated e.g. by a dotted line or an open-ended flow line.

I.1.4 Framework

According to standard dictionary, a methodology is a system of methods and principles used in a particular discipline. A method is a systematic way of doing something, or alternatively the techniques or arrangement of work for a particular field or subject. In the context of this appendix, the disciplines are development of telecommunication systems, protocols etc, called *application systems* (or just *systems* for brevity, where the qualifying term can be derived from the context).

Specification and design are *activities*. There may be several alternative methods for the same activity. Thus, the evaluation of alternative approaches and the selection of one of them is an important issue when developing a methodology. This must be based among other things on the characteristics of the application system.

An activity is characterized by its inputs and outputs. The conducting of an activity is governed by some rules, which are derived from the policies of the underlying enterprise. An activity is usually performed by people, belonging to the enterprise and making use of tools (see Figure I.1-1). An activity can be regarded as a finite state machine or system, and can be described using similar techniques (e.g. by using SDL).

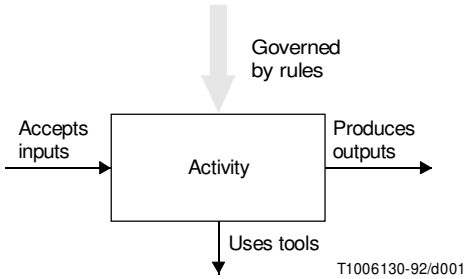


FIGURE I.1-1/Z.100

Specification of an activity

An activity is performed generally on the representation of (part of) the application system. Therefore, the decomposition of an activity into sub-activities must be based on an information model and the architecture of the application system.

Each information item must be expressed in an appropriate language (natural, semi-formal or formal). The real world is described ultimately in some natural language, which is very good to express aim and intention, and bad to express details precisely and unambiguously. A formal language has the opposite properties. A methodology should use different categories of languages, so that they complement each others and so that their advantages are exploited.

Based on the above discussion, a methodology (in the context of this document) generally contains the following components:

- enterprise model;
- activity model;
- system architecture;
- information model;
- languages;
- working procedures;
- rules for product handling and documentation;
- tool usage.

The *enterprise model* describes the overall objectives of a application system in terms of actions, goals and policies. It specifies the activities that take place within the organisation using the application system, the roles that people play in the organisation, and the interactions between the organisation, the application system, and the environment in which application system and organisation are placed.

The *activity model* identifies all relevant activities for a application system, and the relation between these. Each activity is described as indicated above. Note that the activity model is usually called life cycle model. This term should be avoided, since it implies, incorrectly, a linear ordering of activities. An activity model is generally two dimensional, allowing concurrent execution of activities.

The *application architecture* provides a decomposition in principle of the application system into self-contained parts, and rules for the interaction between these parts.

The *information model* identifies all relevant information for the application system to be solicited, produced, maintained and delivered, as well as the relation between different pieces of information.

The *languages* provide means to express the information model with appropriate precision, allowing a number of checks to be made and the use of computer based tools.

The *working procedures* enforce some commonly accepted rules to be followed by all project members, in order to promote effective work and achieve high quality result.

The *rules for product handling and documentation* complement the working procedures, and ensure the maintainability of the application documentation.

The *tools* increase the productivity of the project members and the quality of products by performing well defined and labour intensive tasks.

NOTE – A methodology covers different levels of abstraction. The activity model covers all levels of abstraction, but some other methodology components may not be applicable on all levels of abstraction. Working procedures are applicable, e.g. only on lower levels.

I.1.5 The implications of using SDL

SDL is a formal specification languages. For brevity, the term *formal* will normally be omitted and assumed in the following, when talking about specification languages.

It is probably widely accepted that the key for the success of a system is a thorough system specification and design. This requires, on the other hand, a suitable specification language, satisfying the following needs (the result of the system specification and design activity is called here *specification*, for brevity):

- a well defined *set of concepts*; and
- *unambiguous*, *clear*, *precise* and *concise* specifications; and
- a basis for *analysing* specifications for *completeness* and *correctness*; and
- a basis for determining *conformance* of implementations to specifications; and
- a basis for determining *consistency* of specifications relative to each other; and
- use of computer based *tools* to create, maintain, analyse and simulate specifications.

For a system there may be specifications on different levels of abstraction. A specification is a basis for deriving *implementations*, and it should abstract from implementation details in order

- to give overview of a complex system;
- to postpone implementation decisions; and
- not to exclude valid implementations.

In contrast to a program, a formal specification (that is a specification written in a specification language) is not intended to be run on a computer. In addition to serving as a basis for deriving implementations, a formal specification can be used for precise and unambiguous communication between people, particularly for ordering and tendering.

The use of a specification language makes it possible to analyse and simulate alternative system solutions, which in practice is impossible when using a programming language due to the cost and the time delay. A specification language offers a well defined set of concepts for the user of the language, improving his capability to produce a solution to a problem and to reason about the solution.

I.1.5.1 Understanding a formal specification

As mentioned above, a specification language offers a well defined set of concepts. These concepts form some kind of mathematical model, and therefore may seem strange to some people who are not familiar with the underlying mathematical theory. Therefore, an introductory discussion of how to apply the formal model of a specification language to an application is given below.

The application domain of a system is understood ultimately in terms of the concepts of a natural language. We acquire these concepts through a long process of learning and real life experience. The description of an application in a natural language is descriptive by nature, phenomena are described as they are perceived by an observer. The natural language description of the system makes use of concepts that are derived directly from the application and implementation of the system.

When a system is specified using a specification language, the formal specification neither makes use of application nor of implementation concepts, it rather defines a *model* that represents the significant properties (mainly behaviour) of the system. In order to understand this model, it must be mapped to the intuitive understanding of the application in terms of natural language concepts, (see Figure I.1-2). This mapping can be done in different ways, one way is to choose names for the concepts introduced in the formal specification that give good association to the application concepts, another way is to comment the formal specification. This is much the same issue as the understanding of a program, or an algorithm that solves a problem taken from real life.

A model should have a good *analytical power*, as indicated in I.1.5, and a good *expressive power* to ease the mapping to the application. Unfortunately, the analytical power and the expressive power are generally in conflict: the more expressive a model is, the more difficult it is to analyse it. When designing a specification language, evidently a trade-off must be made between these two properties. In addition, it should be stressed that a model always represents a simplified view of reality, and has consequently always limitations.

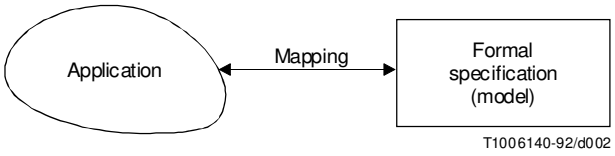


FIGURE I.1-2/Z.100

How to understand a formal specification

I.1.5.2 Relation to implementation

Normally a distinction is made between *declarative* and *constructive* languages. SDL is a constructive language, which means that an SDL specification defines a *model* that represents significant properties of a system (as mentioned above). An important question is what these significant properties should be. This is left open in SDL, *it is up to the user to decide what these properties should be*.

If a decision is made to represent only the behaviour of the system (as seen at its boundary), then people normally talk about a *specification*, and the given structure of the model is only an aid to structure the specification into manageable pieces. If a decision is made to represent also the internal structure of the system, then people normally talk about a *description*. This is the reason why SDL does not make any distinction between specification and description.

Note that the internal structure of the system is normally represented in SDL by *blocks*. *Processes* and internal signalling within blocks need not represent part of the internal structure of the system, and thus need not be considered as requirements on the implementation, as some people erroneously assume. Conformance of implementations to specifications is normally treated by standard bodies by *explicit conformance statement*. This is evidently necessary also when a constructive language is used.

I.1.6 The application area of SDL

Today SDL is mainly known within the telecommunication field, but has a broader application area. The application area of SDL can be characterized as follows:

- *type of system*: real time, interactive, distributed;
- *type of information:* behaviour and structure;
- *level of abstraction:* overview to detail.

SDL has been developed for use in telecommunication systems including data communication, but can actually be used in all real time and interactive systems. It has been designed for the specification of the behaviour of such a system, i.e. the cooperation between the system and its environment. It is also intended for the description of the internal structure of a system, so that the system can be developed and understood one part at a time. This feature is essential for distributed systems.

SDL covers different levels of abstraction, from a broad overview down to detailed design. It has not been intended to be an implementation language. More or less automatic translation of SDL specification to a programming language is, however, possible.

Figure I.1-3 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 activities, whose precise names may vary from organization to organization. Each of the directed lines (flow lines) represents a set of documents passing from one activity to another; SDL specifications 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.

When particularly considering 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.

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

I.1.6.1 On the nature of switching software

Switching software can be seen as having certain characteristics of its own. Most often the software runs as an *embedded system*, on a hardware platform dedicated to a specific task (e.g. a switch). Naturally, the hardware imposes limits on what the software can do. Compared to many other areas of software development, the *functional requirements* are well defined in the sense that they are precise, rather simple, unambiguous, and quite often even formally defined. Very seldom the requirements change during software development. A large portion of switching software can be considered *reactive* – the system is sent a message and it is supposed to give a response. Switching software is usually *real-time* but one should emphasize that the performance requirements are more often *statistical* than absolute. As a rule, switching software is *parallel* and *distributed*.

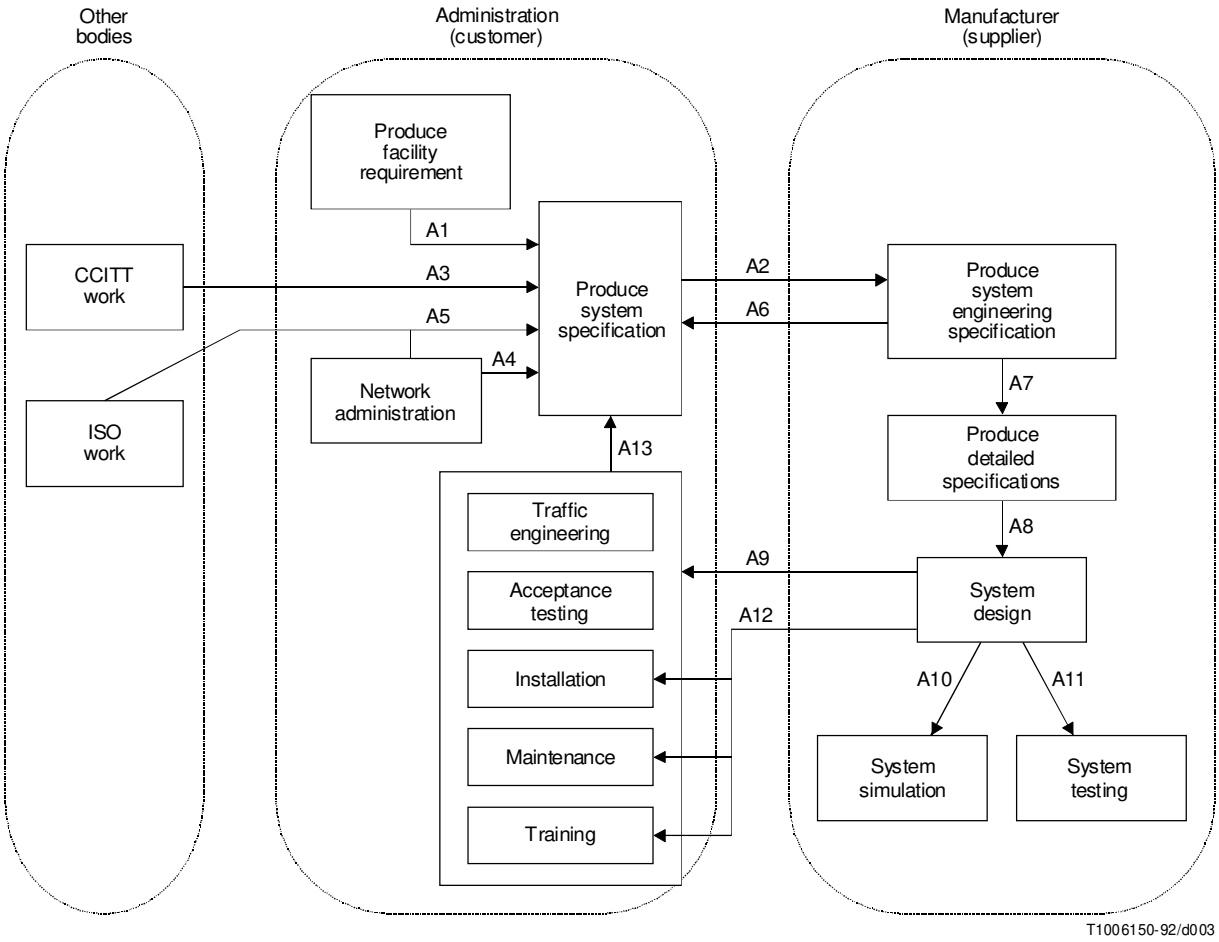
I.1.6.2 The role of SDL in switching software

Many designers involved in the development of switching software perceive SDL to a lesser extent as a *specification language* (expressing what the system should do) than as a *description language* (expressing how the system actually does it). For the designer, the software development activity starts from the receipt of the requirements specification, possibly written in SDL, from which the application system is then derived in an implementation-oriented manner.

The application of SDL as a description language for switching systems is a disciplined activity which integrates several skill-demanding, multi-faceted aspects of state-of-the-art software engineering. SDL provides a sufficient conceptual framework to support the different levels of abstraction from requirements specification down to implementation.

The problem of testing the SDL specification is non-trivial. Firstly, switching software systems are huge, and the developer most often has only an overall understanding of the environment of his/her code. It is not unimaginable that the modules determined to communicate with each other are being developed at the same time. Therefore a *test environment* has to be built for each component. *Integration testing* and *system testing* are possible only later. Secondly, embedded systems do not always provide software development tools of the quality of ordinary computer systems and the developer has to satisfy him/herself with the monitoring of message exchanges and simple debugging. Sometimes an alternative is to simulate the behaviour of the actual SDL specification on a more developed hardware platform in order to ensure the functional correctness of the design. Another choice is to generate test cases or test inputs directly from the SDL specification. Thirdly, it is advisable to check the *conformance* of the SDL design against the requirements specification. However, this is done most often in an unautomated, informal way.

After the delivery, the system design will have to be *maintained* which is one of the reasons for making the SDL specification readable and well documented. Each maintenance action should trigger a new cycle of testing, preferably with the same test data as earlier. Several parts of switching systems are *customized* which makes it necessary to have an adequate *version control* system.



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 Recommendations and guidelines
A4 Contributions to the system specification, showing the network administration and operational requirements
A5 Other relevant standards
A6 Description of an implementation proposal
A7 A project specification
A8 A detailed design specification
A9 A complete system description
A10 Appropiate system and environment description for system simulation
A11 Appropiate system and environment description for system testing
A12 Installation and operation manuals
A13 Contributions to the system specification from specialized activities within an Administration

NOTES

1 Interation is possible at all levels.

2 In some circumstances, SDL specifications that is here shown as being internal to one organization, e.g. A1, A7 and A8, could be supplied to another organization.

FIGURE I.1-3/Z.100

General scenario for the use of SDL

I.2 Modelling of applications

I.2.1 OSI services and protocols

I.2.1.1 Introduction

The OSI concepts used in I.2.1 are defined in [1] and [2]. In order to make the subclause more self-contained, an explanation is given of the key concepts,

A *layer service* is offered by a *service provider* for a given layer. The service provider is an abstract machine offering a communication facility to *users* in the next higher layer. The service is accessed by the users at *service access points* by means of *service primitives*, (see Figure I.2-1). A service primitive can be used for connection management (connection, disconnection, resetting, etc), or be a data object (normal data or expedited data). There are only four kinds of service primitives:

– request (from user to provider);

– indication (from provider to user);

– response (from user to provider);

– confirmation (from provider to user).

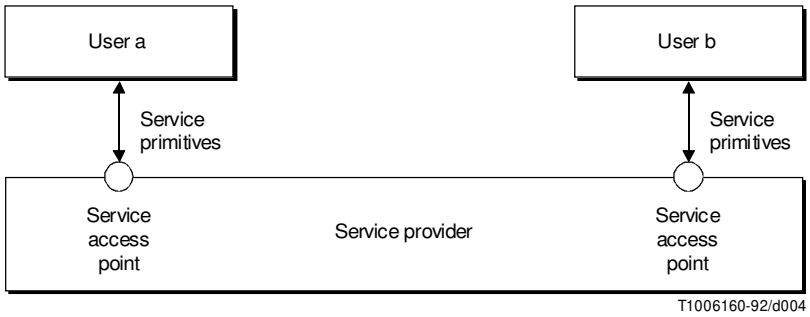


FIGURE I.2-1/Z.100

A layer service provider

A *service specification* is a way to characterize the behaviour of the service provider, both locally: stating legal sequences of service primitives transferred at one service access point, and end-to-end: stating correct relationship between service primitives transferred at different service access points. A service specification does not deal with the internal structure of the service provider; any internal structure given when specifying the service is just an abstract model for describing the externally observable behaviour of the service provider.

Except for the highest layer, the users of a layer service are *protocol entities* of the next higher layer, which cooperate in order to enhance the features of the layer service, thus providing a service of the next higher layer. The cooperation is carried out in accordance with a predefined set of behaviour rules and message formats, which constitute a *protocol*. According to this view, protocol entities of (N)-layer and the (N – 1)-service provider provide together a refinement of the (N)-service provider (see Figure I.2-2).

The refinement of the (N)-service provider shown in Figure I.2-2 may of course be much more complicated. For example, there may be (N)-relay protocol entities which are not connected to any protocol entity of the (N + 1)-layer. Such cases are not considered here for the sake of brevity.

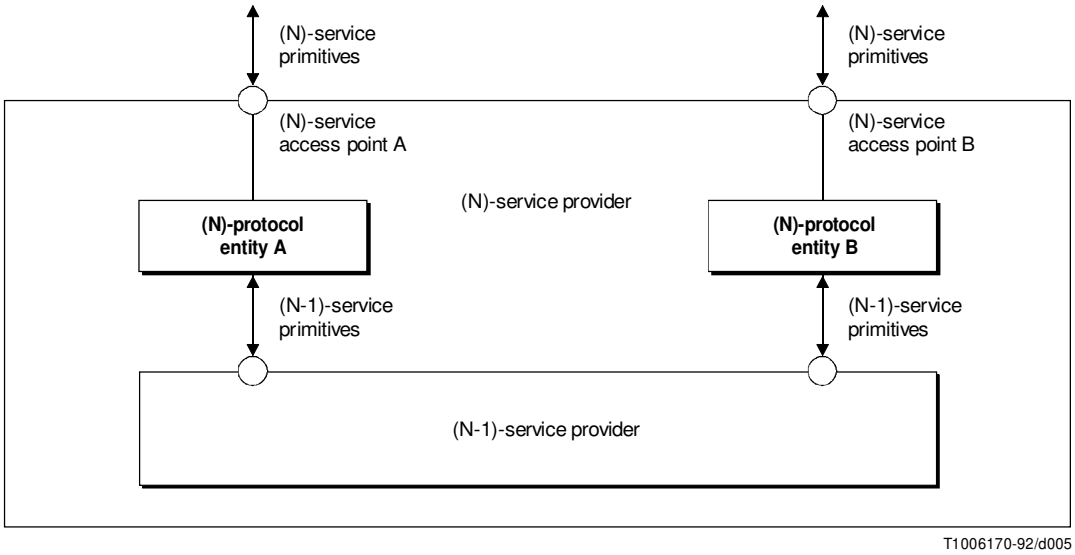


FIGURE I.2-2/Z.100

Refinement of the (N)-service provider

Protocol entities communicate by exchange of *protocol data units*. These are transferred as parameters of service primitives of the underlying layer. The sending protocol entity encodes protocol data units into service primitives, the receiving protocol entity decodes protocol data units from the received service primitives. A protocol is based on the properties of the underlying service provider. The underlying service provider may for example lose, corrupt or misorder messages, in which case the protocol should contain mechanisms of error detection and correction, resynchronization, retransmission etc, in order to provide a reliable and usually more powerful service to the next higher layer.

The OSI architectural concepts can be modelled in SDL in a number of alternative ways, mainly depending on what aspect should be emphasized. First, a basic approach is described, then other approaches are outlined as variants of the basic approach.

In the examples, the graphical syntax of SDL is used as far as possible. Note, however, that for practical reasons, some information that is required by the syntax rules may be omitted, or represented by a series of dots (...), which is, of course, not part of the syntax.

I.2.1.2 Basic approach

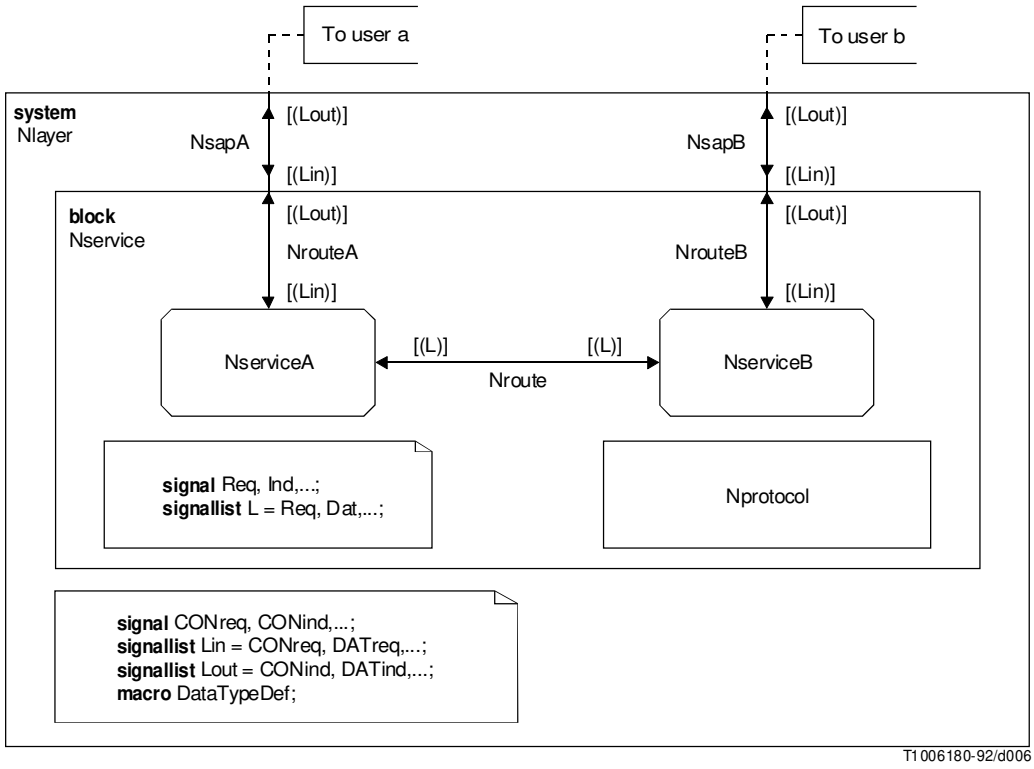
Service specification

A service specification for layer N can be modelled in a straightforward manner as a block *Nservice* containing two processes *NserviceA* and *NserviceB*, (see Example I.2-1).

In Example I.2-1, the users of this service are in the environment of the system and can be considered as processes, capable of communicating with the system on terms of this system.

A service access point is represented by a channel (*NsapA*, or *NsapB*), conveying signals, which represent service primitives. A signal may carry values of the sorts given in the signal specification. The sort specifications (other than for predefined sorts) are contained in the remote *macro* specification *DataTypeDef*, which is omitted here as irrelevant for the purpose of the present discussion.

Of course, in the most general case there may be more than two processes involved in the service specification. We will here consider only two processes, one for each service access point, for the sake of brevity. The following discussion applies, however, also to the case where there are more processes for a service access point.



EXAMPLE I.2-1

(N)-service specification in SDL

In Example I.2-1, some examples of signal specifications are shown. As suggested by some signal names, a connection oriented service is assumed. In the case of a connectionless service, a great deal simplifications can be made. However, this case will not be treated further, for the sake of brevity.

Both local and end-to-end aspects of a service specification are dealt with here. Local behaviour is expressed independently by the processes *NserviceA* and *NserviceB*. These processes communicate with each other by signals (*Req*, *Ind*,...), which are internal to the block and are conveyed on the signal route *Nroute*. End-to-end behaviour is expressed by the mapping (performed by each process) between service primitives and internal signals on *Nroute*. The processes *NserviceA* and *NserviceB* are mirror images of each other. The reason for having two of them, instead of only one, is to faithfully model a possible collision situation in the service provider.

Non-deterministic behaviour is an inherent feature of the service provider, because it may refuse connection attempts and may disrupt established connections on its own initiative.

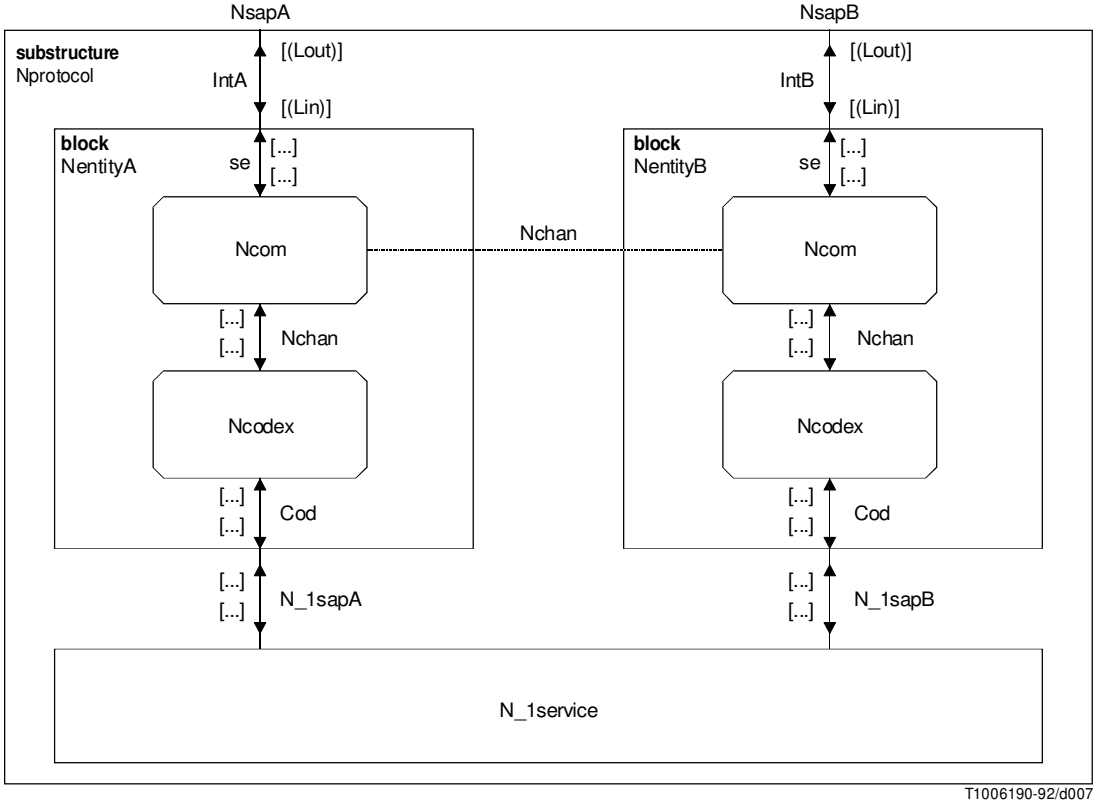
Please note that the specification of the block *Nservice* contains only reference to the processes *NserviceA* and *NserviceB*; these processes are specified by *remote specifications*, placed outside the block specification, and not shown here, because they would force the reader to pay attention to features necessarily specific of a given service.

Protocol specification

The protocol specification for layer N is modelled by the substructure *Nprotocol* of the block *Nservice* (see Example I.2-1).

In the *block diagram* of Example I.2-1, a block substructure reference (a block symbol containing the name *Nprotocol* of the block substructure) has been introduced. The specification of the block substructure is given in a remote *block substructure diagram* (see Example I.2-2), containing three blocks: *NentityA*, *NentityB* and *N_1service*. The first two blocks represent (N)-protocol entities, while the block *N_1service* represents the (N – 1)-service provider. The specification of *N_1service* is analogous to the specification of *Nservice*, and is not shown in this diagram (being a remote specification).

A protocol entity block contains one or more processes, depending on the characteristics of the protocol. In this case, two processes have been chosen, *Ncom* and *Ncodex*. Process *Ncom* handles the sending and reception of protocol data units, while process *Ncodex* takes care of the transmission of protocol data units using the underlying service. Conceptually, the processes *Ncom* communicate directly via an implicit channel *Nchan* (conveying protocol data units), but in reality they communicate indirectly via processes *Ncodex* and the underlying service provider.



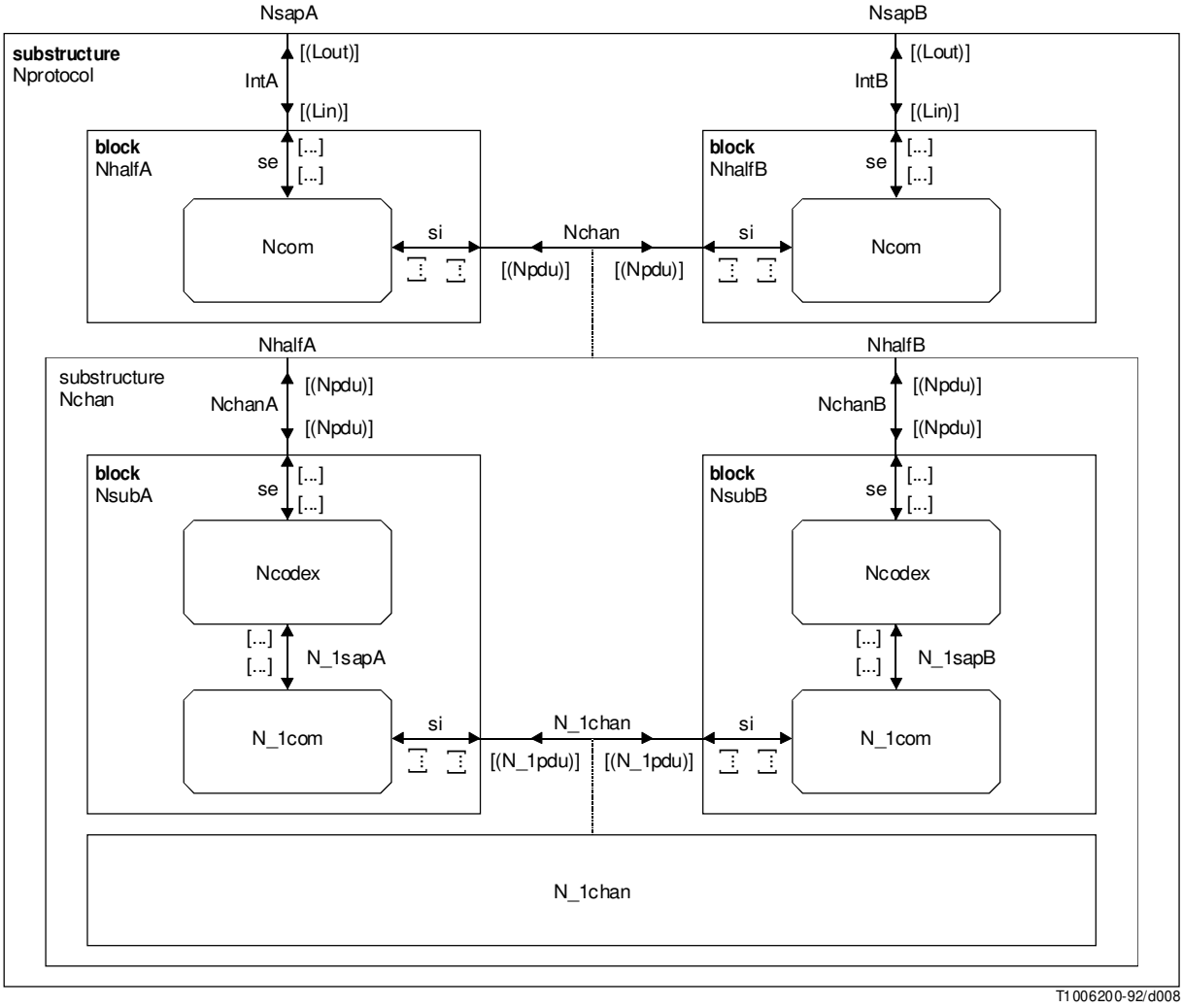
EXAMPLE I.2-2

(N)-protocol specification in SDL

I.2.1.3 Alternative approach using channel substructure

This approach is obtained from the basic approach in Example I.2-2 by grouping the processes differently, introducing the real channel *Nchan* and using channel substructure (see Example I.2-3). The channel *Nchan* conveys protocol data units, indicated by the signal list *Npdu*. This approach emphasizes the protocol view and the horizontal orientation of OSI.

Note that in this approach the blocks within a channel substructure do not represent protocol entities, and overlap two adjacent layers. The service primitives are hidden in these blocks, and are conveyed on the signal routes *N_1sapA*, *N_1sapB*, *N_2sapA*, *N_2sapB* etc. However, the chosen highest (N)-layer must be treated separately, as indicated in the example. Note also that the system diagram (see Example I.2-1) is not affected by this approach.

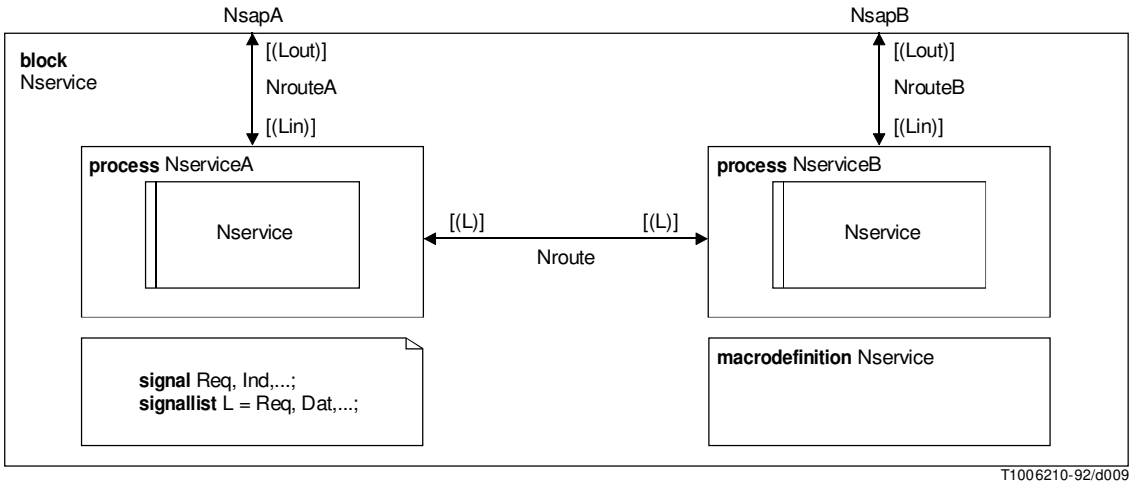


EXAMPLE I.2-3

The protocol view of OSI

I.2.1.4 Symmetrical OSI architecture

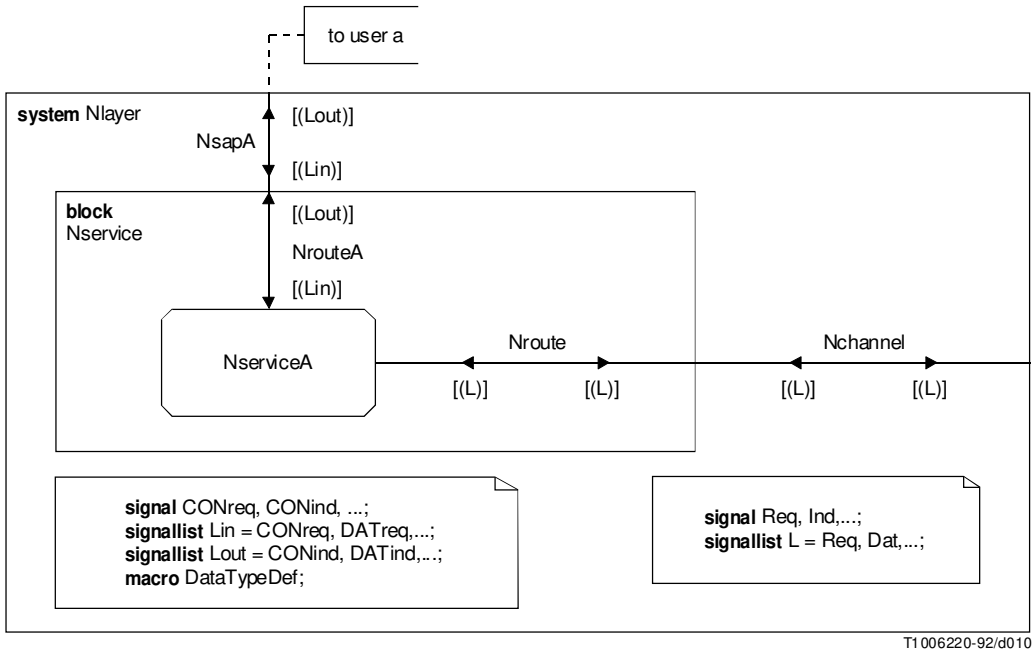
When the OSI architecture is symmetrical, that is when entities of the two sides of the OSI architecture are mirror images of each other, the specifications of these entities are identical except for the entity name. The common specification can be given only once by using a macro (see Example I.2-4). In this example only the block diagram of *Nservice* in Example I.2-1 is shown. Note that service specifications are always symmetrical, only protocol specifications can be asymmetrical.



EXAMPLE I.2-4

Using macro to represent a symmetrical OSI architecture

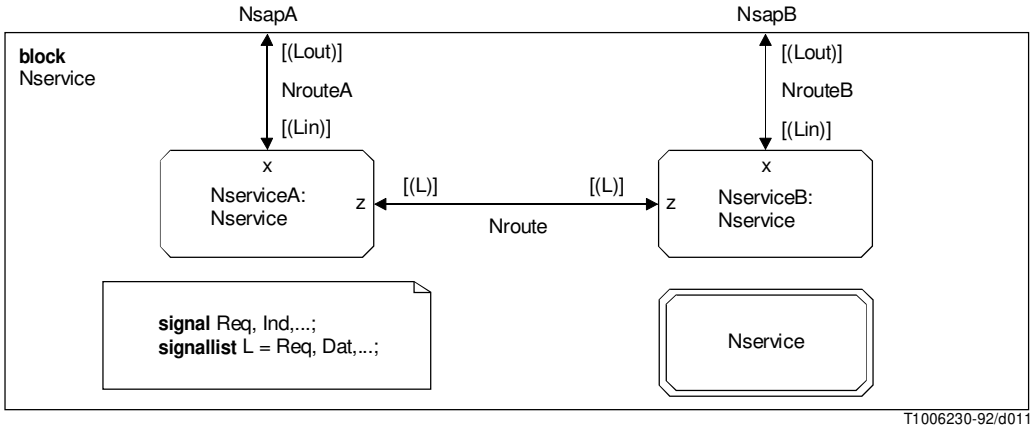
An alternative approach is to represent only one side, (see Example I.2-5), which is a modification of the system diagram in Example I.2-1. The channel *NsapB* has been replaced by the channel *Nchannel*, conveying the internal signals *L*. These signals are now on the system level, and their specifications have been moved accordingly. Note that this approach cannot be used in combination with channel substructuring (shown in Example I.2-3).



EXAMPLE I.2-5

Representing only one of a symmetrical OSI architecture

A third approach can be given by instantiation of a type. Example I.2-4 is modified by introducing a process type *Nservice* in the block *Nservice*, which is then instantiated into *NserviceA* and *NserviceB* (see Example I.2-6). *x* and *z* are actual parameters corresponding to the gates (not shown here) of the *Nservice* process type.



EXAMPLE I.2-6

Using process type to represent a symmetrical OSI architecture

I.2.2 ISDN according to I.130 and Q.65

ISDN services are described using a methodology according to Recommendation I.130 [3]. The methodology contains three stages:

- Stage 1: Service aspects;
- Stage 2: Functional network aspects;
- Stage 3: Physical network aspects.

Stage 2 is further detailed in Recommendation Q.65 [4] and consists of the following steps for each service:

- Step 1: Functional model - identification of functional entities and their relationships;
- Step 2: Information flow diagrams;
- Step 3: SDL diagrams for each functional entity;
- Step 4: Functional entity actions;
- Step 5: Allocation of functional entities to physical locations.

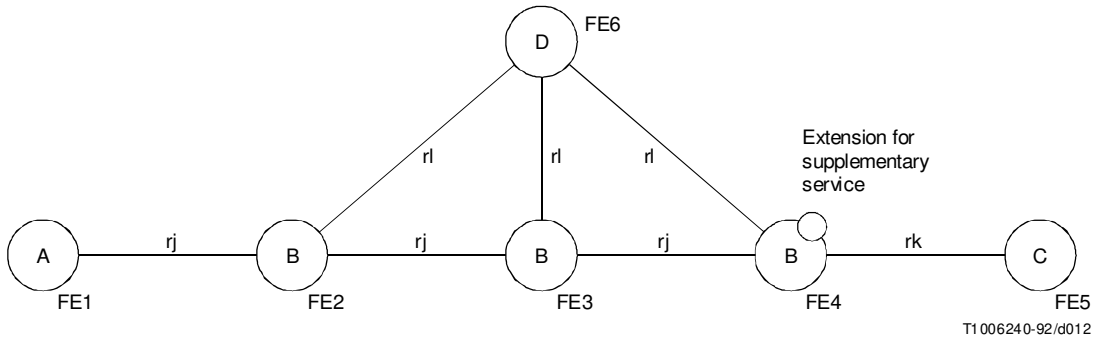
Up to 1988, SDL has only been used in step 3. This usage of SDL has been based on SDL80. The intention is to use SDL88 for new recommendations.

Ideally, it should be possible to use structural concepts of SDL in step 1, and behavioural concepts in steps 2 to 4 with stepwise introduction of data. Step 5 is outside the scope of SDL. The general argument for using SDL in steps 1 to 4 of Recommendation Q.65 is mainly that adherence to standards increases readability and enables tool support.

It will be shown below, how concepts from most steps of Recommendation Q.65 can be mapped onto SDL.

I.2.2.1 Structure

Step 1: Functional model, identification of functional entities and their relationships matches the use of structural concepts in SDL. Figure I.2-3 shows an example of a functional model according to Recommendation Q.65.



NOTE – The elements of the figure are:
– Names within circles (e.g. *A*): Types of functional entities;
– Names in upper case adjacent to circles (e.g. *FE1*): Names of functional entities (instances);
– Names in lower case between circles (e.g. *rj*): Relationships between types of functional entities;
– Added shadowing circle: Extension for supplementary service.

FIGURE I.2-3/Z.100

Example of a functional model according to Recommendation Q.65

Note that this model clearly distinguishes between types of functional entities (e.g. *A*) and their instantiation (e.g. *FE1*) when describing the structure of the system. Recommendation Q.65 also mentions the convenience of describing two functional entity types as subsets of the same single functional entity type, if the two functional entity types have much commonality. These features can be expressed by using the object-oriented extensions of SDL92, but not in SDL88.

Analysis has shown that the "functional entities" in Recommendation Q.65 can be mapped onto SDL structural concepts as follows:

- Model → system;
- Functional entity → block with one process;
- Relationship between functional entities → channel.

In addition, by using object-oriented constructs:

- Type of functional entity → block type.

A difference between the functional model according to Recommendation Q.65 and SDL structure diagrams is that SDL structure diagrams normally model open systems.

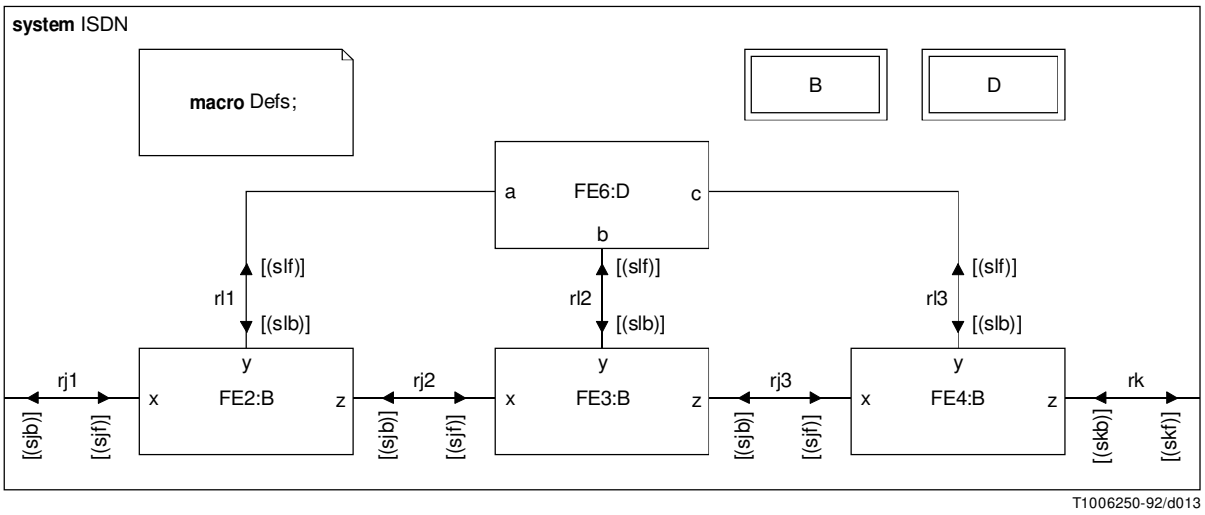
Example I.2-7 shows the functional model of Figure I.2-3 in SDL. Note that *FE1* and *FE5* are missing within the system. The communication with these functional entities is modelled as communication with the environment. Leaving *FE1* and *FE5* in the environment anticipates that one does not intend to describe the behaviour of *FE1* and *FE5*.

The distinction between relationship and relationship type can be modelled by the use of signal list definitions in the SDL structure diagrams. *B* and *D* are block types according to SDL92.

The advantages of using SDL structure diagrams are mainly that SDL structure diagrams contain definitions of signals, data types etc. which are important for subsequent steps in Recommendation Q.65. In Example I.2-7, these definitions are indicated by the macro *Defs*.

I.2.2.2 Behaviour

The behaviour is described in steps 2 to 4 of Recommendation Q.65. The interaction between functional entities is described in information flow diagrams. Based on these, an SDL diagram (i.e. a process diagram) is produced for each functional entity type.



EXAMPLE I.2-7

SDL version of the example in Figure I.2-3

The information flow diagrams correspond to message sequence charts (MSC) being standardized by WPX/3 in Recommendation Z.120 [5]. The information flow diagrams are produced for "normal" cases, and SDL is then used to describe the behaviour of the functional entities.

Step 4 of Recommendation Q.65 is concerned with defining a number of basic actions for each functional entity. The idea is then to reuse these basic actions in different service specifications. The idea is still being studied. For this approach, SDL can offer the macro or procedure concept: each basic action can be "encapsulated" in a macro or procedure. Step 4 will not be further described below.

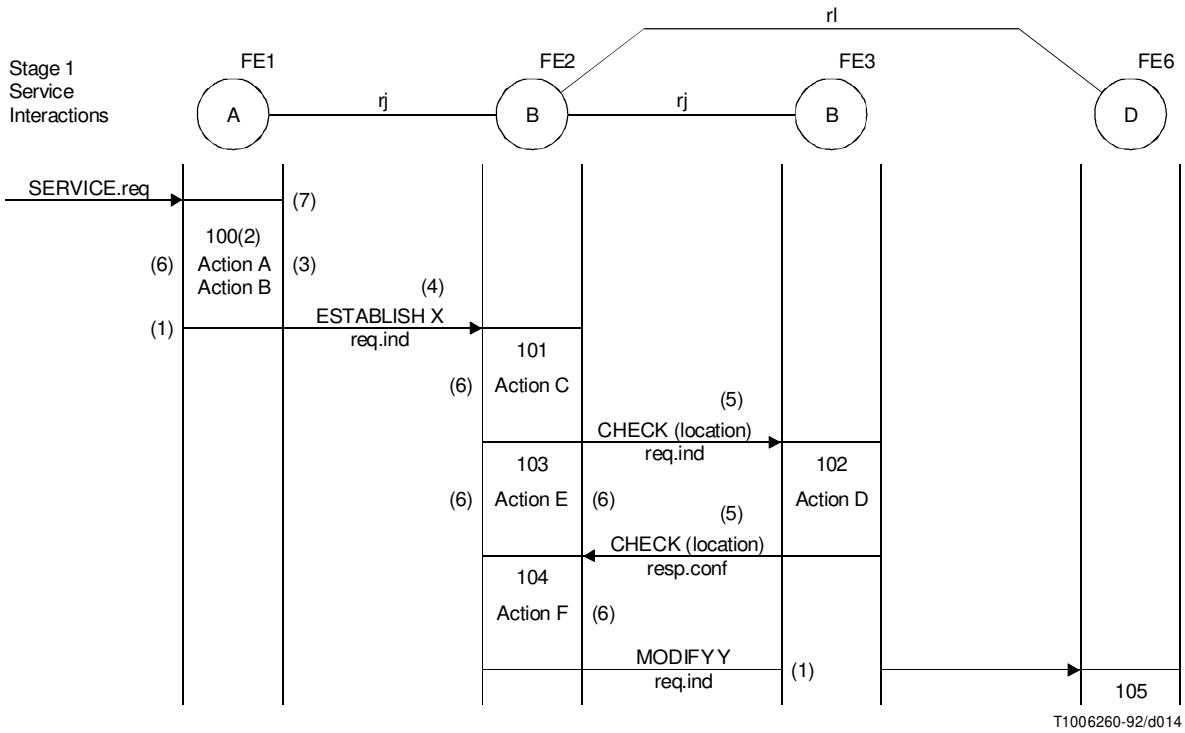
Information flow diagrams

An example of an information flow diagram is shown in Figure I.2-4.

Example I.2-8 shows the MSC that corresponds to the information flow diagram in Figure I.2-4. The mapping between the two is evidently:

- functional entity name → process instance (one process instance per block);
- information flow → message;
- additional information → message parameters;
- basic functional entity action → action;
- labels to the SDL diagrams → comments.

The information flow diagrams recommended in Recommendation Q.65 thus basically contain the same information as the MSCs. The SDL specifications can in general be checked against the MSCs during simulation, and in some simple cases (no dynamic creation of process, no dynamic addressing of outputs) this check can easily be done without simulation. MSCs can be derived from SDL specifications, but normally only a subset of the behaviour ("the normal behaviour") is meant to be shown in a MSC, so some human interaction is needed to derive the appropriate MSCs from the SDL specification. Another possibility is to derive automatically skeletons of SDL process diagrams from the information flow diagrams. See also **step 5** in I.3.2.



NOTE – The elements of the diagram are:
– Names on top of columns (e.g. *FEI*): Functional entities;
– Names in lower case between circles (e.g. *rj*): Relationships between types of functional entities;
– Names on arrows between columns (e.g. *ESTABLISH X req.ind*): Information flow;
– Names within brackets adjacent to information flow names (e.g. *location*): Additional information conveyed by the information flow;
– Names within columns (e.g. *Action B, 100*): Names of basic functional entity actions;
– Numbers within parentheses (e.g. *(5)*): Labels to the SDL diagrams.

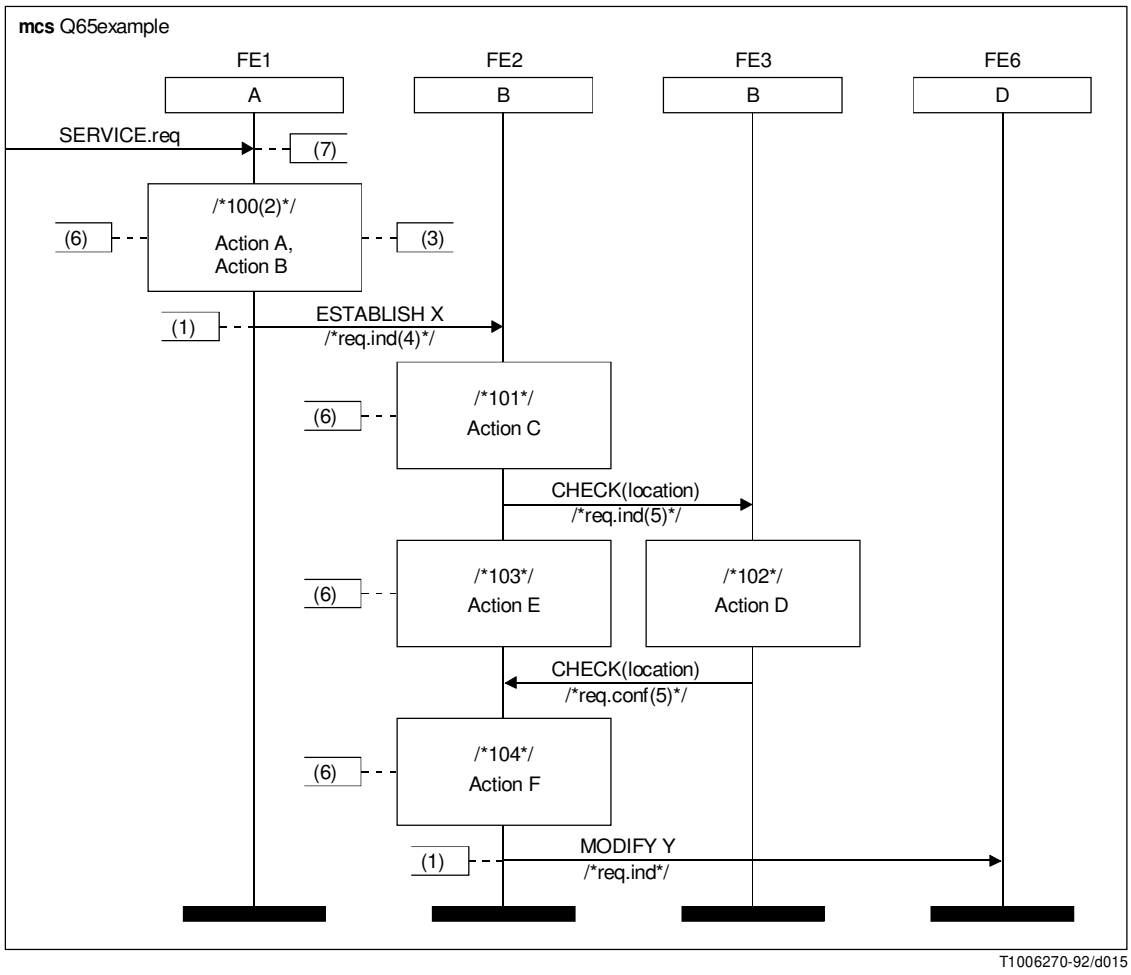
FIGURE I.2-4/Z.100

Example of information flow diagram from Recommendation Q.65

Process diagrams

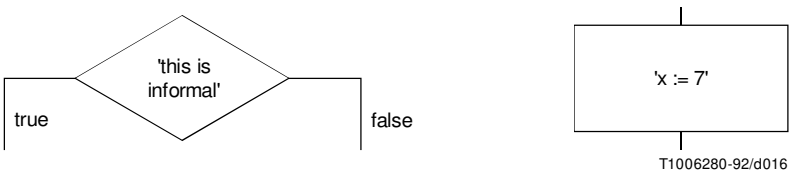
This is the part of SDL that has actually been used in the ISDN recommendations. Much work has now been invested in updating the SDL diagrams to SDL88. Some observations from this update are: All texts in symbols were informal from the start. It was rather easy to convince people about the need for formalism in **output** and **input** symbols, since these symbols refer to the interfaces of the processes (i.e. the preceding step in the established methodology). It was (and is) annoying for many people to have to put apostrophes (‘ ‘) around informal text in **task** and **decision**, since the content of these boxes is mostly informal text. Some examples are shown in Example I.2-9. Several creative suggestions have been made informally for defining a more ‘user friendly’ and perhaps less ‘tool friendly’ rule for distinction between informal and formal text. However, it seems difficult to formulate simple and unambiguous rules for informal text, without invalidating the current use of *Charstring* as informal text.

Another conflict was the tradition for assigning descriptive sequences of names to a state or a signal. The first solution suggested by SG X was to use underscores to chain a number of words into one word, e.g. “hang up req.ind” becomes “hang_up_req.ind”. Later, the lexical rules for SDL88 was reformulated, making “hang up req.ind” a legal name in SDL. The use of names in SDL88 now matches the established use of names.



EXAMPLE I.2-8

MSC version of the example in Figure I.2-4

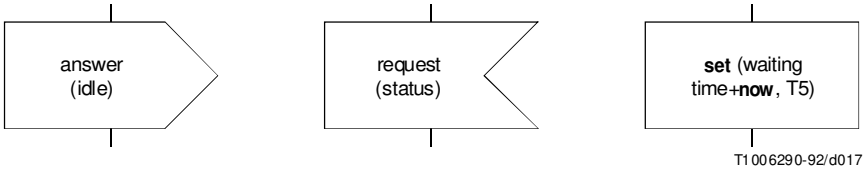


EXAMPLE I.2-9

Informal text

I.2.2.3 Data

Tasks and **decisions** are mainly informal in the existing SDL-diagrams for ISDN. Therefore, there is little need of defining operators for data types. Data mainly appear in **input**, **output** and **timer** constructs, see Example I.2-10.



EXAMPLE I.2-10

Use of data

In most cases, *Boolean* and *Integer* are sufficient, e.g. to carry the value of some flag or counter. *Charstring* is also widely used, because it allows one to handle parameters in a way close to informal text.

The use of enumerated types could be useful. A **newtype** with only literals matches an enumerated data type (as in e.g. Pascal):

```
newtype Indication

literals idle, busy, congestion;

endnewtype Indication;

...

signal Check location (Indication) /*resp.conf */;

...

output Check location (busy) /*resp.conf*/;

...
```

Data are introduced in steps 1 to 4 when formalising the specifications, e.g. signal specifications apply data types. Data can be introduced on several refinement levels. The benefit of introducing data is of course that it can be checked that input and output parameters, questions and answers, etc. are consistent throughout the whole SDL specification.

I.3 Stepwise production of an SDL specification

This subclause presents one method of producing an SDL specification. The text does not imply a recommendation of this particular method by CCITT, since it is recognised that an SDL specification may be obtained by many different methods. The method can be considered guidelines for developing an SDL specification through well defined steps, although the result of each step may not be a complete and formal specification. Adaptations of the method may very well include rearranging some of the steps, etc.

The steps are identified from a user's point of view, and formally by stating the language constructs introduced. The method is illustrated by a simple example.

The method presented here only introduces a subset of SDL, that is required for the simple example used. A more elaborated method would include more concepts of the language, e.g. *services* for reducing the complexity of *processes*.

I.3.1 Introduction

The production of a formal specification normally starts with an informal and incomplete specification. The production process is completed when the specification is both formal and complete.

The term "formal" is taken to mean "complies with the rules of the formal language", i.e. a formal SDL specification complies with the concrete syntax and the static semantics of SDL. Moreover, it contains no *informal text*.

The term "complete" is taken to mean "conveys all the information about the system as intended by the author".

The task of producing a complete formal specification is considered difficult, especially by people with little training in this area. The training situation for Formal Description Techniques is quite different from that of e.g. programming languages, and the methodological support for formal specification work must take this into account.

Since each step sticks closely to SDL rules, dedicated SDL tools can be utilized to a large extent.

SDL specifications can be produced with respect to:

- structure;
- behaviour;
- data.

The specification is of course not complete, until all three aspects are completely covered. However, it can be a formally valid SDL specification before that stage.

It is the intention to cover all three aspects in parallel. The principles followed for the definition of the steps are that each step:

- is a natural halting point in the production process;
- it should produce a self-contained result that appears meaningful and can be checked, preferably by tools;
- is not (or to a small extent) dependent on successive steps.

Each step is described by:

- **Description** – The conceptual process for the step. It is formulated by using general concepts and the corresponding SDL concepts (in *italics*). As far as possible, some heuristics are given. It is partitioned into a summary of actions and a discussion part.
- **Result** – The result of performing the step.
- **Example** – The step applied to the specification of a small lift system (the complete specification is given in I.3.3). *Example* has been omitted for some steps, in order to limit the size of the clause.

I.3.2 The steps

Step 1 – System boundary

Description

- Delimit the *system* from its environment. Find a suitable name for the *system*. Identify the agents in the environment of the *system* (i.e. the different entities which the *system* will interact with). Describe the purpose and characteristics of the *system* informally in a *comment*.
- Specify one *channel* for each identified agent in the environment, and give a suitable name corresponding to the name of the agent.
- Introduce one dummy *block* within the *system*. This *block* will later be replaced by the actual system structure.

– Identify the information flow, in terms of discrete events, through the *system* boundary. These events constitute the system alphabet, and are modelled by *signals*. The *signals* for external communication are specified on the *system* level. State the purpose of each *signal* in a comment for each *signal specification*. Identify the information to be conveyed by *signals*, and indicate the *sort* of *signal parameters*. Use syntype, enumerated and predefined *sorts* as far as possible. Associate *signals* to *channels*, either directly or by using *signal lists*.

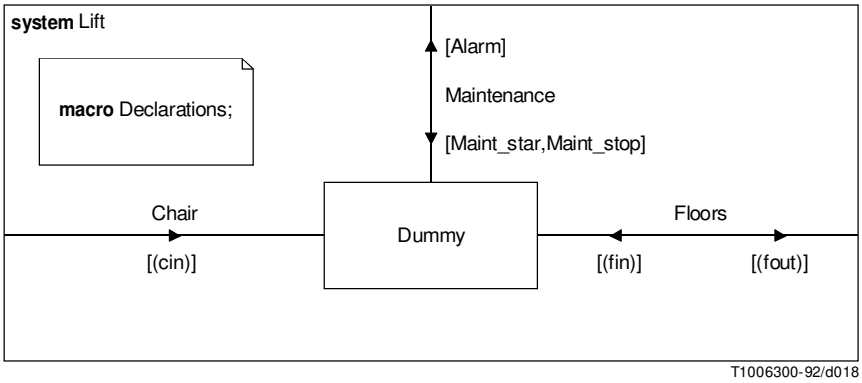
– Provide a skeleton specification (without signature) of each new *sort* introduced .

One must keep in mind that the number of *signals* can be reduced by qualifying *signals* with *signal parameters*.

It is realized that this step is a great simplification of the considerable efforts needed in the early phases of requirement collections, systems analysis etc. Projects may profit from the use of other techniques than SDL, as a prelude to this step. Object-oriented analysis is covered in more details in I.4.

Result – The specification of the boundary of a *system*, having its internal structure undefined.

Example




```
macrodefinition Declarations;

/*  A lift consists of a chair with GOTO(floor) buttons and floor controls where at
each floor the users can ask for upwards or downwards service.

This specification does not model the mechanical part of the lift.

The service order is: If there are more floors to be served in the current direction,
then continue to the next of these floors. If there is any floor to be served in
opposite direction, then change direction. If there are no floors waiting to be
served, stay at the current floor.

A user can control the lift by:

- pressing a button at any floor for service in the desired direction;

- pressing a button in the lift chair for a certain floor.

In addition, the specification also covers some unusual cases, such as start and stop
of the operation of the lift.        */
```

```
block Dummy;
/* minimal block syntax */
process Dummy;start;stop;endprocess;
endblock;
signal
  Goto(Floor),                /* request to go to the given destination  */
  Floor_req(Direction,Floor), /* request for the lift from a floor */
  Open_door(Floor),           /* open door at floor                */
  ...........
signallist cin = Goto, .......;
signallist fin = Floor_req, .......;
signallist fout = Open_door, .......;
syntype Floor
......
endsyntype;
newtype Direction
......
endnewtype;

endmacro Declarations;
```

Step 2 – System structure

Description

- Identify the main, conceptual constituents within the system and name them. These are the *blocks* of the *system*. Find a suitable name for each *block* and describe the *block* and its relation to its environment (its enclosing structure) informally in a *comment* within the *block specification*.
- Connect the *blocks* to the *system* boundary with *channels*, already introduced in **step 1.**
- Identify the information flow (*channels* and associated *signals*) between *blocks*. Specify new *signals* introduced, as in **step 1**, on the system level.
- Provide a skeleton specification of each new *sort* introduced, as in **step 1**, on the system level.

It is advised not to have many *blocks* in the same enclosure. If the number of *blocks* is too large, i.e. hindering overview, nesting (examined in **step 3**) should be applied. According to heuristics: a *block* must have strong internal cohesion and must be easily understandable on its own. A block must correspond to a concept on its own, e.g. “local exchange”, “lift control”.

Properties to be taken into account when identifying *blocks* are:

- a *block* delimits visibility: local *signals* and *sorts* can be specified within a *block*;
- communication between *blocks* (i.e. *channels*) possibly involves delay.

A *block* is considered a system of its own: It defines an external alphabet for the subsystem inside the *block* and the boundary to its environment.

A block can be specified directly as a block instance. However, object oriented SDL makes a clear distinction between *type* and *instance*. If the block or another block similar to it, is going to be used somewhere else in the system, one should first identify the more basic features of the similar blocks in a block type. The actual blocks can then be instantiated from the general type or specializations of the general type. This observation is a simplification of object oriented methods which advocate the definition of concepts in *type hierarchies,* etc. See I.4 for more details.

A *channel* may be associated with a transmission delay, so special modelling effects can be achieved by having several *channels* between two constituents. Normally, two *blocks* within the *system* need only to be interconnected via one *channel*, but modelling by more *channels* can be useful in special cases.

The use of message sequence charts (MSC) can be considered already for this step, to get a useful overview of typical communication scenarios between system constituents (the use of MSC is discussed in **step 5** and is covered in detail in I.6).

Result – Identification of *blocks* at *system* level.

Example – In this example only one block is needed, and it is specified directly.

```
block Control;

/* The block shall describe the controls of the complete lift system */

/* minimal process syntax */

process Dummy; start; stop; endprocess;

endblock;
```

Step 3 – Block partitioning

Description

- Partition each complex *block* into *subblocks* (this is the same as **step 2** for the *system*).
- Repeat this until there are no complex *blocks* left.

If the system is large, some *blocks* can be considered systems on their own, and be further partitioned according to the rules given for *system*. This results in nesting of *blocks* for increased overview by information hiding. In the resulting structure, each *unpartitioned block* is a place-holder for the specification of behaviour. (The behaviour is described in successive steps by a number of *processes* within each *block*.)

Result – A *block* tree having the *system* as the root and *unpartitioned blocks* as the leaves.

Example – None, since we claim that the *Lift* system is so small, that no block partitioning is needed.

Step 4 – Block constituents

Description

- Identify the activities within each unpartitioned block. These are the *process sets* of the *block*. Find a suitable name for each *process set* and specify it and its relation to its environment (the enclosing *block*) informally in a *comment* within the *process specification*.
- Connect the *process sets* to *channels* at the *block* boundary with *signal routes***.**
- Identify the information flow (*signal routes* and associated *signals*) between *process sets*. Specify new *signals* introduced, as in **step 1**.
- Provide a skeleton specification of each new *sort* introduced, as in **step 1**.

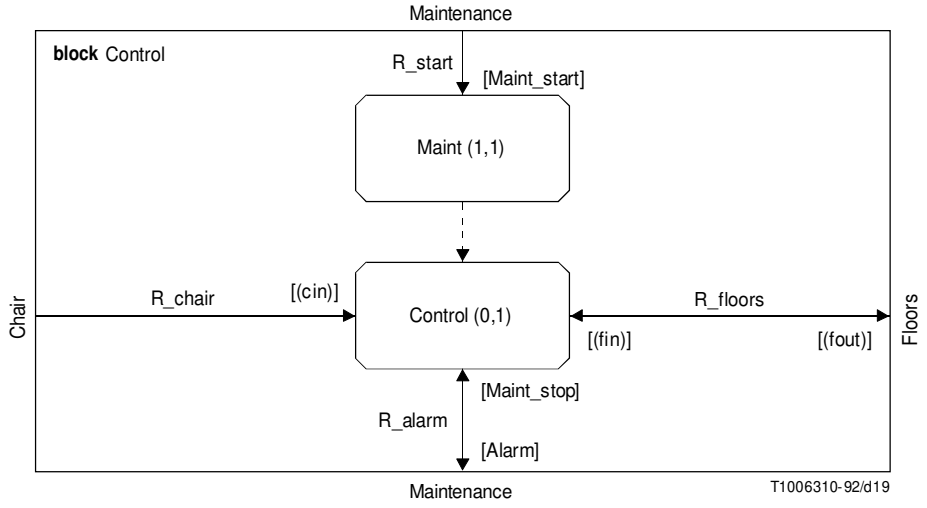
Specify for each process set the *number of instances*, i.e. *initial* and *maximum number* of instances. For each *block*, at least one *process set* must have the *initial number* greater than zero. It can be useful to state the *maximum number* if each *process instance* corresponds to a limited resource (e.g. some hardware device). The considerations of *type* identification as stated in **step 2** apply also to this step for *process types*.

Guidelines for the identification of *process sets* are: each *process set* represents a pattern of activity, and this pattern can exist in a number of concurrent instances. Most of the heuristics from **step 2** apply to this step as well.

The input alphabet of a *process set* can be specified either by *signal routes* or by a *signalset*. A *signal route* is a communication path between *process sets* within a *block* or between *process sets* and the surroundings of a *block* (compare with *channel* which is a communication path between *blocks* and their surroundings).

Result – Identification of *process sets* within the unpartitioned *blocks*.

Example




```
process Control;
/* controls the operation of the lift */
start; stop; /* minimal process syntax */
endprocess;
process Maint;
/* initializes the operation of the lift */
start; stop; /* minimal process syntax */
endprocess;
```

Step 5 – Skeleton process specifications

Description

- Identify the typical use cases, and describe these e.g. by Message Sequence Charts (MSCs).
- Make necessary additional decisions concerning how to model the behaviour. This may require the introduction of new *signals* and *sorts*, these are specified as in **step 1**.
- Write a skeleton *process specification* covering the use cases, but do not yet consider the combination of these.
- Consider the use of *procedures* to hide details and *timers* for time supervision. Introduce *external synonyms* for unspecified values.

The order of events on a vertical axis (see example below) in a MSC gives one ordering of events in a *process specification*. This ordering constitutes (part of) a skeleton *process specification*. MSCs can even be used in earlier steps to specify the communication within the *system*. In this case, an axis of a MSC may denote a whole *block*. The use of MSC is covered in detail in I.6.

Starting from the start symbol of each *process specification*, build a tree of *states* by considering the "normal" traversing of the *process specification*. Introduce dynamic *creation* of *process instances* if needed, but *parameters* are not included for *process creation* yet. Indicate *parameters* in *inputs* and *outputs*.

Time supervision can also be shown in an MSC. According to heuristics: time supervision is used to model a time-span within the model, to supervise the release of a resource, and to supervise replies from non-reliable sources (e.g. another network node). Time supervision is achieved by the introduction of *timers* and by *set* and *reset* actions.

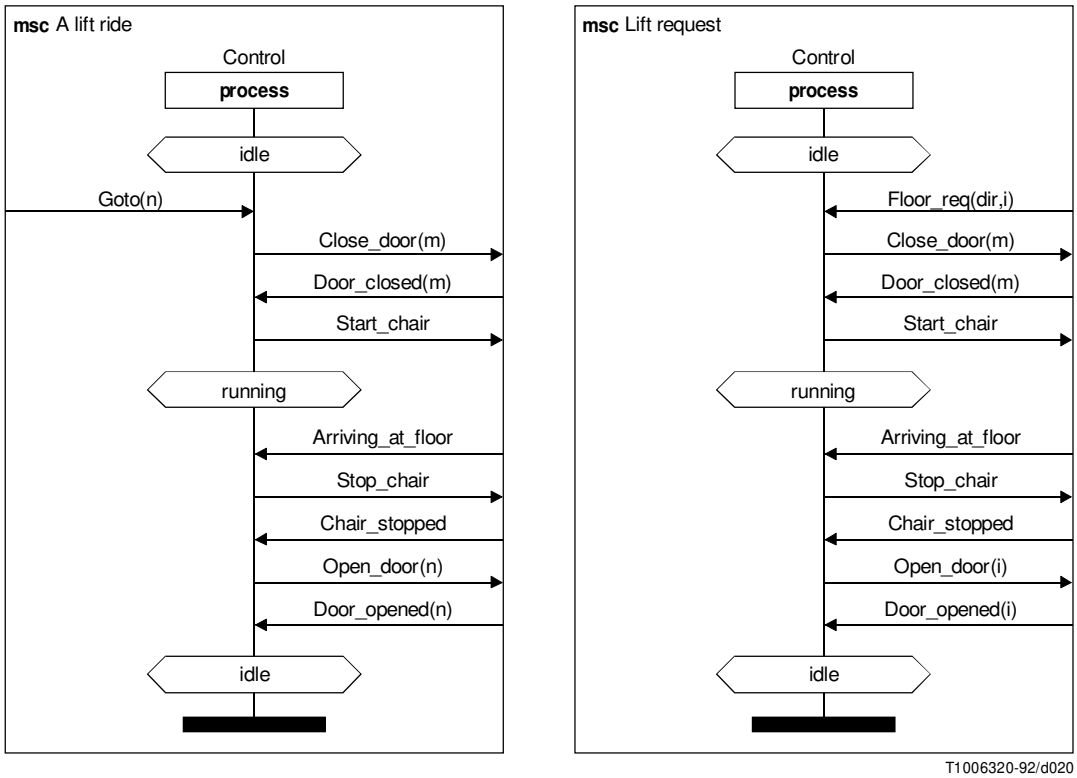
If some values are naturally left unspecified (e.g. the top floor in the lift example), the described system is *generic*. This can e.g. be expressed by using *external synonyms*. Considering which parts of a specification should be generic is important for increasing the usefulness of a specification (i.e. an SDL specification of a lift system for a house with any number of floors is more useful than one for a house with say 14 floors!).

Result – MSCs and skeleton *process specifications*.

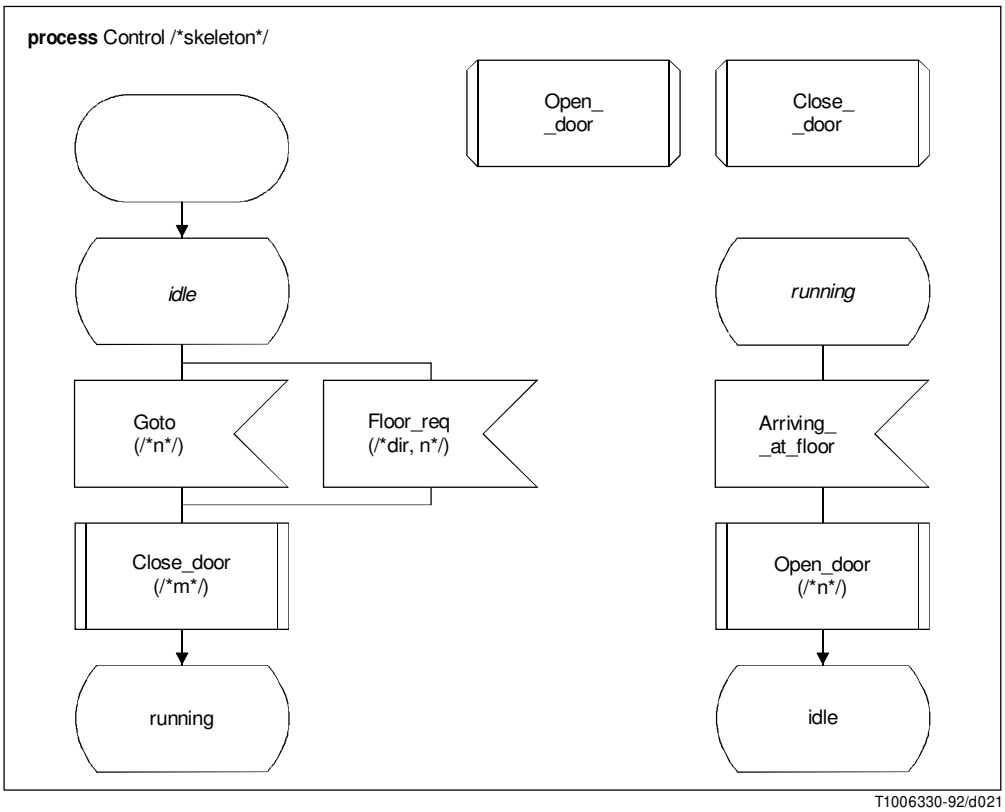
Example – Some decisions about modelling the behaviour must be made, for example how to indicate the arrival of the lift chair at a floor. Here, this is done by the reception of a signal *Arriving_at_floor* from the environment.

The MSC *A lift ride* describes the following simple situation: A user standing at the *m*th floor enters the lift chair (which happens to be waiting for him at the floor), presses the button *goto(n)* and is transported to the *n*th floor where the door opens on arrival.

Now, if the lift chair happens to be waiting at another floor, then the user must first call for the lift. This situation is depicted in the MSC *Lift request* (The user is standing at the *i*th floor).



The skeleton *process specification* covers the single-user case, and is derived from the above MSCs (note that floor *i* is treated in the same way as floor *n*). Some detailed signalling is hidden in the process specification by the use of the procedures *Open_door* and *Close_door*:



Step 6 – Informal process specifications

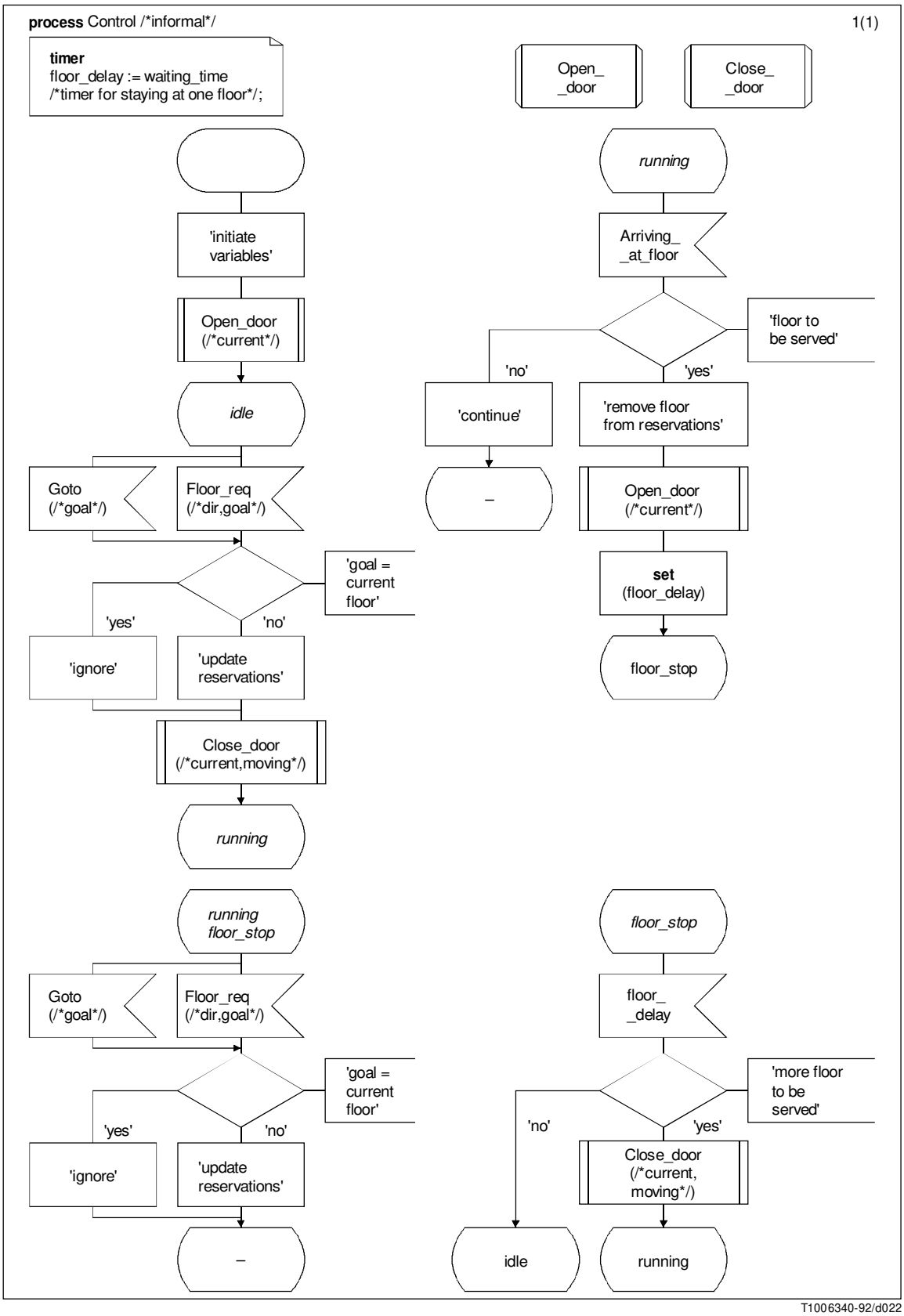
Description

- Consider the combination of use cases, and describe these, if appropriate, using MSC.
- Identify the information that needs to be stored in the *processes*.
- Introduce *tasks* and *decisions*, but use only *informal text* in these. Introduce possibly new *states*.
- Write a skeleton specification of each *procedure*, and indicate the *sort* of *procedure parameters*.
- Indicate the *sort* of *process formal parameters*.
- Specify each new *sort* introduced, as in **step 1**.

Choice among different *inputs* is accomplished through a *state*. Choice among *outputs* are accomplished through an *informal decision*. Choice among *inputs* and *outputs* are accomplished by a combination of *informal decision* and *state*.

Result – Informal *process specifications*.

Example – In a multi-user case, *Goto* and *Floor_req* can be received while the lift chair is running, and these requests must be stored in a reservation table. A new state *floor_stop* is introduced for a temporary stop at a floor. The duration of a temporary stop is determined by the timer *floor_delay*.


process Control /*informal*/
1(1)
timer
floor_delay := waiting_time
/*timer for staying at one floor*/;
Open_
_door
Close_
_door
'initiate
variables'
running
Arriving_
_at_floor
Open_door
(/*current*/)
'floor to
be served'
'no'
'yes'
idle
'continue'
'remove floor
from reservations'
Goto
(/*goal*/)
Floor_req
(/*dir,goal*/)
–
Open_door
(/*current*/)
'goal =
current
floor'
set
(floor_delay)
'yes'
'no'
'ignore'
'update
reservations'
floor_stop
Close_door
(/*current,moving*/)
running
running
floor_stop
floor_stop
Goto
(/*goal*/)
Floor_req
(/*dir,goal*/)
floor_
_delay
'goal =
current
floor'
'more floor
to be
served'
'yes'
'no'
'no'
'yes'
'ignore'
'update
reservations'
Close_door
(/*current,
moving*/)
–
idle
running
T1006340-92/d022

The multi-user case can also be described by MSCs (see I.9.1.2).

```
macrodefinition Declarations;
......
synonym waiting_time Duration = external;
       /* duration of stay at one floor */

endmacro Declarations;
```

Step 7 – Complete process specifications

Description

- Consider also unusual cases, such as error situations.
- Complete the procedure specifications.
- Check that all signal-state combinations are covered.

This step terminates, when no uncovered *nextstates* are introduced in *transitions*. An uncovered *nextstate* corresponds to a *state* which has not been considered yet. A *signal-state matrix* can be useful here (see I.9.4).

Result – Complete but informal *process specifications*.

Step 8 – Formal process specifications

Description

- Identify *sorts* for stored information. Specify the *signature* of each *sort* introduced so far, and use *informal text* for *equations.*
- Specify *variables* for stored information and *input parameters.* Specify *process formal parameters* .
- Change *informal text* in *tasks*, *decisions* and *answers* to *assignments* and *expressions.*
- Introduce parameters in *inputs*, *outputs*, *create requests* and *procedure calls.*

The *signature* of a *sort* covers the *operators* with typing and the *literals*. Inherit as much as possible from predefined *sorts*. Identify (in a *comment*) the set of *operators* and *literals* which can be used to represent all possible values. These are the constructors of the *sort* (see I.5).

Result – Semi-formal *sort specifications* and formal *process specifications*.

Example

```
macrodefinition Declarations;
......
newtype Direction
  literals up, down;/* constructors:up,down */
  operators
   change_dir: Direction -> Direction;
  axioms
   'if direction is up then down, else up'
endnewtype;
........
endmacro Declarations;

process Control;
dcl
  goal Floor,               /* the target floor */
  towards Direction,        /* the indicated direction */
  here Floor,               /* current floor */
  moving Direction,         /* current direction of lift chair */
  table Reservations,       /* table of reservations */
  ........
endprocess;
```

Furthermore:

```
task 'initiate variables';
```

becomes:

```
task here := bot_floor;
task moving := up;
task table := empty_serv;
```

and

```
decision 'goal = current floor';
```

becomes:

```
decision goal = here;
```

Step 9 – Formal sort specifications

Description

- Formalize the *equations* by substituting informal *equations* by formal ones.
- Add *equations* to the *sort specifications*, until the set of *equations* is complete.
- Alternatively to *equations*, specify operators by means of *operator specifications*.

Specify first *equations* for the constructors. This gives the possible values of the *sort*. Then specify *equations* for the remaining *operators* and *literals*. The set of equations is complete, when all expressions containing non-constructor *operators* and *literals* can be rewritten into expressions only containing constructor *operators* and *literals*. See I.5 for more details.

Operator specifications are similar to value returning procedures (in fact, they are defined by a transformation to such procedures), and are recommended for those who are not familiar with equations.

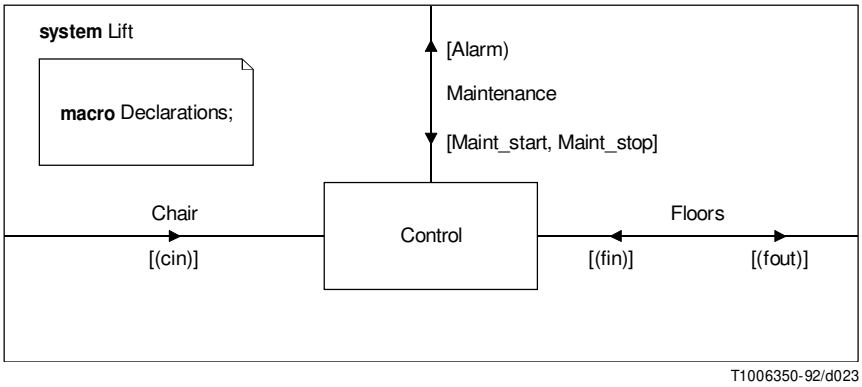
Result – Complete and formal *sort specifications*.

Example

```
macrodefinition Declarations;
......
newtype Direction
  literals up, down;/* constructors:up,down */
  operators
   change_dir: Direction -> Direction;
  axioms
   change_dir(up) == down;
   change_dir(down) == up;
endnewtype;
synonym waiting_time Duration = external;
       /* duration of stay at one floor */
newtype Reservations array(Floor,Floor_indicator) adding
  literals empty_serv;
  operators
   goto_req: Reservations, Floor, Floor -> Reservations;
    /* marks a request to go from current floor to another floor  */
..........
  operator goto_req;
   fpar table Reservations, here Floor, goal Floor;
   returns Reservations;
   start;
    decision goal > here;
      (true):
        task table(goal)!upwards := true;
      (false):
        decision goal < here;
         (true):
```

```
            task table(goal)!downwards := true;
          (false):
             /* no action */
        enddecision;
    enddecision;
    return table;
  endoperator;
........
endnewtype Reservations;
........
endmacro Declarations;
```

I.3.3 Example: The lift

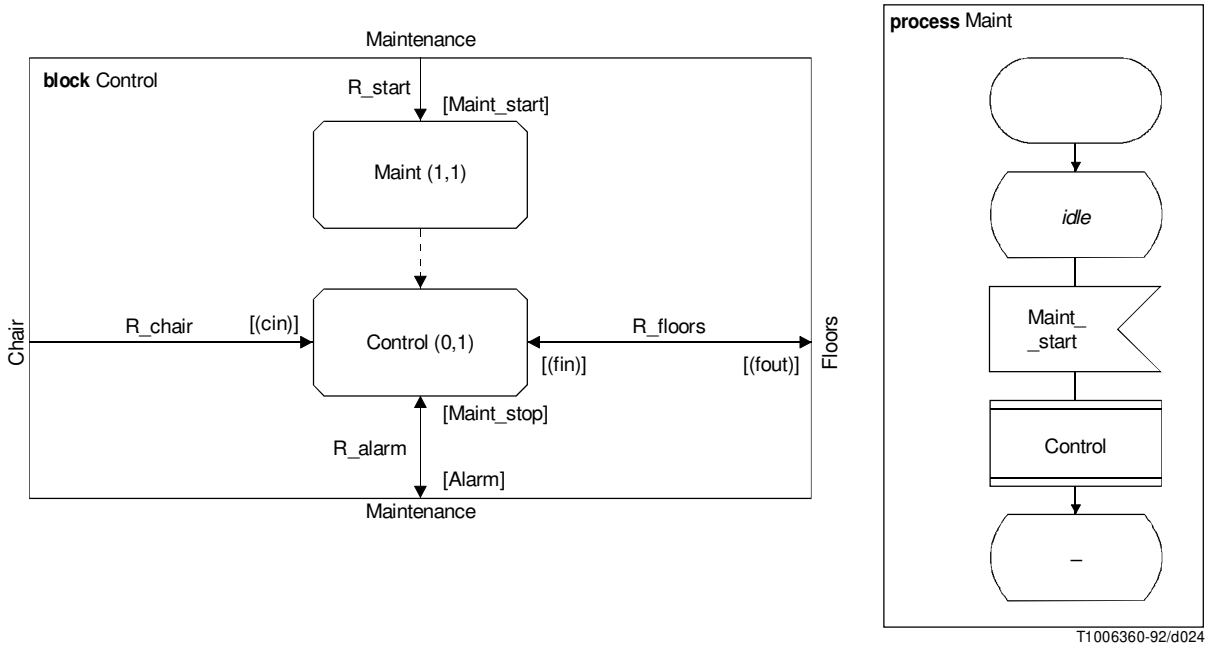


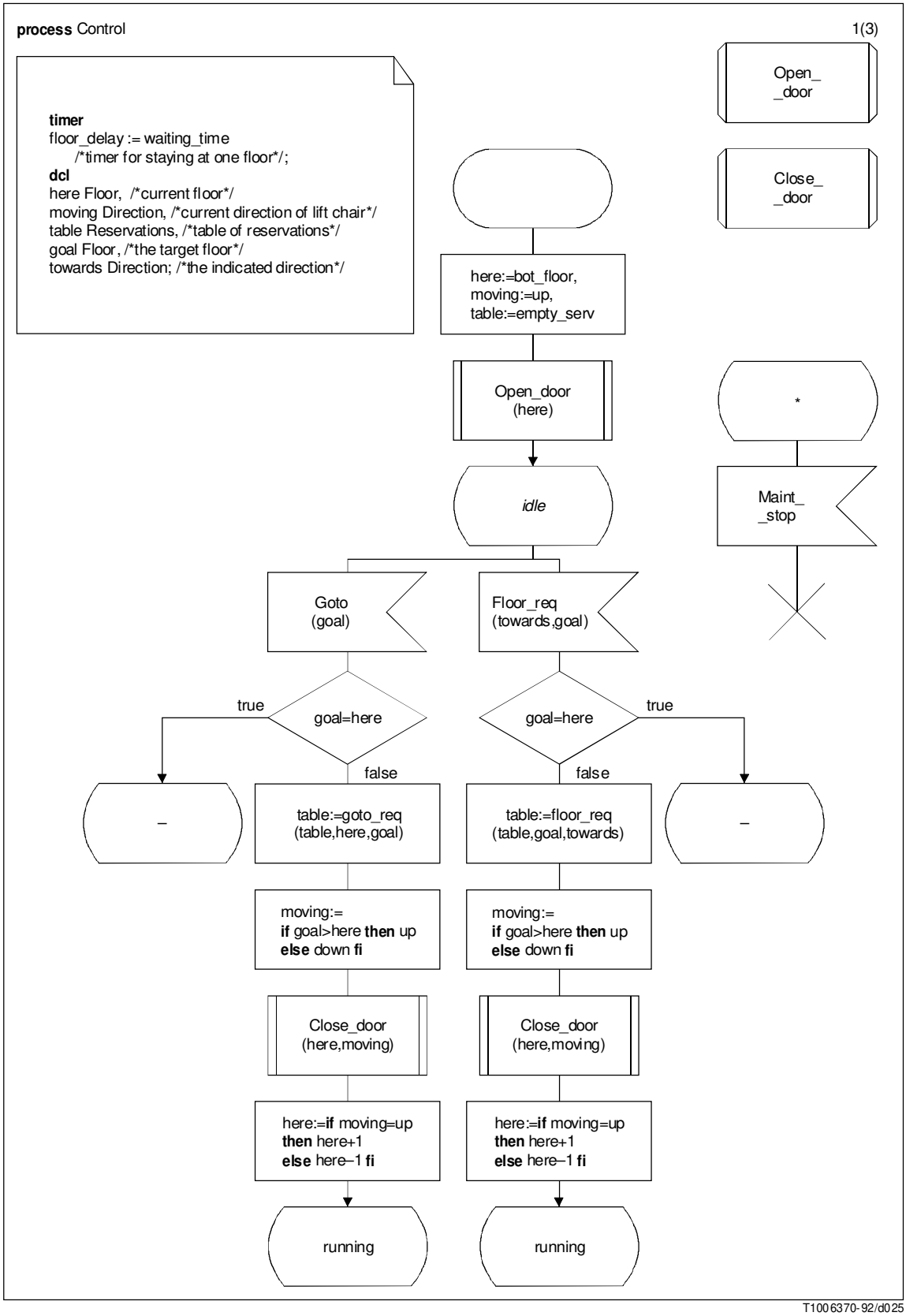
T1006350-92/d023

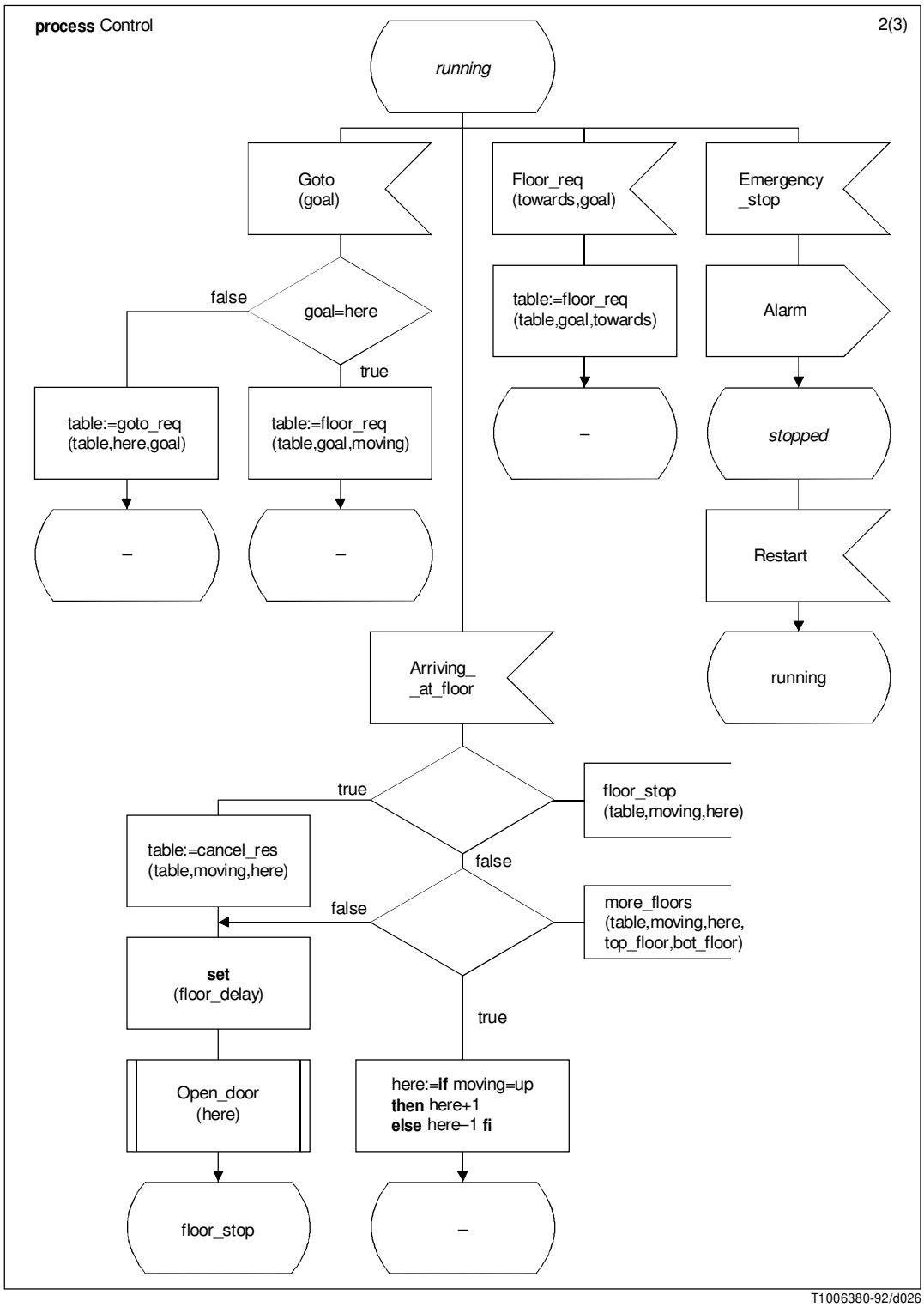
```
macrodefinition Declarations;

/*   A lift consists of a chair with GOTO(floor) buttons and floor controls where at
each floor the users can ask for upwards or downwards service.

This specification does not model the mechanical part of the lift.

The service order is: If there are more floors to be served in the current direction,
then continue to the next of these floors. If there is any floor to be served in
opposite direction, then change direction. If there are no floors waiting to be
served, stay at the current floor.

A user can control the lift by:
- pressing a button at any floor for service in the desired direction;
- pressing a button in the lift chair for a certain floor.

In addition, the specification also covers some unusual cases, such as start and stop
of the operation of the lift.       */

signal
  Alarm,                      /* alarm message                 */
  Arriving_at_floor,          /* arriving at a floor       */
  Chair_stopped(Floor),       /* chair stopped at floor               */
  Close_door(Floor),          /* close door at floor               */
  Door_closed(Floor),         /* door closed at floor              */
  Door_opened(Floor),         /* door opened at floor             */
  Emergency_stop,             /* stop chair immediately            */
  Floor_req(Direction,Floor), /* request for the lift from a floor */
  Goto(Floor),                /* request to go to the given destination  */
```

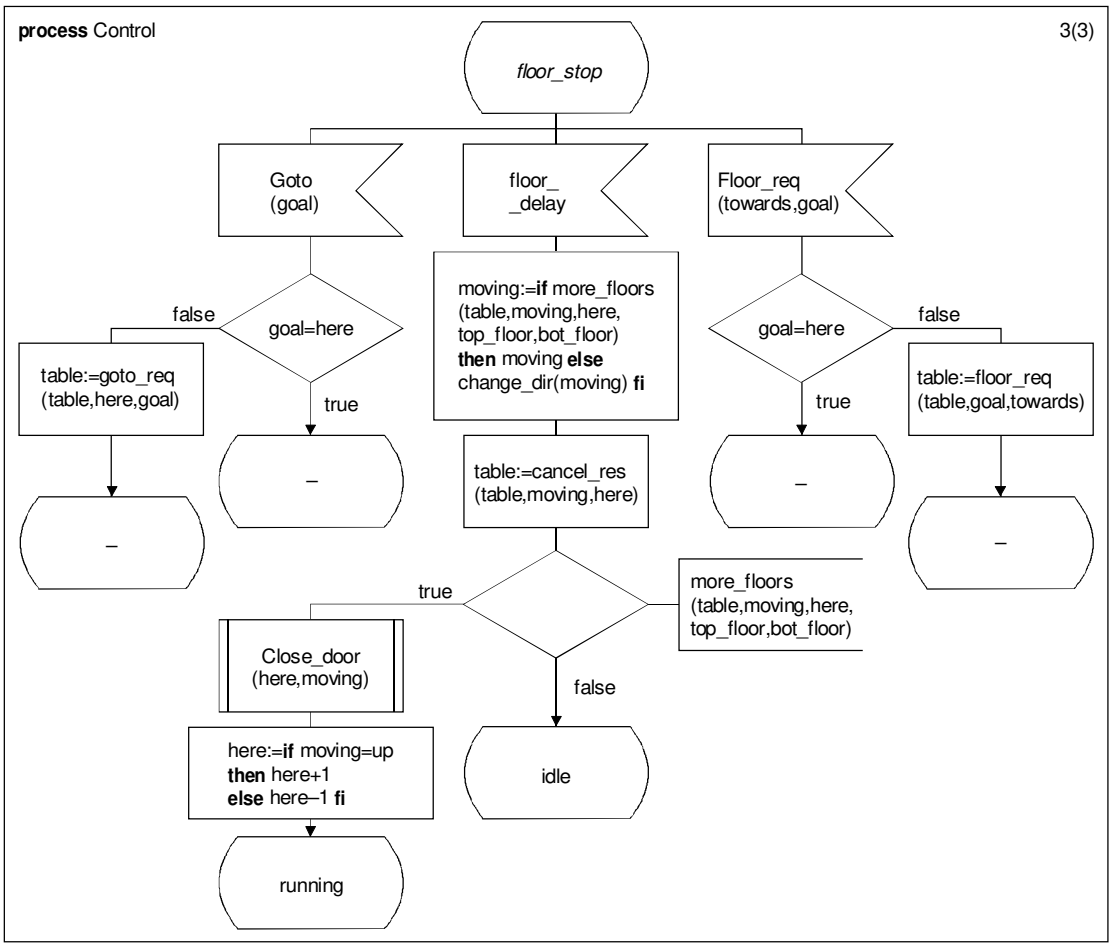
```
  Maint_start,                /* start lift operation          */
  Maint_stop,                 /* stop lift operation          */
  Open_door(Floor),           /* open door at floor                */
  Restart,                    /* restart stopped chair             */
  Start_chair(Direction),     /* start chair in the given direction */
  Stop_chair(Floor);          /* stop chair at floor               */
synonym top_floor Natural = external    /* the top floor */;
synonym bot_floor Natural = external     /* the bottom floor */;
synonym waiting_time Duration = external;
       /* duration of stay at one floor */
signallist cin = Goto, Emergency_stop, Restart;
signallist fin = Floor_req, Door_closed, Door_opened, Chair_stopped,
  Arriving_at_floor;
signallist fout = Open_door, Close_door, Stop_chair, Start_chair;
syntype Floor = Natural
  constants bot_floor : top_floor
endsyntype;
newtype Direction
  literals up, down;
  operators
   change_dir: Direction -> Direction;
  axioms
   change_dir(up) == down;
   change_dir(down) == up;
endnewtype;
newtype Floor_indicator
  struct upwards Boolean; downwards Boolean;
endnewtype Floor_indicator;
newtype Reservations array(Floor,Floor_indicator) adding
  literals empty_serv;
  operators
   goto_req: Reservations, Floor, Floor -> Reservations;
    /* marks a request to go from current floor to another floor  */
   floor_req: Reservations, Floor, Direction -> Reservations;
    /* marks a request to go from a floor in the given direction  */
   floor_stop: Reservations, Direction, Floor -> Boolean;
    /* checks if the floor has requested service in the given
       direction */
   cancel_res: Reservations, Direction, Floor -> Reservations;
    /* removes a service request from floor in the given direction */
   more_floors: Reservations, Direction, Floor, Floor, Floor -> Boolean;
    /* checks if there are more floors to be served from the current
     floor to top or bottom depending on the current direction   */
   more_floors!: Reservations, Direction, Floor, Floor, Floor -> Boolean;
    /* additional operator needed for iteration over a range  */
  operator goto_req;
   fpar table Reservations, here Floor, goal Floor;
   returns Reservations;
   start;
    decision goal > here;
      (true):
       task table(goal)!upwards := true;
      (false):
       decision goal < here;
        (true):
          task table(goal)!downwards := true;
```

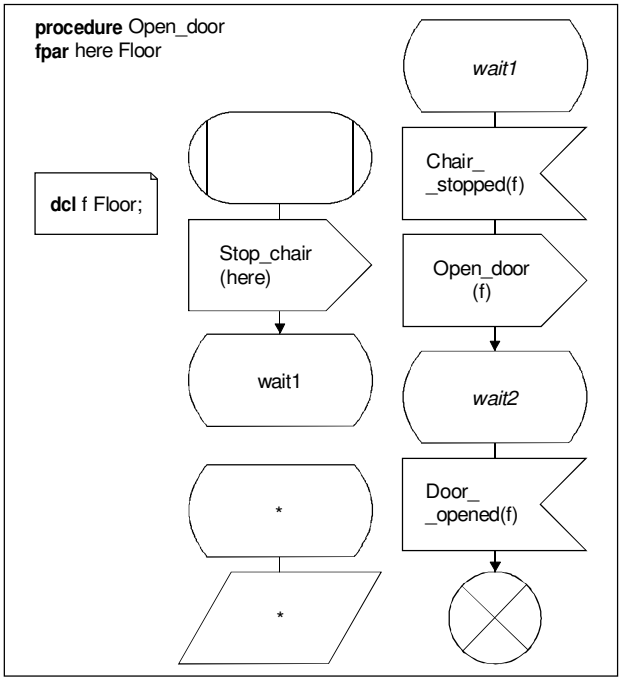
```
            (false):
               /* no action */
            enddecision;
         enddecision;
         return table;
      endoperator;
      axioms
         empty_serv == Make!(Make!(false,false));
         floor_req(r,goal,d) ==
            Modify!(r,goal, if d = up
                 then upwardsModify!(Extract!(r,goal),true)
                 else downwardsModify!(Extract!(r,goal),true) fi);
         floor_stop(r,d,f) ==
           if d = up then upwardsExtract!(Extract!(r,f))
                     else downwardsExtract!(Extract!(r,f)) fi;
         cancel_res(r,d,f) ==
            Modify!(r,f, if d = up
                 then upwardsModify!(Extract!(r,f),false)
                 else downwardsModify!(Extract!(r,f),false) fi);
         more_floors(r,d,f,top,bot) ==
            if d=up then
               if f>=top then false else more_floors!(r,d,f+1,top,bot) fi
            else
               if f<=bot then false else more_floors!(r,d,f-1,top,bot) fi
            fi
         more_floors!(r,d,f,top,bot) ==
            upwardsExtract!(Extract!(r,f))
            or downwardsExtract!(Extract!(r,f))
            or if d = up then
                  if f>=top then false else more_floors!(r,d,f+1,top,bot) fi
               else
                  if f<=bot then false else more_floors!(r,d,f-1,top,bot) fi
               fi
   endnewtype Reservations;
   endmacro Declarations;
```

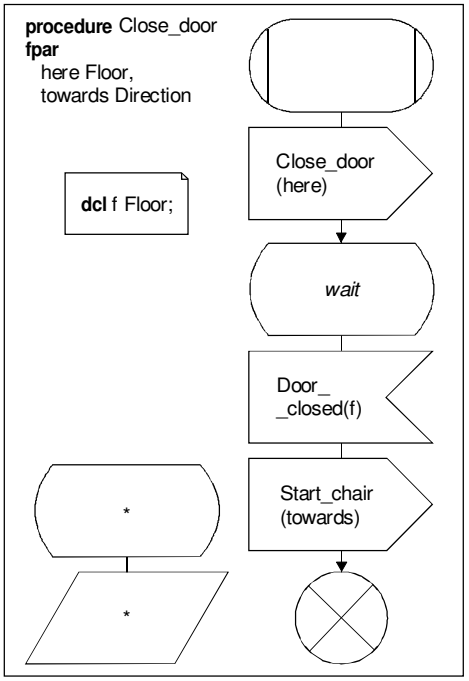



process Control
1(3)
timer
floor_delay := waiting_time
/*timer for staying at one floor*/;
dcl
here Floor, /*current floor*/
moving Direction, /*current direction of lift chair*/
table Reservations, /*table of reservations*/
goal Floor, /*the target floor*/
towards Direction; /*the indicated direction*/
Open_ _door
Close_ _door
here:=bot_floor, moving:=up, table:=empty_serv
Open_door (here)
*
idle
Maint_ _stop
Goto (goal)
Floor_req (towards,goal)
true
goal=here
false
goal=here
true
false
–
table:=goto_req (table,here,goal)
table:=floor_req (table,goal,towards)
–
moving:= if goal>here then up else down fi
moving:= if goal>here then up else down fi
Close_door (here,moving)
Close_door (here,moving)
here:=if moving=up then here+1 else here–1 fi
here:=if moving=up then here+1 else here–1 fi
running
running
T1006370-92/d025


process Control
2(3)
running
Goto
(goal)
Floor_req
(towards,goal)
Emergency
_stop
false
goal=here
true
table:=floor_req
(table,goal,towards)
Alarm
table:=goto_req
(table,here,goal)
table:=floor_req
(table,goal,moving)
–
stopped
–
–
Restart
Arriving_
_at_floor
running
true
floor_stop
(table,moving,here)
table:=cancel_res
(table,moving,here)
false
false
more_floors
(table,moving,here,
top_floor,bot_floor)
set
(floor_delay)
true
Open_door
(here)
here:=if moving=up
then here+1
else here−1 fi
floor_stop
–
T1006380-92/d026







T1006390-92/d027

I.4 Object-Orientation and SDL

I.4.1 Object-oriented analysis

Object-orientation is a perspective on how to structure systems and system descriptions, and it may even suggest some ways of organizing the system development process, but it by no means gives a complete method.

Object-orientation is often associated with programming, graphical user-interfaces and databases. Applied to programming, it is often associated only with reuse of code. It has, however, turned out that an object-oriented approach is suited just as well in the very early phases of a system development process, where the emphasis is on the understanding of the application domain and on the first specification and analysis of an application. Object-orientation in this situation is not primarily applied because of reuse, but because it implies a new and fruitful way of thinking. Two main reasons for applying object-orientation during analysis are:

- Analysis shall lead to a better *understanding* of what the system is supposed to do. What are the customers needs and what are the requirements on the system?

- It is regarded as a benefit that there is *no need to switch paradigm* when going from specification to implementation.

Part of this understanding is to identify the relevant application specific *phenomena*, and to classify them into *concepts*. The phenomena are represented by *objects*, the concepts are represented by *types* of objects. The concepts are organized in concept hierarchies, which contributes to the understanding of similarities and differences between phenomena. Types representing well-proven application specific concepts are candidates for reuse in many different applications. In this respect it is important to have a general view of object-orientation which allows for the definition of concepts with both data and action aspects. The set of application related concepts is of great value in order to allow people involved in the development of the system to understand and reason about the system. The set of application related concepts will also constitute a common knowledge base relevant for the following phases in the system life-cycle (functional specification, design, implementation, and operation/maintenance). System development involves different groups of people, and these concepts will facilitate the communication and interaction between these groups.

When formal functional specifications may be done in an object-oriented way, this will lead to specifications where essential elements survive in the implementation.

- Mirroring elements external to the system by components of the specified system will also be relevant in implementations.

- Identifying key application (area) specific concepts, representing them as types, and representing concepts hierarchies by subtype hierarchies, will lead to type definitions that are worthwhile also in implementations. Subtype hierarchies that are the result of application concept hierarchies will be more stable than subtype hierarchies invented just for the purpose of implementation. Application specific concept hierarchies are often the result of many years of experience by users and experts in the field.

- The structure of the specified system can be made very close to the structure of the implemented system.

The fact that essential parts of the specification will be stable across design to implementation will not only be useful in maintaining the system, but it will also raise the quality of the specification as a *contract* between the supplier of the system and the customer.

The object oriented analysis is illustrated by a semi-realistic example: a *Bank* system. The example is build upon concepts already developed (in connection with an *AccessControl* system, stored in a library). This reflects industrial praxis, where new systems are not developed from scratch, rather they build heavily on existing systems and components.

The analysis may roughly be divided in four phases: *understanding, searching, generalizing* and *specializing*.

Step 1 – Prose description

- Sketch the system to be specified, using prose description and drawings .

In most cases, some prose descriptions are available about the area which you are modelling. There may be requirement specifications from a requester, manuals and informal descriptions of earlier systems or merely informal sketches intended for completely different purposes.

A *bank* is a place where *money* is handled. Money can be handled in many different ways, there is *cash* and there are *cheques*. Money can also be handled without any physical media denoting their value, e.g. by electronic means, internal transactions of the *accounts*. The emphasis is on those parts of the bank where physical representation of money is handled.

There are a number of *counters* in the bank. Some of them (called *cashiers*) have a *clerk*, the others are automatic counters, called *minibanks*.

Some of the *customers* prefer the minibank where they insert their *card* and key in their personal code in order to get access to their account and receive cash money. They may also get other services such as a *printout* of their account balance. The minibank is provided with a *display* which serves as the interaction device to the customer.

Some of the customers prefer the manual cashiers which may be talked to and communicated with in a human fashion. The clerk, however, has his own data display and keyboard. The cashier can perform more services than the automatic minibanks.

This step leads to the very first understanding of what the application area is all about. It helps defining the boundaries of the system, and it gives hints to what concepts should be included in the dictionary (see below).

Step 2 – Dictionary

- Make or obtain a dictionary for the subject area. The dictionary should be kept updated throughout the development.

In this step the understanding is improved by listing the relevant concepts. Some of the concepts can be found in dictionaries, others have to be defined.

The classical notion of a concept is characterized by:

- *Extension*, the collection of phenomena that the concept covers.
- *Intension*, a collection of properties that in some way characterize the phenomena in the extension of the concept.
- *Designation*, the collection of names by which the concept is known.

Representing concepts by types of instances follows this pattern: the instances belong to the extension, the type definition gives the intension, and the type name is the designation.

Nouns in the prose description, representing key concepts of the subject being analysed, are isolated. These are given in italics in **step 1**. In object-oriented terminology these key concepts are candidates for *types*, of which the objects will be generated. During the following analysis steps, the dictionary should be updated and supplemented.

The bank could have the following initial dictionary:

Account	A computer representation of money associated with an owner
Bank	A place where money is handled
Card	Personal identification means
Cash	Physical representations of money guaranteed by the state
Cashier	Counter with clerk
Cheque	Representation of money with various guarantees
Counter	Communication ports with the bank system
Customer	External processes signalling to and receiving money from the bank
Display	Screen to convey written, but volatile messages to customer or clerk
Key	Part of keyboard
Minibank	Automatic counter
Money	The ultimate signals from a bank
Printout	Written message on paper (signal to customer)

Step 3 – Aggregation

- Make a sketch of a typical instance of the system being analyzed. Label the elements of the sketch.

- From the typical instance, describe the "consists-of" relationships between concepts.

When looking at a bank, it can be seen what pieces it consists of and what pieces each of the pieces consists of etc. Aggregation means collecting separate parts into a whole, and is concerned with "consist-of" relationships, which is an important relationship in most analysis paradigms, and so it is for object-orientation as well. But when specifying the "consist-of" relationships, care must be taken to make a distinction between concepts (types) and instances of the types. The avoid confusion, capital initial letters will be used for concepts (e.g. *Bank*) and lower-case letters for instances (e.g. *bank*). Figure I.4-1 shows the "consist-of" relationships for one specific bank instance having four specific instances of counters.

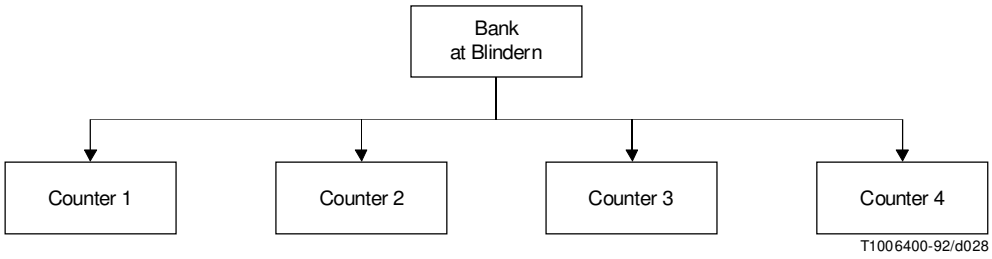


FIGURE I.4-1/Z.100

"Consist-of" relationships between instances

A system analysis is mostly concerned with systems in general, and not so much with specific system instances. The same holds for banks and counters. Accordingly, the "consist-of" relationship between *Bank* and *Counter* in general is of interest (see Figure I.4-2).

In this figure, we have actually relationship types, which are associations between concepts (types of objects). A relationship type has a cardinality, which indicates the maximum number of instances of each type involved. It is also possible to indicate that some instances are optional (dashed arrow), that is they may not be part of some bank instances.

Step 4 – Similarities

- For each concept in the dictionary, ask whether all the objects that fall within the extension of the concept have the same properties. If they have not, find specializations, which may or may not extend the dictionary.

- During the specialization, extend the description in the dictionary with properties which adds to the understanding of the concept.

In the previous step, the relationships between instances have been analysed. This step is concerned with the relationships between concepts.

As has been stated in earlier steps, *Minibank* and *Cashier* are specializations of *Counter*, meaning that all cashiers and minibanks are counters, and whatever general things can be said about counters will be valid both for minibanks and cashiers. We get the specialization hierarchy according to Figure I.4-3.

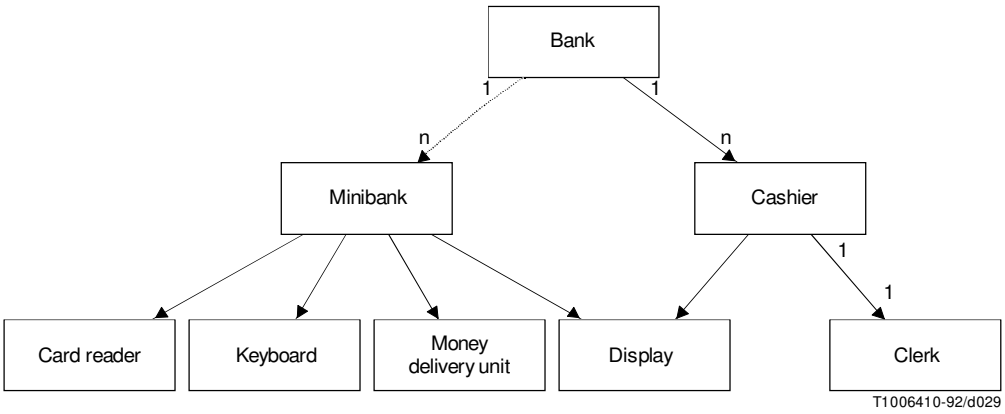


FIGURE I.4-2/Z.100

"Consist-of" relationships between concepts

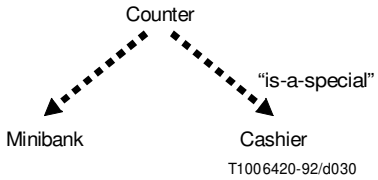


FIGURE I.4-3/Z.100

Specialization hierarchy of counter

When we find such specialization hierarchies, we must go in greater detail into the definitions of what characterizes a *Minibank* or a *Cashier*, and what distinguishes them from each other and from the general notion of a *Counter*. The idea behind such specialization relationships is to describe the general, common aspects only once. The description of *Counter* will comprise all the common features, while the descriptions of *Minibank* will include only the features special to minibanks.

The bank example has more specialization hierarchies, see Figure I.4-4.

Step 5 – Library search

– See if there is a ***Y*** in a library which is similar to an ***X*** in the dictionary. When a similarity is found, either make ***X*** a direct specialization of ***Y***, or restructure the library by creating a ***Z*** which is a generalization of both ***X*** and ***Y***.

Following our thorough understanding of the *Bank*, we want to see whether there are pieces residing in a library which may be used in the design of the *Bank* system. We find that there are many similarities between the *AccessControl* system (see Figure I.4-5) and the *Bank*. A *LocalStation* is quite similar to a *Counter*.

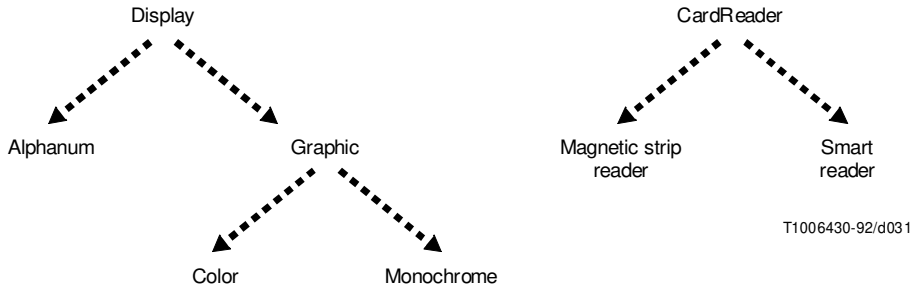


FIGURE I.4-4/Z.100

Specialization hierarchies of Display and CardReader

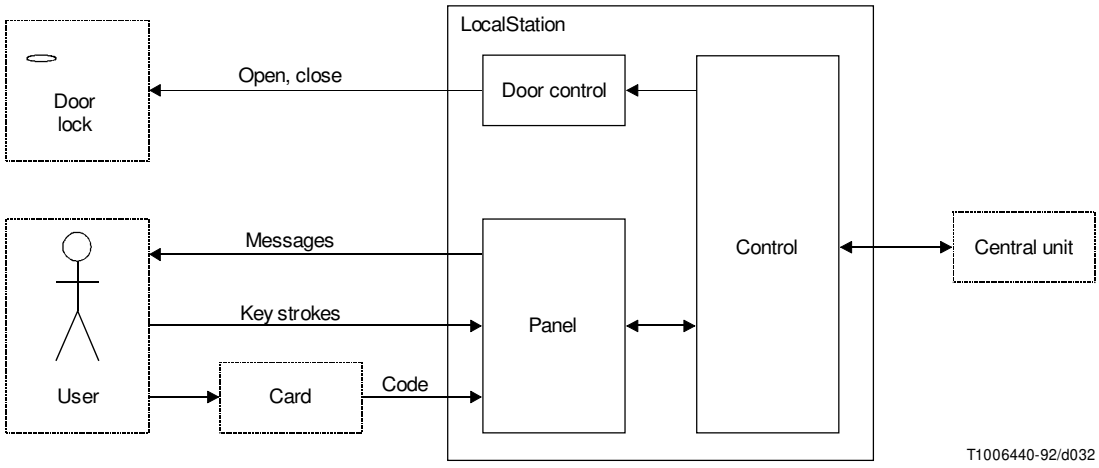


FIGURE I.4-5/Z.100

The AccessControl system

How can we take advantage of the similarities between the *AccessControl* and the *Bank*? There are two different approaches depending on the specialization relationship between the library concept and the application concept: either the application concept is a direct specialization of the library concept, or there is no direct specialization relationship between them, but they are still quite similar.

If the above specialization relationship is the case, then we are quite well off. We will inherit large parts of the description from the library and need only specify additional features (see Figure I.4-6). The potential problem is that even though the specialization relationship seems all right at the first look, it may not hold completely at a closer

investigation. The library units, which were not designed with a bank in mind, may have some internal structures that are not quite what we want for a bank. They may not be “virtual” and then they cannot be redefined. If this is the case, we must conclude that even though the general concepts seemed to be in direct specialization relationship, they were not after all.

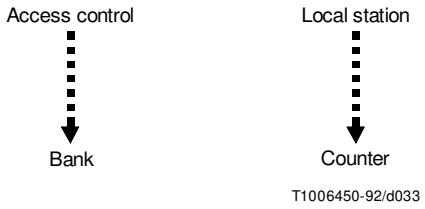


FIGURE I.4-6/Z.100

Direct specialization relationship between application and library

The second case is that we know that *AccessControl* and *Bank* are similar, but they are not in direct specialization relationship. Then they are both specializations of a common, more general concept (see Figure I.4-7).

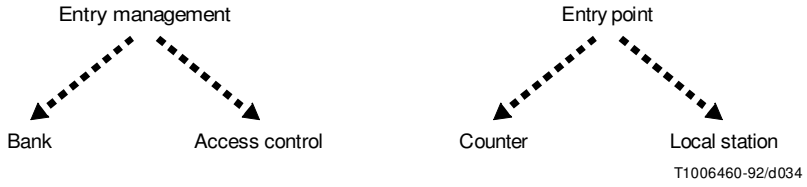


FIGURE I.4-7/Z.100

Indirect specialization relationship between application and library

The analysis now becomes somewhat more comprehensive. The library should be restructured by introducing the new, more general concepts *Entry Management* and *Entry Point*, and then the "old" library units *AccessControl* and *LocalStation* should be made direct specializations of the new library units.

We have now achieved two things. First, the library has become more general, and thus more applicable to new applications other than *AccessControl* and *Bank*. Second, we have created library units of which the *Bank* concepts are direct specializations.

Furthermore, we have not lost anything since the *AccessControl* is in effect identical to what it was before the restructuring. We are both forward and backward compatible!

There are some effects of this library restructuring. We will notice that the more general concepts are, the more virtual types they will include. (This in turn decreases the analysis power, but not necessarily the descriptive power). In making *AccessControl* identical to what it was before (such that former analysis is still valid for its specializations), we must take care to "finalize" the inherited virtual types of the new general concepts which were not virtual in the original *AccessControl*.

Another very typical and very important effect of library search is that when realizing the similarity between *AccessControl* and *Bank*, we can find features of *AccessControl* which may be applied to *Bank*, but which we had not thought of in the *Bank* context. A possible example of this is the *LoggingStation* in *AccessControl*. A logging station for bank transactions may be a good idea!

I.4.2 Applying object-oriented SDL concepts

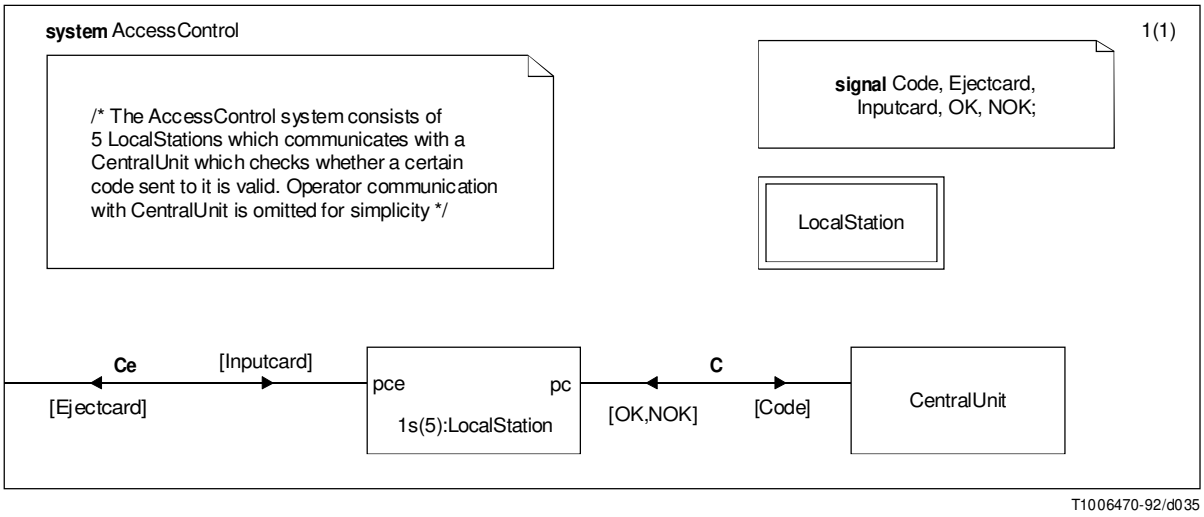
The previous subclause describes a language independent object-oriented analysis. This subclause elaborates on **step 4** and **step 5** by applying object-oriented SDL concepts.

Step 6 – Generalizations

- Achieve flexible components by introducing *virtual types*. Make sure that such types get proper general names. Balance the flexibility of virtual types by using **atleast** to constrain the possibility of redefinitions.

- Achieve independence of *signals* and *sorts* by introducing *context parameters*. Balance this independence by constraining the context parameters.

Our starting point is an *AccessControl* system with some fairly general types (see Example I.4-1).



EXAMPLE I.4-1

The *AccessControl* system diagram

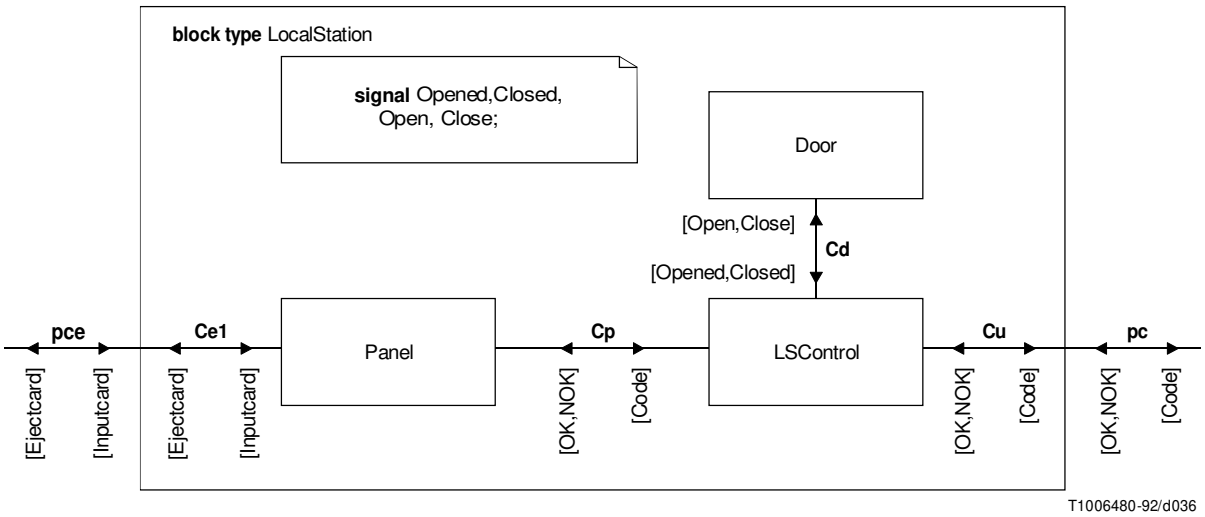
One of the central types is the block type *LocalStation* which (in one version) is defined as in Example I.4-2.

Even though we may be able to see that a *Counter* may be modelled in somewhat the same way, the names of the components are not at all right, and the definition is not flexible enough for our new purpose.

We shall have to generalize in order to have a more general type definition which may be specialized to both *LocalStation* (of the *AccessControl*) and to *Counter* (of the *Bank*). We need flexibility rather than analyzability here.

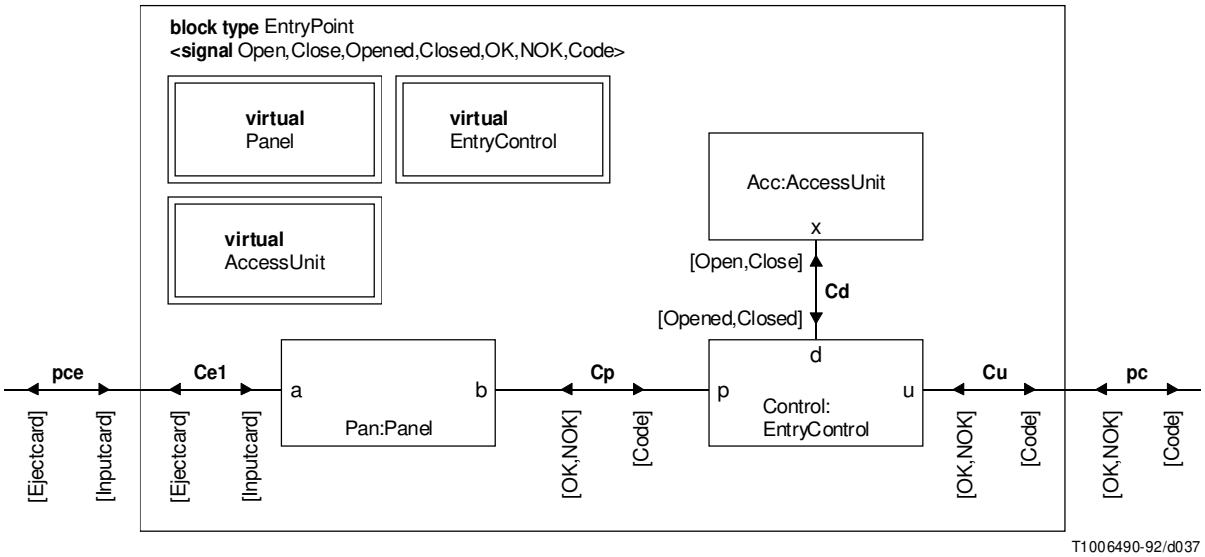
There are two dimensions over which we want to generalize. First, we want to make the components adjustable, i.e. redefineable in specializations. This is achieved by virtual types. Second, we may assume that signals in a *Bank* system are different from those of an *AccessControl* system. Thus, we will want to generalize over signal names. This

is achieved by using context signal parameters (see Example I.4-3). Note that the block type *EntryPoint* is parameterized, and instances cannot be made directly from it.



EXAMPLE I.4-2

LocalStation* of (old) *AccessControl



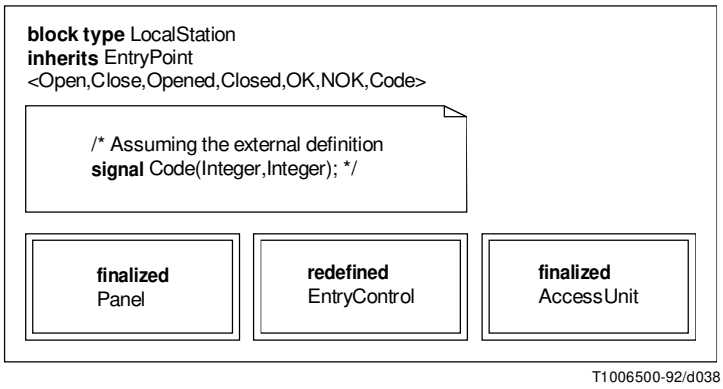
EXAMPLE I.4-3

The general block type *EntryPoint*

Step 7 – Specializations

– In specializing, take care to balance flexibility and analyzability properly using **atleast** and **finalized** to constrain the virtual types.

Using the block type *EntryPoint*, *LocalStation* can be defined as a specialization (see Example I.4-4).

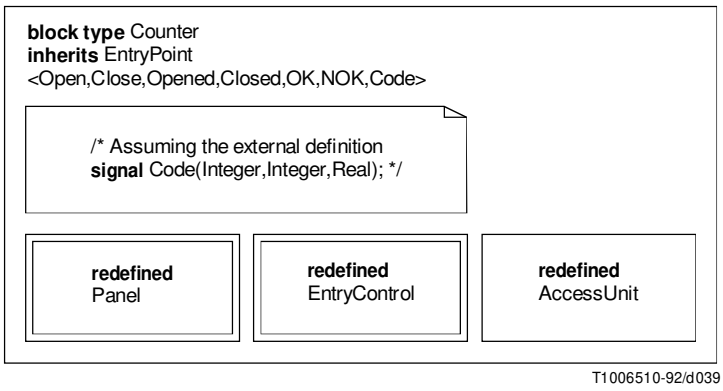


EXAMPLE I.4-4

The (new) block type *LocalStation*

We notice that the *Panel* and the *AccessUnit* are **finalized** in order to restrict the flexibility and increase the analyzability. Furthermore, this is done in this case to make the new *LocalStation* functionally compatible with the old one. We have not shown the system level here, but indicated that on the system level, there is a signal definition of the actual parameters, and especially interesting is the definition of the signal *Code* having two integer parameters.

The bank *Counter* can now be defined analogously (see Example I.4-5).

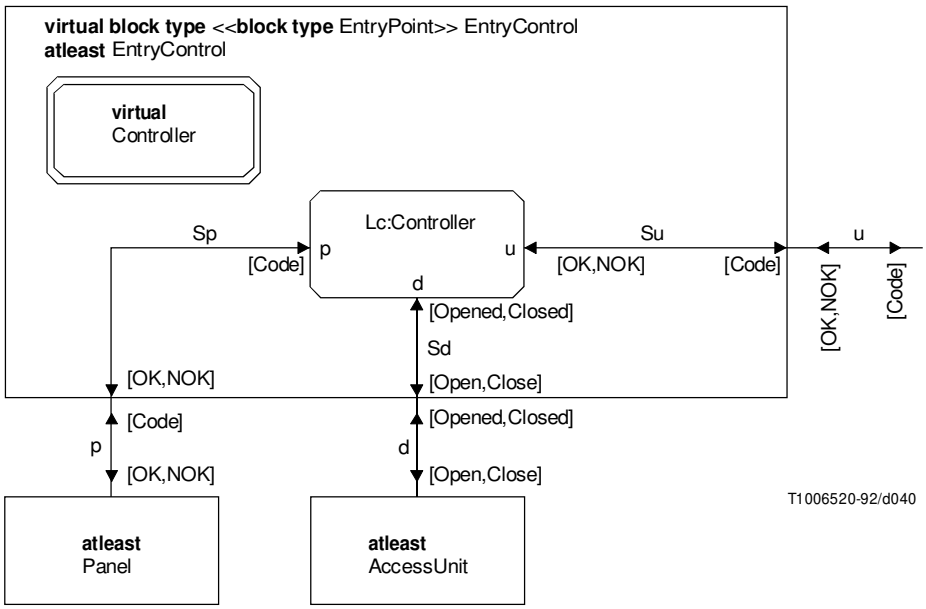


EXAMPLE I.4-5

The block type *Counter*

Note that the *Bank* system is assumed to have a *Code* signal which has three parameters (as opposed to the two parameters of the *AccessControl* system).

Let us now look at the definition of the block type *EntryControl* (see Example I.4-6).

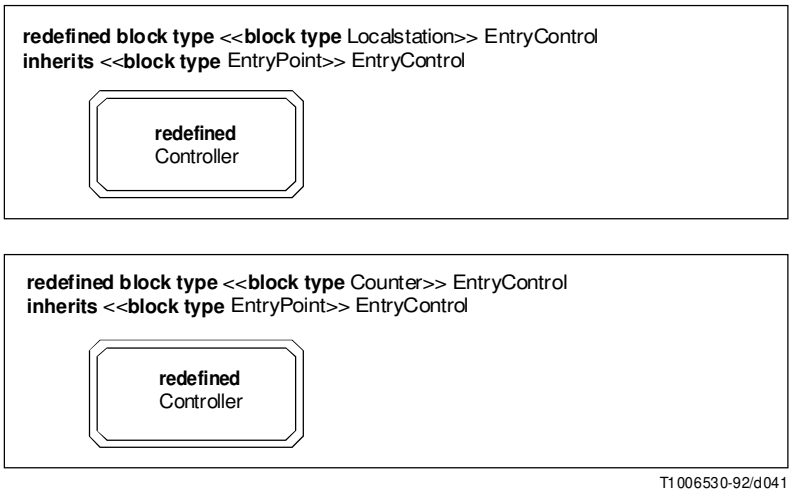


EXAMPLE I.4-6

The block type *EntryControl*

We have here the default definition of the virtual type, and we find that it specifies restrictions by an **atleast**-clause. The restriction is that all redefinitions must be specializations of this one. This block type itself has a virtual process type.

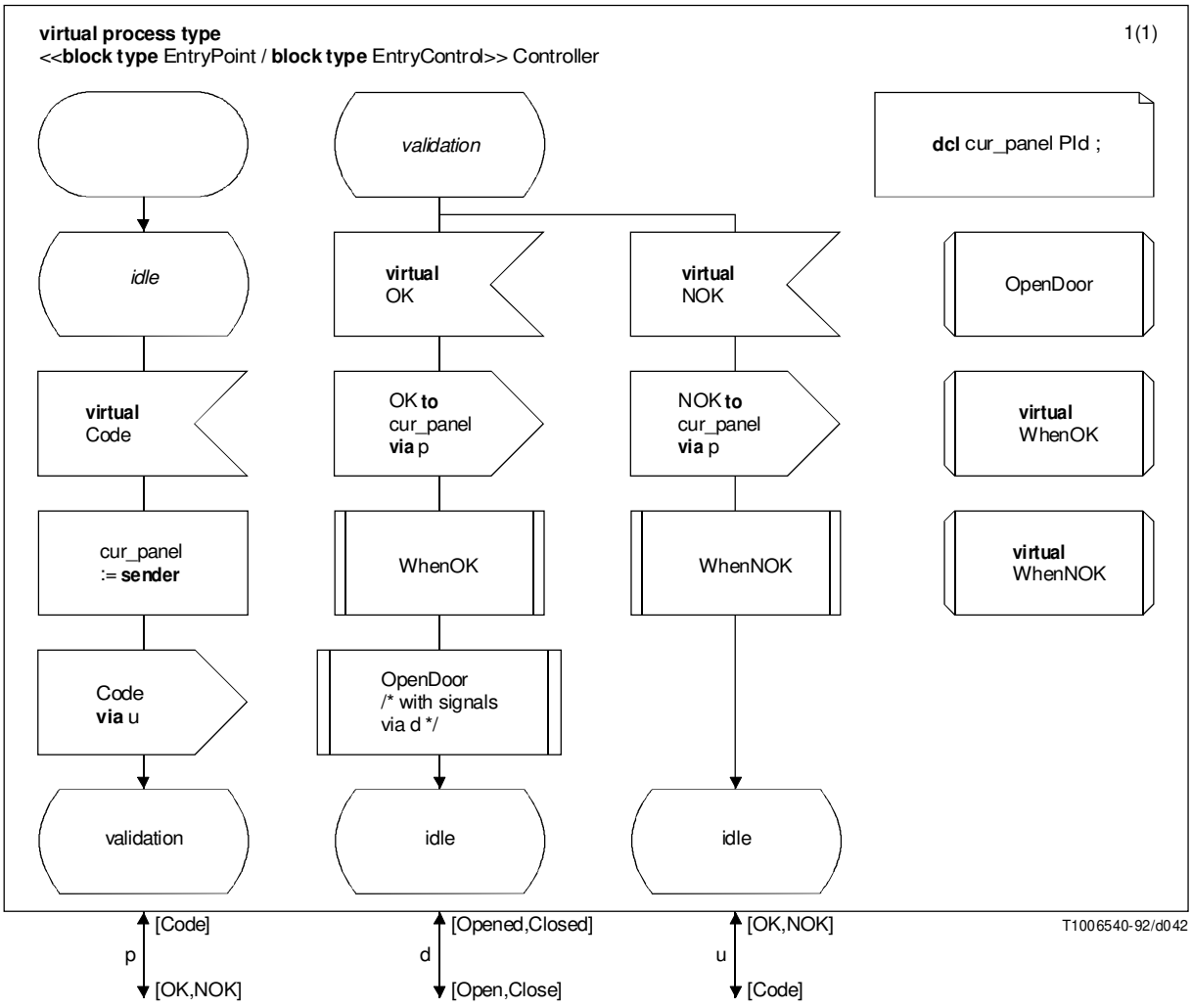
The corresponding redefinitions in the *AccessControl* system and *Bank* system are very simple indeed (see Example I.4-7). Note the lengthy qualifications needed in order to identify the diagrams. This is because the use of virtual types increases the number of name clashes. The users of SDL should not have to pay attention to this problem at all, since this will surely be handled by the tools.



EXAMPLE I.4-7

Redefinitions of *EntryControl*

Let us now look at the definition of the process type *Controller* (see Example I.4-8).



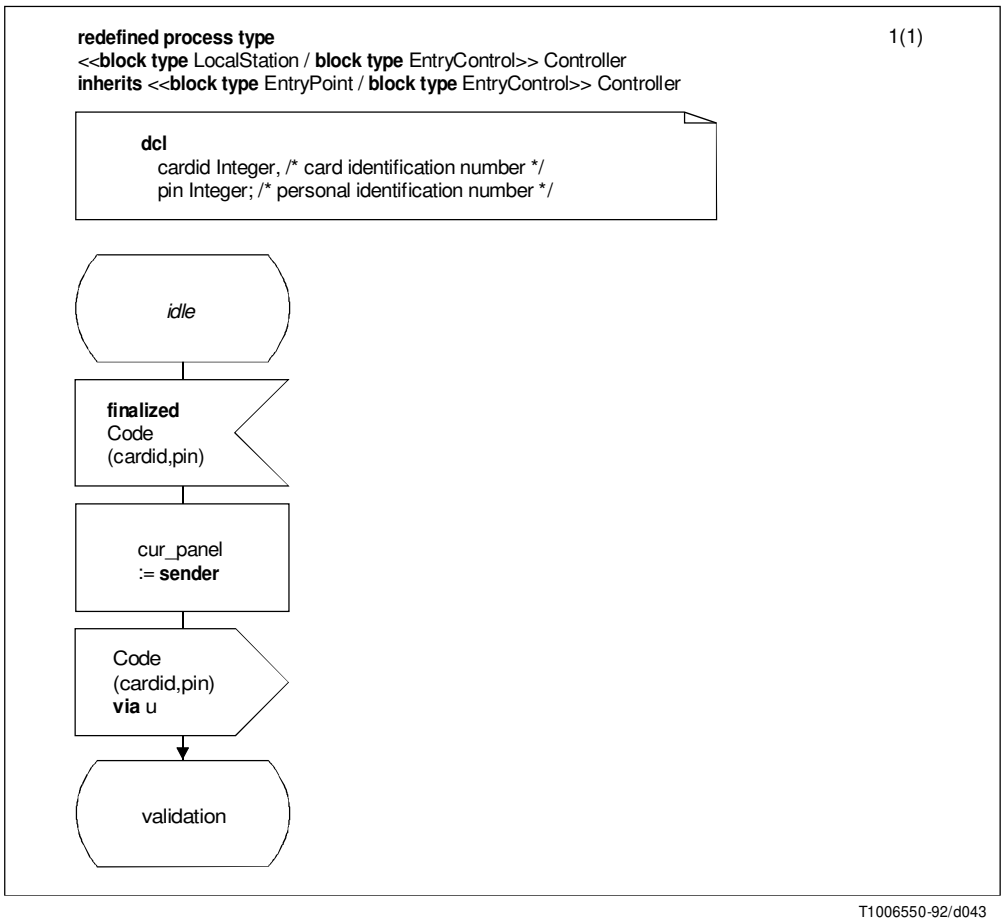
EXAMPLE I.4-8

The process type *Controller*

Again we find that the inner parts of the definition is made flexible by declaring them **virtual**. Here we have made all the transitions virtual. How these are redefined in specializations, is shown in Examples I.4-9 and I.4-10.

The redefinition in Example I.4-9 is concerned with the reception of the *Code* signal. The transition makes use of two signal parameters: *cardid* and *pin*. (The signal definition can be the same, since it is allowed to have fewer parameters in an input than specified in the signal definition, the extra information conveyed by the signal instance will be discarded.)

This redefinition in Example I.4-10 is again concerned with the reception of signals: *Code* and *OK*. The transitions make use of new signal parameters: *cardid*, *pin*, *amount* and *leftonaccount*.



EXAMPLE I.4-9

Redefinition of *Controller* in *AccessControl*

I.5 Stepwise production of a complete ADT specification

For the predefined sorts in this Recommendation it is claimed that these are complete. Why are these definitions complete? To answer this question one has to determine first what completeness is. In this clause the so-called constructor function method (CFM) will be explained and demonstrated. This CFM gives some support for writing complete ADT specifications.

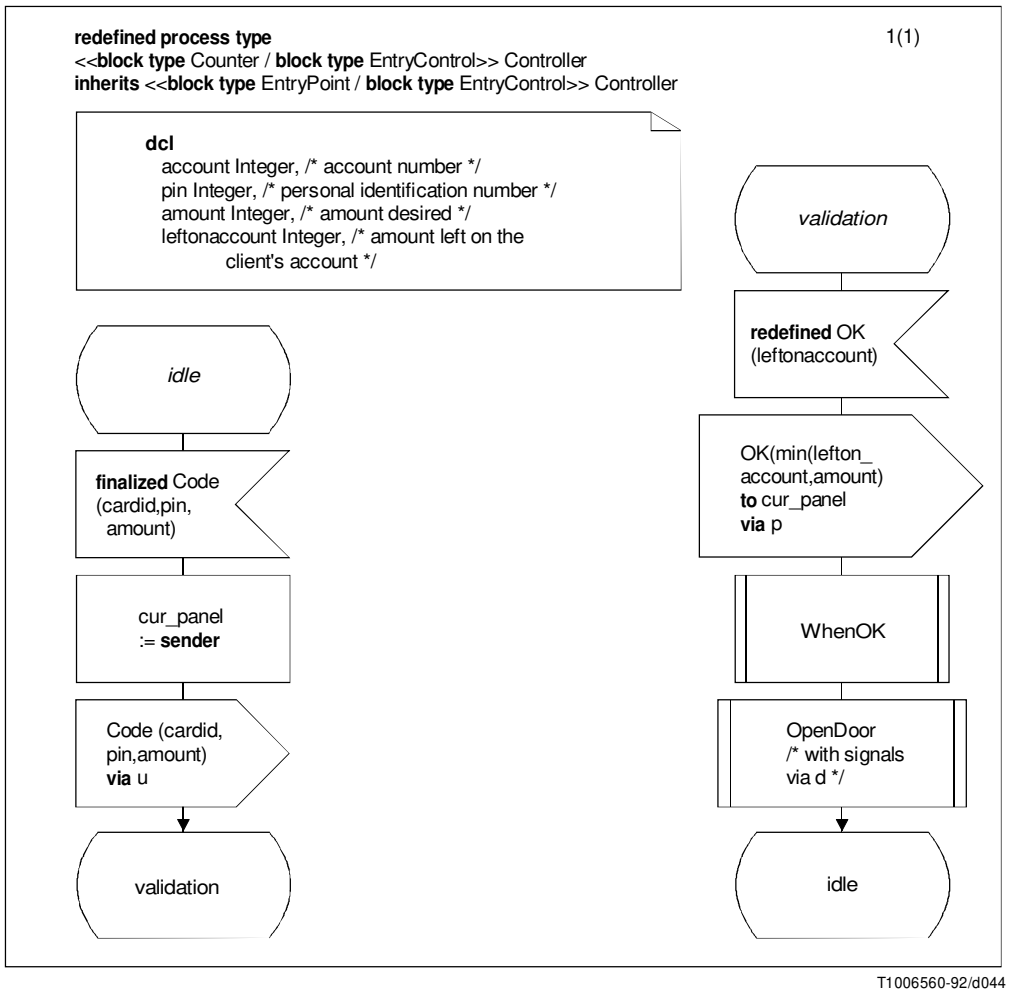
In the sequel it is assumed that the reader is familiar with the mathematical background of ADTs (signatures generate terms divided into equivalence classes by equations; the equivalence classes define the values of the sort).

I.5.1 Completeness of an ADT specification

An absolute definition of completeness can *not* be given. A sort is specified completely when the values of the sort correspond to the values the specifier intended to define. Since the intention of a person is hard to measure, completeness of a sort specification cannot be defined formally[1)].

---

1) In the theory of rewrite systems, “complete” and “completion” has a formal meaning. In this section, however, “complete” is used in the non-formal sense.



EXAMPLE I.4-10

Redefinition of *Controller* in *Bank*

We can do something with the relative definition (relative to the intention of some person) of completeness. To start with, one has to take care that two terms that are intended to represent different values are not put in the same equivalence class. Secondly, care has to be taken that all terms are put in one of the intended equivalence classes (= values), otherwise these terms would become new, unintended values.

The intention of the specification of the sort *Boolean* is to create two values in the sort *Boolean*, such that the two literals belong to different equivalence classes. That means that in the equations one should take care that *true* and *false* are not put in the same equivalence class, and that, besides the equivalence classes to which *true* and *false* belong, no other equivalence classes are created. The equations are written such that an arbitrary *Boolean* expression can be reduced from the inside.

In every expression the innermost sub-expression that is not a literal can, through application of one or more equations, be reduced to a literal. Repeated application of this procedure reduces every *Boolean* expression, regardless of its complexity, to a literal. Thus, it can be concluded that there are at most two values.

It is in principle not possible to show that *true* and *false* are not in the same equivalence class. However, the functions defined in the sort *Boolean* are well-known mathematical functions that behave nicely.

It should be noted that even with well-behaving mathematical functions one has to be careful. One way to define all natural numbers (sort *Nat*) is with the literal *0* and the successor operator *succ*[2]. To express equality of two natural numbers with the operator *eq*, the expressions are "peeled off":

```
succ( x ) = succ( y )   ==   x = y ;
        succ( x ) = 0    ==   false ;
                0 = 0    ==   true ;
                x = y    ==   y = x ;
```

So far no problem. Assume that now an operator *fac: Nat -> Nat* has been defined with equations:

```
   fac( 0 ) == succ( 0 ) ;
fac( succ(n) ) == succ( n ) * fac( n ) ;
```

where $*$ is the mathematical multiplication operator, and it has to be determined whether or not the term *fac(6) = fac(7)* belongs to the same equivalence class as *true*. That requires a lot of equation applications (first a lot of multiplication, which requires even more additions, followed by peeling off *succ fac(6)* times according to the first equation above). A smart specifier therefore adds the equation

```
fac( x ) = fac( y )  ==  x = y ;
```

The consequences are drastic: *true* and *false* are the same value. The proof is as follows:

```
fac( 0 ) = fac(succ( 0 )) == succ( 0 ) = succ( 0 ) == true
```

but, on the other hand

```
fac(0) = fac(succ(0)) == 0 = succ(0) == false
```

and it follows that

```
true == false
```

I.5.2 The basic constructor function method

In this subclause the constructor function method (CFM) is presented. This method is based on a "statement of intention" of the specifier regarding the values in the sorts. In the method, the operators and literals are divided into "constructors" and "functions", and from this division the name of the method has been derived. The terminology "function" and "constructor" have no relation with any SDL construct; these phrases have a meaning in the method only.

In the first two subclauses the definitions of "constructor" and "function" are given. In the third subclause the first four steps of the CFM are presented (i.e. basic CFM).

I.5.2.1 Constructors

When studying ADT specifications, one will note that in general there is a small number of operators[3] that together can generate at least one term in *every intended* equivalence class. A set of operators that "span" a sort are called the "constructors" of that sort.

Example

Sort	Constructors
Boolean	true, false
Integer	0, 1, plus, neg
Tree	empty, leaf, node
IntSet	EmptyIntSet, Insert

The first sentence of this subsection has three essential words:

every

The operator *neg* for *Integer* should not be omitted; without it there would be no term of constructors only (constructor term) in the equivalence class of the term *minus (0, 1)*.

---

2) This is not the way it is done in the predefined sorts.

3) In the sequel, literals are considered as operators without arguments.

intended

The subjectivity remains, but once the constructors have been determined, there is a "statement of intention": all other operators do not generate values that cannot be described by a constructor term.

can

The set of constructors is in general not unique. For the sort *Boolean* one could have selected { *true*, *Not* } or { *false*, *Not* } as constructors.

It is not possible to give much guidance for the selection of the set of constructors. In general, it holds that a small number of constructors is better than a large number. The number of constructors should at least be finite. Usually some experiments are required to find a set of constructors for which the method generates "intuitively attractive" equations. (ADT experts are people who have some experience in selecting a set of constructors that give "nice" equations, the rest is mostly intimidation.)

It does not hold that every value of a sort can be described in a unique way with some constructor term, though that simplifies the use of the method. As an illustration: it is probably the intention that the terms

Insert(1, Insert(0, EmptyIntSet)) and

Insert(0, Insert(1, EmptyIntSet))

represent the same set of integers (i.e. *{0,1}*).

I.5.2.2 Functions

The definition of functions is simple: every operator that is not a constructor is a function. Don't be misled by the name; in the CFM, a literal (and therefore constant) can be a function!

It is important that during the use of the CFM, for all sorts the sets of constructors and functions are not changed. E.g. it is not to be recommended to use the constructors {*true*, *false*} in the specification of the sort *Boolean*, while in the specification of some other sort {*true*, *Not*} are regarded the constructors.

I.5.2.3 The method

In I.5.2.1 it has been indicated that selection of the constructors is a statement of intention: a function (or an expression with several functions) should not generate values that cannot be described with a constructor term. This statement of intention will be exploited.

The following steps should be applied for each function, one function at a time.

Step 1

– Determine for each argument of a function all possible forms that argument can take if only using constructors with only variables as argument(s).

Note that strictly speaking, "value identifier" should be used instead of "variable", since the value of the "variable" does not vary.

Example – (to be referenced in other steps) Assume that there is a function

```
X: Intset, Tree -> IntSet
```

(one could think of the set of integers (result) that occur both in the set of integers (argument) and in a leaf-node of the tree). The first argument of *X* can have one of the following forms:

EmptyIntSet,

Insert(j, s) with *j* of sort *Integer* and *s* of sort *IntSet.*

The second argument can take one of the following forms:

empty,

leaf(j) with *j* of sort *Integer*,

node(b1, b2) with *b1* and *b2* of sort *Tree*.

The expression *Insert(j1, EmptyIntSet)* does not belong in the list of forms for the first argument because the constructor *Insert* has a constructor as argument (i.e. *EmptyIntSet*). For the same reason, *node(b3, leaf(0))* does not qualify as form for the second argument.

Step 2

– Produce a list of applications of the function with for the arguments all possible combinations of forms. Variables in different arguments should have different names (rename if necessary), and all used variables should be placed in a qualification.

Example – For the function *X* in the example above, **step 2** yields the following quantified list of applications:

```
for all  j, j1, j2 in Integer,
         s  in IntSet,
         b1, b2 in Tree
     X( EmptyIntSet, empty )
     X( EmptyIntSet, leaf( j ) )
     X( EmptyIntSet, node( b1, b2 ) )
     X( Insert( j, s ), empty )
     X( Insert( j1, s ), leaf( j2 ) )
     X( Insert( j, s ), node( b1, b2 ) )
```

Note that in the last but one application, *j* has been renamed.

In a variant of the CFM, **step 2** is replaced by:

– Produce one application by giving variables for all arguments of a function.

Then, **step 3** and **step 4** (see below) remain the same, but in particular **step 4b** will be applied more often. The choice between the CFM as presented here and this variant is a matter of taste.

Step 3

– Take each application in the result of **step 2** as the left-hand side of an equation and write the right-hand side using only

1) constructors;
2) variables from the left-hand side;
3) fully defined functions; and
4) the function itself, provided that the expressions for the arguments are smaller or equal than those in the left-hand side, and at least one of the expressions should be strictly smaller.

– If this does not succeed, go to **step 4**.

What is meant by *smaller or equal* and *strictly smaller* is not easy to explain. In general, it means an expression obtained by the elimination of some operators, but that does not always hold. The essential point is that circular definitions must be avoided.

Example – If the interpretation of *X* is as suggested in **step 1** [the set of integers (result) that occur both in the set of integers (argument) and in a leaf-node of the tree], then the following equations can be given. The quantification is omitted for brevity.

```
X( EmptyIntSet, empty )              ==  EmptyIntSet ;
X( EmptyIntSet, leaf( j ) )          ==  EmptyIntSet ;
X( EmptyIntSet, node( b1, b2 ) )     ==  EmptyIntSet ;
X( Insert( j, s ), empty )           ==  EmptyIntSet ;
X( Insert( j1, s ), leaf( j2 ) )     ==  ? ;
X( Insert( j, s ), node( b1, b2 ) )  ==  Union(X(Insert( j, s ), b1),
                                         X(Insert( j, s ), b2 ) ) ;
```

The first four equations do not provide any problem. In the fifth equation, one has to distinguish between the cases where *j1* and *j2* are equal and not equal, and therefore **step 4** is waited for.

The sixth equation uses the previously fully defined function *Union* and the recursive call of *X*. These recursive calls are allowed because the first argument of *X* remains equal, and the second argument is strictly smaller (both with respect to the left-hand side).

Step 4

– There are two possible reasons why the second half of the equation cannot be provided in **step 3**:

1) The right hand side depends upon a relation between variables. Go to **step 4a**.
2) The right hand side depends upon the structure of one or more variables. Go to **step 4b**.

Example – In the case of the fifth equation for *X*, the right-hand side depends upon the relation between the variables *j1* and *j2*.

Step 4a

- In this step, an equation is split in several conditional equations by putting the required relation between the variables in a condition. The left-hand side of the equation obtained in **step 2** is not changed.
- Write the relations between the variables as (non-conditional) equations and/or boolean expressions, using only
  1) constructors;
  2) variables;
  3) fully defined functions; and
  4) the function itself, provided the expressions for the arguments are smaller or equal than those in the left-hand side, and at least one of the expressions should be strictly smaller.
- Check that the conditions for the same application are complementary, or that in case of overlapping conditions the right-hand side is equal.

Example – The required conditions for providing right-hand sides for the application

```
X( Insert( j1, s ), leaf( j2 ) )
```

are

```
j1 = j2
j1 /= j2
```

These conditions are complementary. Therefore, the following equations can be written:

```
j1 = j2 ==> X( Insert( j1, s ), leaf( j2 ) )    ==  Insert( j1, EmptyIntSet )
j1 /= j2 ==> X( Insert( j1, s ), leaf( j2 ) )   ==  X( s, leaf( j2 ) )
```

The first conditional equation satisfies the conditions of **step 3** since the right hand side only uses constructors and a variable. The second conditional equation satisfies the conditions because the recursive call has as its first argument an expression that is strictly smaller, and the second argument did not grow.

Step 4b

- In this step an equation is split in several equations by replacing one of the variables by several terms. Determine all forms of the variable upon which structure the right-hand side depends as in **step 1.** Replace the variable by each of these forms, such that the variable names in these forms do not coincide with variable names that already occur in the application term (i.e. the left-hand side). Add the newly introduced variables in the quantification and complete the equations.

Examples – Assume there is an operator *Fib* : *Nat* –> *Nat* (*Fib* for Fibonaci). For *Nat*, the constructors are assumed to be the literal *0* and *succ* : *Nat* –> *Nat*. Application of the **Steps 1**, **2** and **3** yields

```
for all n  in  Nat
        Fib( 0 )          ==  1 ;
        Fib( succ( n ) )  ==  ? ;
```

Since *Fib* is defined by *Fib(0)=Fib(1)=1* and *Fib(n)=Fib(n-1)+Fib(n-2)* for *n>1*, the second equation for *Fib* depends on the structure of *n*, and **step 4b** applies. The result is

```
for all n  in  Nat
  Fib( succ( 0 ) )           ==  1 ;
  Fib( succ( succ( n ) ) )   ==  plus( Fib( succ( n)), Fib(n ) ) ;
```

The right hand side of the last equation satisfies the conditions of **step 3** because plus is a fully defined function and the arguments are recursive calls of *Fib* with an argument smaller than the argument in the left-hand side.

In some cases, **step 4a** and/or **step 4b** need to be applied more than once.

The above four steps are the complete basic constructor function method. It should be stressed that the basic CFM is safe with respect to completeness, but at the cost of time and paper. Often a group of equations can be replaced by a single equation.

A second issue that should be noted is that the basic CFM does not generate equations where both in the left and in the right-hand side the outermost operator is a constructor. Sometimes such equations are useful, however.

I.5.3 Four additional steps

As stated before, basic CFM is safe, but produces more text than strictly necessary. A good method does not only provide safety, but should also yield a readable result. The steps in this subsection intend to work in that direction.

I.5.3.1 Reduction of equations through extra variables

In the running Example of I.5.2.3, it appears that if the first argument of the function *X* equals the empty set, then the second argument does not influence the right-hand side. That means that the equations

```
X( EmptyIntSet, empty )            ==  EmptyIntSet ;
X( EmptyIntSet, leaf( j ) )        ==  EmptyIntSet ;
X( EmptyIntSet, node( b1, b2 ) )   ==  EmptyIntSet ;
```

can be summarised in

```
X( EmptyIntSet, b ) ==  EmptyIntSet ;
```

where *b* is a variable of sort *Tree*.

Step 5

– Replace a set of equations with an invariant right-hand side and with left-hand sides having a varying argument by one equation in which a variable is substituted for the varying argument.

Merging **step 5** and **step 2** is possible, but usually only at the cost of safety. If not all forms can be replaced by a variable, one can end up with e.g.

```
X( EmptyIntSet, leaf( j ) )  ==  Insert( j, EmptyIntSet) ;
X( EmptyIntSet, b )          ==  EmptyIntSet ;
```

In this case one can, by substituting *leaf(j)* for *b*, obtain the equivalence

```
Insert( j, EmptyIntSet ) ==  EmptyIntSet
```

This can be interpreted as “Insert is a function”, which contradicts the “statement of intention”. That makes sense, because the above equivalence means that there is only one value in *IntSet*.

I.5.3.2 Reduction of equations through commutativity

If there is a function *f* which is commutative in a pair of arguments, then the number of equations can be reduced.

Step 6

– If there is a function *f* for which the equation

f(... ,x,... ,y,...) == f(... ,y,... ,x,...) ;

(commutativity) holds for some argument *x* and *y*, then

1) add the commutativity equation,

2) search for pairs of equations where one left-hand side matches the left-hand side of the commutativity equation and another left-hand side matches the right-hand side of the commutativity equation, and delete one equation from each pair.

Example – Assume that an operator *eq* : *Tree*, *Tree* –> *Boolean* has to be defined for the user defined equality between trees. After **step 2** there are the following applications:

```
eq( empty, empty )                    (1)
eq( empty, leaf( j ) )                (2)
eq( empty, node( b1, b2 ) )           (3)
eq( leaf( i ), empty )                (4)
eq( leaf( i ), leaf( j ) )            (5)
eq( leaf( i ), node( b1, b2 ) )       (6)
eq( node( b1, b2 ), empty )           (7)
eq( node( b1, b2 ), leaf( j ) )       (8)
eq( node( b1, b2 ), node( b3, b4 ) )  (9)
```

After **step 3** it appears that *eq(t1, t2) == eq(t2, t1)* and this equation is inserted in **step 6** while the equations 4, 7 and 8 are deleted, as they form pairs with 2, 3 and 6 respectively.

Step 6 can be combined with **step 2**, but again only at the cost of some safety.

I.5.3.3 Reduction of equations through extra functions

If we have one more look at the equations for the function *X* in *IntSet*, it can be noted that its first argument often occurs in the right-hand side without change. This suggests a possible change to a situation where first some function operates on the second argument before its result is combined with the first argument. To be more concrete: for *X* we look for functions *X1* and *X2* such that

```
X( s, b ) = X1( s, X2( b ) ) ;
```

In this case, the functions *Intersection* : *Intset*, *IntSet* -> *Intset* for *X1* and *Projection* : *Tree* -> *IntSet* for *X2* would be suitable. Assuming that *Intersection* is specified anyhow, this split reduces the number of equations from seven (for *X*) to four (one for *X* and three for *Projection*).

Step 7

- If there is a function *f* where one or more arguments occur (almost) unchanged in the right-hand sides, then try to find new functions *f1*,..., *fn*, *C* such that
  1) *f1*,..., *fn* operate on (subsets of) the changing arguments of *f*,
  2) *f* can be expressed in *C* applied to the non-changing arguments of *f* and the results of *f1*,..., *fn*.
- New functions that are "artificial" can be hidden for the users of the ADT by using an exclamation mark as the last character in the function names.

Example – See above.

I.5.3.4 Combination of conditional equations

In many cases there are two conditional equations for an application, and the conditions are a boolean expression and its negation. These can be combined using a conditional term on the right-hand side.

Step 8

- If for some application conditional equations are used, and there are two conditional equations of the format

  c ==> lhs == rhs1 ;

  not(*c) ==> lhs == rhs2 ;*

then these conditional equations are replaced by the non-conditional equation:

lhs == ***if*** *c* ***then*** *rhs1* ***else*** *rhs2* ***fi***;

The conditional term on the right-hand side of an equation should not be used in other circumstances than those described in **step 8.** The reason is that one tends to program in equations, often with nested conditional terms, which makes the equations hard to read/understand.

I.5.4 Equations for constructors

With the basic CFM no equations can be generated with a constructor as the outermost operator (so-called constructor equations) on both sides of the == sign. However, in some cases there is a need to express the equivalence between constructor terms.

One example is the sort *Integer* with the constructors *0* (literal), *succ*, *neg* : *Integer* –> *Integer*. This allows for terms like *succ(neg(...))*. It is, however, possible to put a constructor term in each equivalence class if *neg* is used at most once as the outermost operator. This fact can even be used to simplify some equations for functions.

Step 9

- For those constructors *C* for which one wants to write constructor equations, a function C^f is created. The CFM without this **step 9** is applied to these functions. Then the ^{f}-markings are deleted, and the equations with the same left and right-hand side are deleted.

The result of **step 9** is safe, but usually superfluous (but harmless) equations are generated.

Step 9 is presented as the last step in the CFM since starting with marking constructors as functions would be rather confusing. However, once the reader is a little familiar with the CFM, it is recommended to produce the equations for the constructors before the definition of the functions. The reason is that the equations for the constructors give a good insight in the structure of constructor terms.

Example – For the *Integer*, constructor equations for *neg* and *succ* make sense. Normal application of the CFM yields:

```
neg^f( 0 )                  ==  0 ;
neg^f( succ( n ) )          ==  neg( succ( n ) ) ;
neg^f( neg( n ) )           ==  n ;
neg^f( 0 )                  == succ( 0 ) ;
succ^f( succ( n ) )         ==  succ( succ( n ) ) ;
succ^f( neg( 0 ) )          ==  succ( 0 ) ;
succ^f( neg( succ( n ) ) )  ==  neg( n ) ;
succ^f( neg( neg( n ) ) )   ==  succ( n ) ;
```

The other sub-steps of **step 9** yield:

```
neg( 0 )                    ==  0 ;
neg( neg( n ) )             ==  n ;
succ( neg( 0 ) )            ==  succ( 0 ) ;
succ( neg( succ( n ) ) )    ==  neg( n ) ;
succ( neg( neg( n ) ) )     ==  succ( n ) ;
```

The third and fifth equation in the last list are superfluous since they are implied by the first and second equation respectively.

I.5.5 Limitations

The presentation of the CFM in these guidelines starts at the point when it is decided to build a new sort from scratch. The method gives no guideline on when to do this, and when to base a sort specification on predefined sorts plus the *Structure* sort constructor. That is simply outside the scope of the CFM.

Within the scope of the CFM would be the use of e.g. **error**! and **nameclass**, but this is not the case presently. A more serious point is that the presented CFM is based on the theory for reductive rewrite systems, see [6]. This is rather restrictive. Consider for instance the function *Qsort* : *IntList* –> *IntList* (quicksort) with the equations

```
Qsort( EmptyIntList ) == EmptyIntList ;
split( il, i ) == pair( l1, l2 ) ==>
   Qsort( MkString( i ) // il ) == Qsort( l1 ) // MkString(i ) // Qsort( l2 );
```

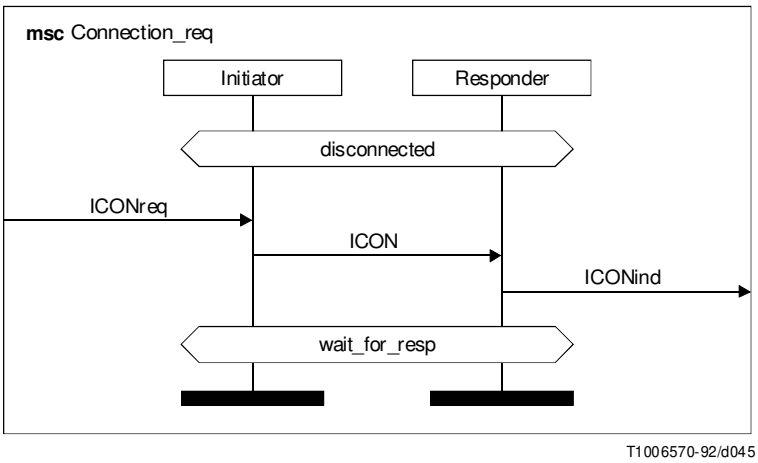
where *pair* is a constructor for pairs of *IntLists*, and *split* splits its first argument in a list with integers smaller than its second argument and a list with integers larger than its second argument.

This definition of *Qsort* would be perfectly right in the theory of quasi-reductive rewrite systems, but with the CFM, this kind of “elegant” equations will be probably never produced.

I.6 Using message sequence charts

I.6.1 Introduction

Within the system life cycle, increasing attention is paid to the stage of system specification, since the quality of all following stages is dependant on it. In particular, in the field of telecommunication, this has been taken into account by the use of the language SDL. In addition to a general proof of correctness (e.g. absence of deadlocks), the consistency of the SDL specification with respect to the required system behaviour has to be checked. A convenient way to describe the required system behaviour is offered by system traces which are suitably presented in form of message sequence charts (MSCs). MSCs are widespread means for the description, and particularly graphical visualization, of selected system traces within distributed systems, especially telecommunication systems. A MSC shows sequences of messages interchanged between entities (such as SDL services, processes, blocks) and their environment (see Example I.6-1). Formally, a MSC describes the partial ordering of events, i.e. message sending and consumption, see [5].



EXAMPLE I.6-1

A message sequence chart

Since each sequentialization of a MSC describes a system trace, a MSC can be derived from an existing SDL system specification. However, a MSC is generally created before the system specification, and then serves as

- a statement of requirements for SDL specifications;
- a basis for automatic generation of skeleton SDL specifications;
- a basis for selection and specification of test cases;
- a semi-formal specification of communication; or
- an interface specification.

The MSC in Example I.6-1 describes a selected piece of trace of the connection set-up in the INRES service specification, see [12]. It could equally be represented using SDL process diagrams with certain additions or modification (see Figure I.6-1), where untraversed branches are dashed and the signal flow is indicated by bold arrows.

The diagram in Figure I.6-1 contains at least the same information as the MSC in Example I.6-1. However, the MSC obviously is more useful in this context, since it concentrates on the relevant information, namely the entities (*Initiator*, *Responder*) and the signals involved in the selected piece of trace (*ICONreq*, *ICON*, *ICONind*). In addition, a MSC can describe the interaction of entities belonging to different levels of abstraction.

A comparison of Example I.6-1 and Figure I.6-1 gives a good intuitive idea about the meaning of a MSC. It also demonstrates that a MSC describing one possible scenario can also be looked at as a skeleton SDL specification.

The most basic language constructs of MSCs are *instances* and *messages* exchanged between them. In the graphical form, the instances and environments are represented by vertical lines or alternatively by columns. Messages are presented by horizontal directed lines with a possible bend to admit message overtaking or crossing. The message arrow denotes signal consumption, the opposite end (message origin) the signal sending.

One MSC in general describes a small part of a complete system trace. Therefore, the MSC has to be characterized by the specification of its initial and final *conditions* and possibly by intermediate conditions. Graphically, a condition is represented by a hexagon containing the condition name (see the initial condition *disconnected* in Example I.6-1). In addition, a MSC can contain *actions* triggered by signal consumption and *timeouts*.

I.6.2 System specification using MSC-composition mechanisms

MSCs are used mainly as a requirement language to describe the purpose of a system in form of trace examples. In the sequel, a systematic MSC methodology based on composition mechanisms is presented. The conditions introduced for this purpose may be employed also for the derivation of skeleton SDL specifications.

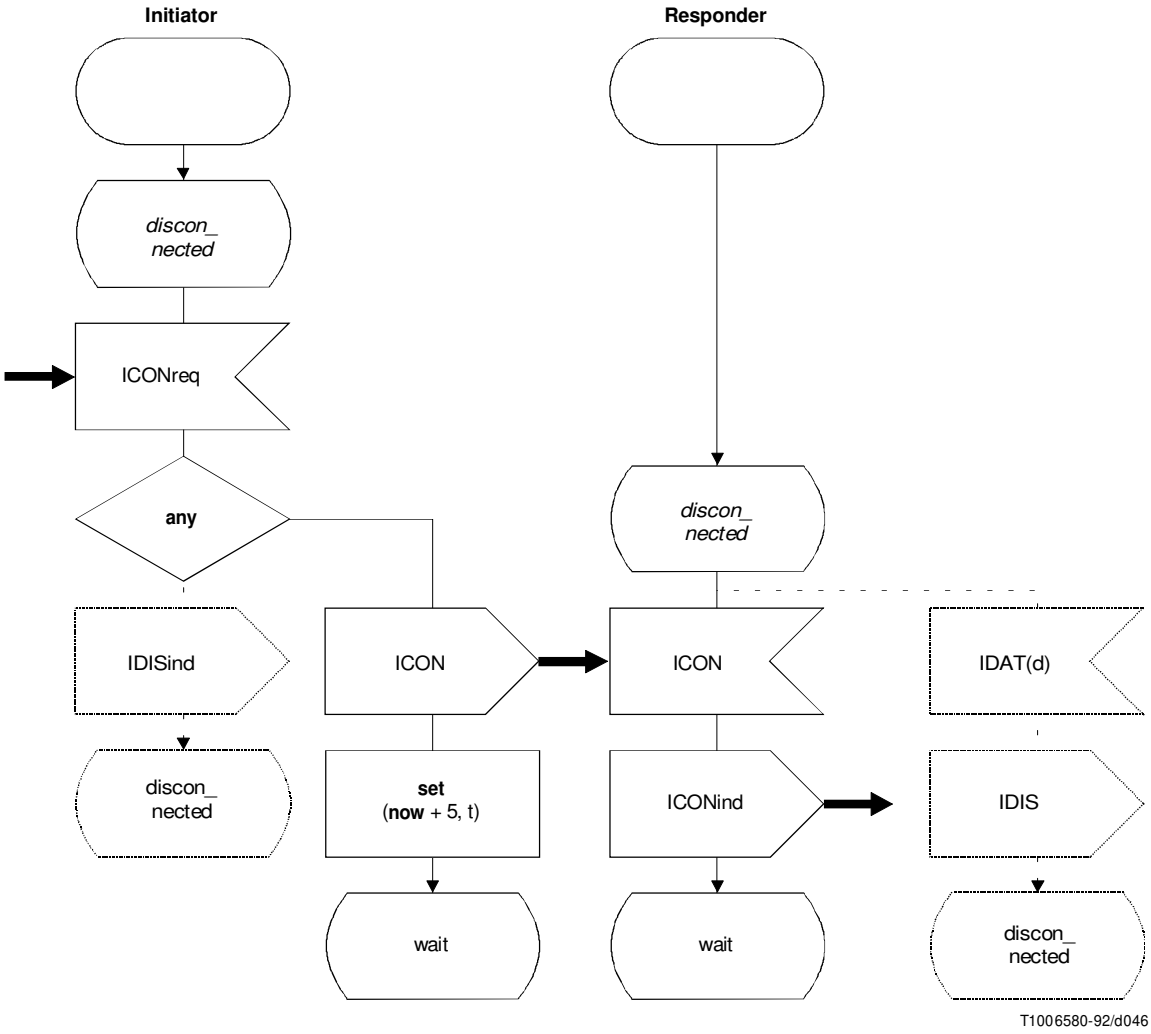


FIGURE I.6-1/Z.100

Combined SDL – Signal flow diagram (informal)

Since one MSC only describes a partial system behaviour, it is advantageous to have a number of simple MSCs that can be combined in different ways. (This can be actually carried out, which is called here composition, or seen only as an interpretation order.) MSCs can be composed by (name)-identification of final and initial conditions. The other way round, MSCs can be decomposed at intermediate conditions.

Composition and decomposition of MSCs follow the subsequent rules for global and non-global conditions, whereby global conditions refer to all instances involved in the MSC, whereas non-global conditions are attached to a subset of instances.

I.6.2.1 Composition of MSCs

Global conditions

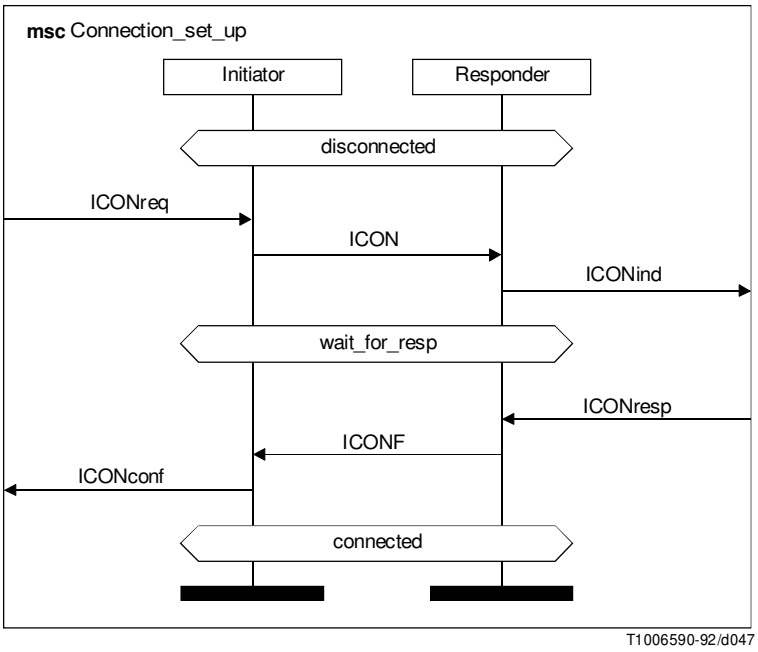
Two message sequence charts, *MSC1* and *MSC2*, can be composed if both message sequence charts contain the same set of instances and if the initial condition of *MSC2* corresponds to the final condition of *MSC1* according to name identification. The final condition of *MSC1* and the initial condition of *MSC2* become an intermediate condition within the composed MSC. Symbolically:

MSC1 = MSC1' Condition
MSC2 = Condition MSC2'

MSC1 * MSC2 = MSC1' Condition MSC2'

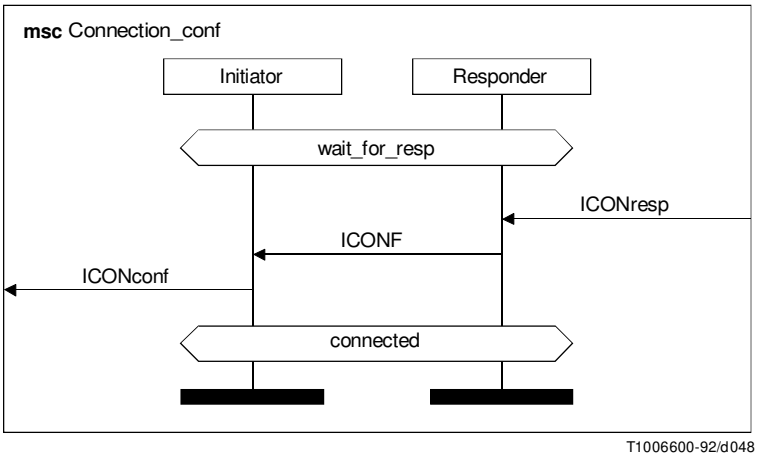
The first equation means that *MSC1* can be written as a MSC-section *MSC1'* and a succeeding final condition. The second equation denotes that *MSC2* starts with an initial condition which is followed by the MSC-section *MSC2'*. The third equation denotes the composition of *MSC1* and *MSC2* (using the asterisk symbol for composition). The composed MSC can be written in form of a starting MSC-section *MSC1'*, an intermediate condition and a succeeding MSC-section *MSC2'*.

The MSC *Connection_set_up* in Example I.6-2 is a composition of the MSCs *Connection_req* in Example I.6-1 and *Connection_conf* in Example I.6-3.



EXAMPLE I.6-2

A composite MSC, based on Examples I.6-1 and I.6-3



EXAMPLE I.6-3

MSC Connection_conf

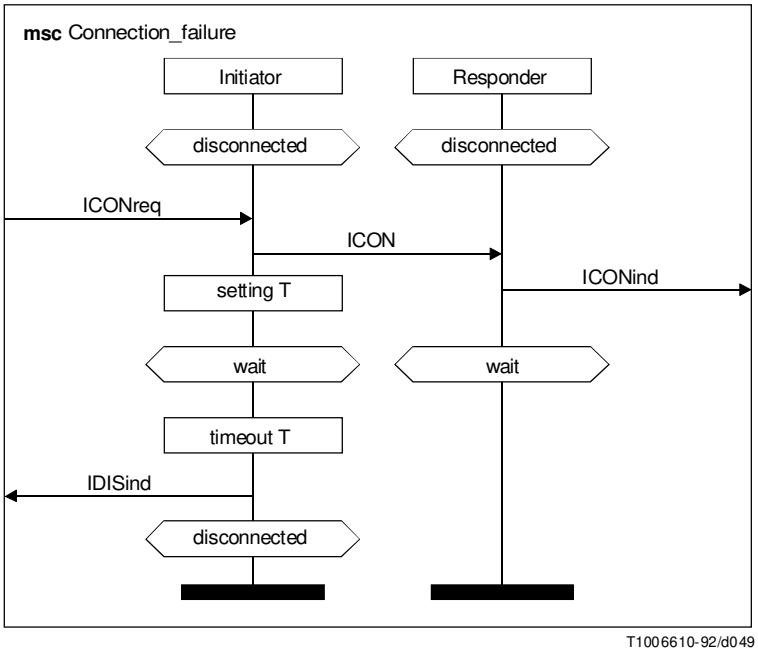
Non-global conditions

Two message sequence charts *MSC1* and *MSC2* can be composed by means of non-global conditions if for each instance (*I*) which both MSCs have in common *MSC1* ends with a non-global condition and *MSC2* begins with a corresponding non-global condition. In addition, each non-global condition of *MSC2* must have a corresponding non-global condition in *MSC1*. If *I(MSCi) (i = 1,2)* denotes the restriction of an *MSCi* to the events of instance *I*, this can be written symbolically:

I(MSC1) = I(MSC1)' Condition
I(MSC2) = Condition I(MSC2)'

I(MSC1) * I(MSC2) = I(MSC1)' Condition I(MSC2)'

For example, MSC *Connection_failure* (see Example I.6-4) is a composition of MSC *Connection _request* (see Example I.6-5a) and *Timeout* (see Example I.6-5c) via the local condition *wait*. The MSC *Connection_request* contains two instances: *Initiator* and *Responder*. To each of the instances a local (final) condition *wait* is attached. It should be noted that the two local conditions *wait* with identical names are different and are discriminated by the instances to which they are attached. MSC *Timeout* contains only one instance, *Initiator*, to which the initial local condition *wait* is attached. The composition of MSC *Connection_request* with MSC *Timeout* only refers to the instance *Initiator*, i.e. MSC *Connection_request* is continued along instance *Initiator* by MSC *Timeout*. This also shows the usefulness of non-global conditions, which makes a composition with respect to a subset of the instances involved in the MSCs possible. Note also that in Example I.6-4 the setting of timer *T* and the timeout are indicated as actions, for compatibility with Examples I.6-5a and I.6-5c.



EXAMPLE I.6-4

MSC Connection_failure

I.6.2.2 Decomposition of MSCs

Global conditions

An intermediate condition makes it possible to decompose a MSC, *MSC3′*, by splitting it at the intermediate condition into *MSC1* and *MSC2*, the intermediate condition being converted into a final condition for *MSC1* and an initial condition for *MSC2*:

MSC3 = MSC1' Condition MSC2'

MSC1 = MSC1' Condition
MSC2 = Condition MSC2'

Non-global conditions

A subset of intermediate nonglobal conditions allows a decomposition of an MSC, *MSC3′*, into *MSC1* and *MSC2*, if all non-global conditions of this subset refer to different instances and no message is cut into pieces by means of the decomposition, i.e. both message input and the corresponding output belong to either *MSC1* or *MSC2*:

I(MSC3) = I(MSC1)' Condition I(MSC2)'

I(MSC1) = I(MSC1)' Condition
I(MSC2) = Condition I(MSC2)'

For example, the MSC *Connection_failure* in Example I.6-4 can be decomposed into MSC *Connection_request* in Example I.6-5a and *Timeout* in Example I.6-5c at the local condition *wait*.

I.6.2.3 Normalizing message sequence charts

A set of MSCs developed in an early stage of design generally is rather unstructured. It is difficult to have a good overview and to estimate the coverage of all possible system traces. Also, large MSCs often have many cyclic subtraces in common, which could be avoided by an appropriate decomposition rule.

In order to overcome these shortcomings, so called *normalized MSCs* can be constructed out of a given set of MSCs applying decomposition and composition rules defined below. These normalized MSCs represent “standardized” building blocks. They describe the same system behaviour as the original set of MSCs from which they are obtained, i.e. the same system traces can be derived from them, taking into account the composition rules. In this sense, the derived set of normalized MSCs is equivalent to the original set of MSCs.

Normalized building blocks and a corresponding structured composition method have been suggested already for Petri net processes by means of so called *process periods*.

Normalized MSCs are essentially maximal message sequence charts which are inseparable, i.e. they always appear as a whole without cyclic insertions. Maximal in this context means that no bigger MSC with the listed properties exists which contains the MSC as a subtrace. Normalized MSCs describe either sequential traces or cyclic traces which do not contain cyclic subtraces emerging from intermediate states.

Normalized MSCs are defined in such a way that it is possible to generate them automatically from a given set of MSCs at each stage of system specification. Thus, normalized MSCs represent building blocks which may be used separately and in parallel with the originally specified set of MSCs. It is recommended to specify the MSCs already from the very beginning in a structured manner obeying the rules of normalized MSCs as far as possible. In practice, however, such an approach may impose unnecessary and inconvenient restrictions on the user, since the idea of normalized MSCs refers to the system as a whole, whereas in early stages usually only parts of the system are specified.

The computer aided composition of normalized MSCs offers the possibility of an immediate simulation of the specified behaviour. Furthermore, normalized MSCs may be used for test case generation and test case selection. Viewing normalized MSCs as the smallest units admits the abstraction from details of communication and the analysis of the essential behaviour of the system. Normalized MSCs are interesting also for system analysis providing an alternative to the reachability graph, since they represent the concurrency aspects of the system in a direct manner.

The procedure for normalizing MSCs is given by the following schematic steps, assuming a distinguished initial state for the system.

Step 1 – Initial set of MSCs

– Selected MSCs are specified for a system within the stage of requirement definition.The relations between these MSCs are specified by means of conditions which allow corresponding compositions and decompositions.

In order to illustrate the steps, we use the requirement specification for the INRES-Service, see [12], which shows a (simplified) connection set up and a following data transfer between *Initiator* and *Responder*. Due to an unreliable medium, the signal transfer may fail. For practical reason, the chosen initial set of MSCs is identical to the set of atomic MSCs defined in **step 2**.

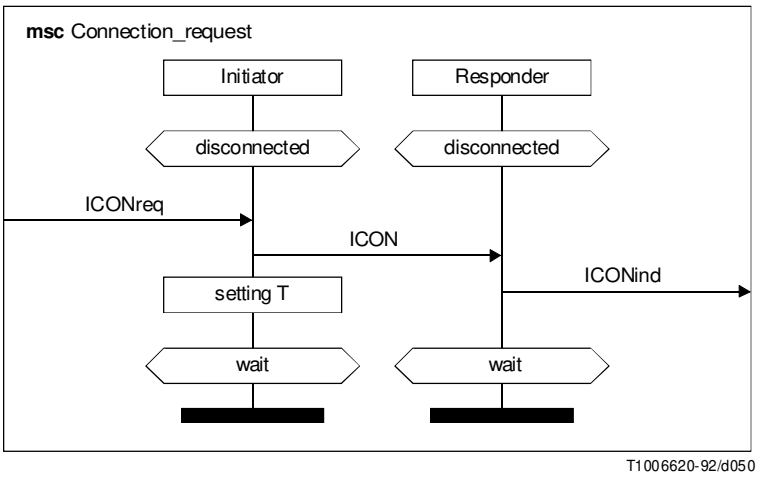
Step 2 – Decomposition into atomic MSCs

– Decompose the given MSCs into "atomic" MSCs, i.e. MSCs which do not contain intermediate conditions.

The atomic MSCs in Examples I.6-5a to I.6-5g only show the non-failure cases with respect to the medium, i.e. cases where no signals are lost. The timeout situation in Example I.6-5c is due to the fact that the responder is not answering in time and not due to loss of signals.

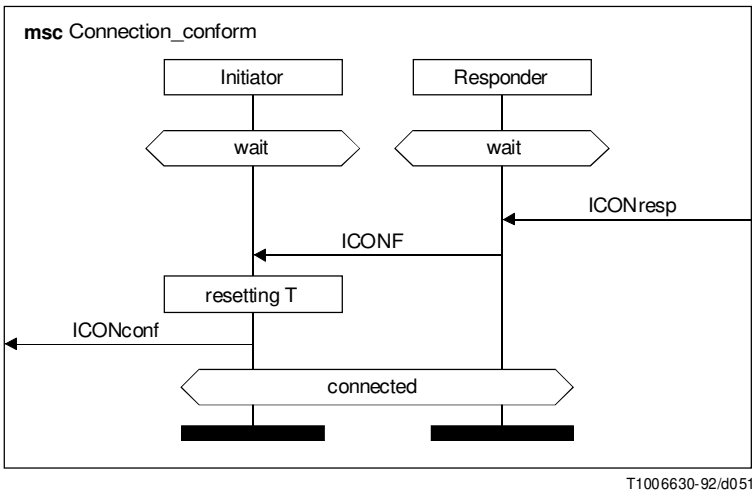
The MSCs are closely related to the time sequence diagrams used within the OSI Basic Reference Model. However, MSCs contain more information: besides actions and timeouts, they also contain *conditions* which provide the basis for composition mechanisms.

Both local and global conditions are used in the examples. Local conditions are employed to facilitate the continuation of one MSC by another with respect to a subset of instances. For example, the MSC *Connection_request* in Example I.6-5a can be continued by the MSC *Timeout* in Example I.6-5c with respect to the instance *Initiator*. Global conditions, e.g. *connected*, are used for a global composition, i.e. a composition with respect to all contained instances. Note that the setting of timer *T* and the corresponding resetting and timeout are in different MSCs, and must therefore be indicated as actions.



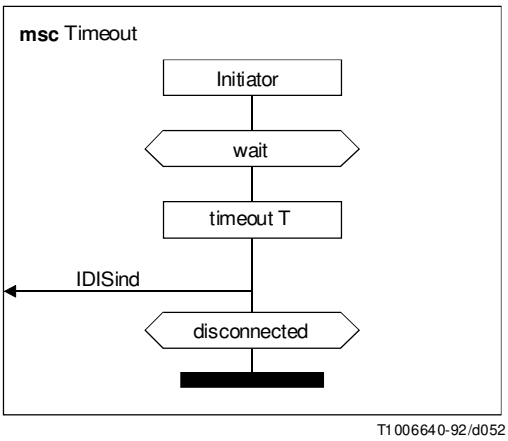
EXAMPLE I.6-5a

Atomic MSC



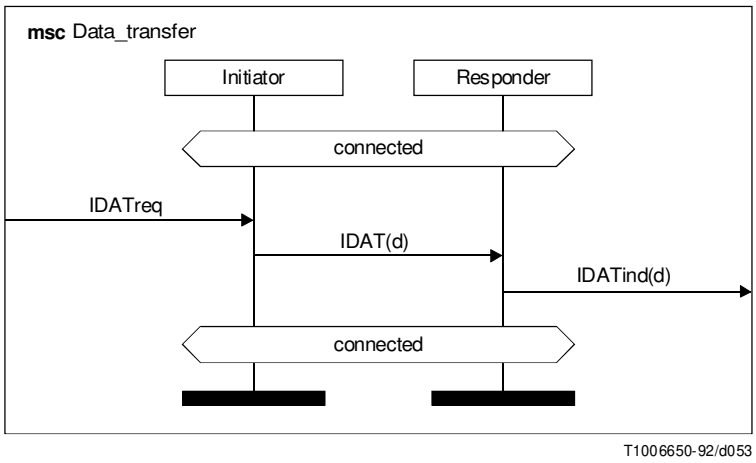
EXAMPLE I.6-5b

Atomic MSC



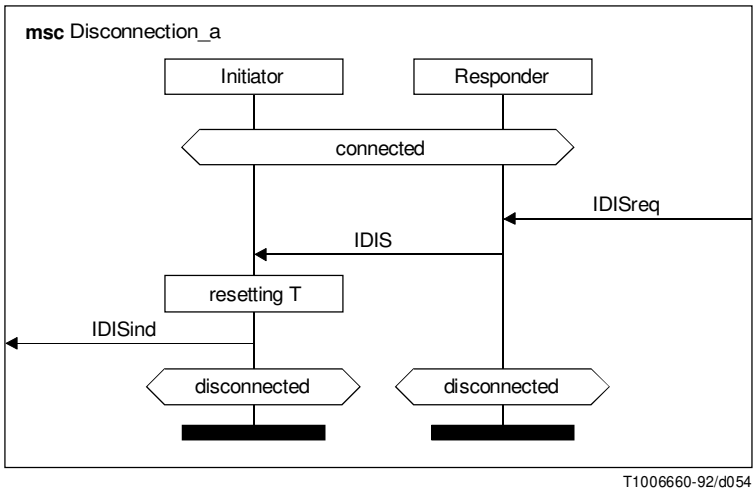
EXAMPLE I.6-5c

Atomic MSC



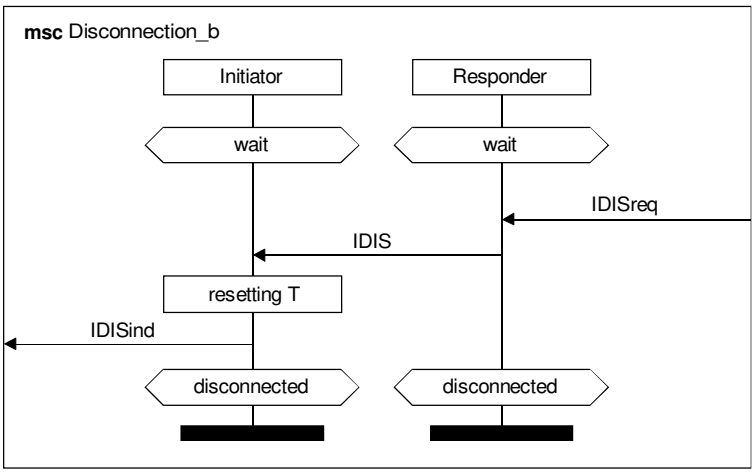
EXAMPLE I.6-5d

Atomic MSC



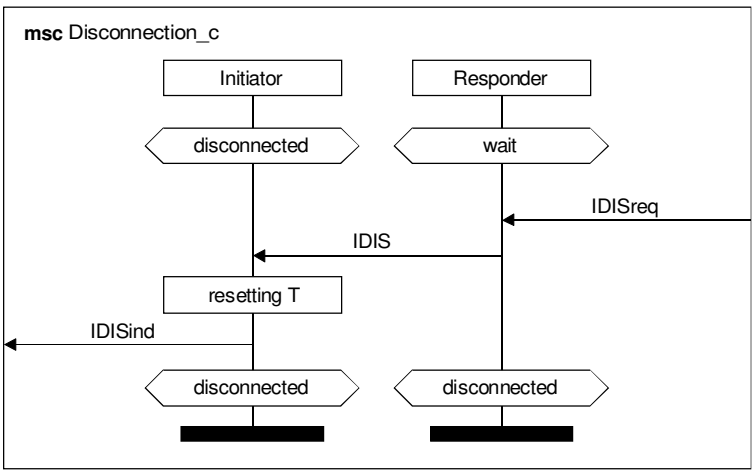
EXAMPLE I.6-5e

Atomic MSC



EXAMPLE I.6-5f

Atomic MSC



EXAMPLE I.6-5g

Atomic MSC

Step 3 – Creation of a MSC overview diagram

– Starting with the initial state and taking into account the MSC composition rules, the atomic MSCs may be joined together. The obtained set of sequences of MSCs may be presented in form of a *MSC overview diagram* by identifying the reached system states, provided by MSC-conditions.

Figure I.6-2 shows the MSC overview diagram obtained from the set of atomic MSCs in Examples I.6-5a to I.6-5g. The nodes represent global system states. The initial state *disconnected* is marked. Such a MSC overview diagram may be useful generally to show the relation/connection between several MSCs. A MSC overview diagram can be regarded as an auxiliary diagram for MSCs, corresponding to the state overview diagram in SDL.

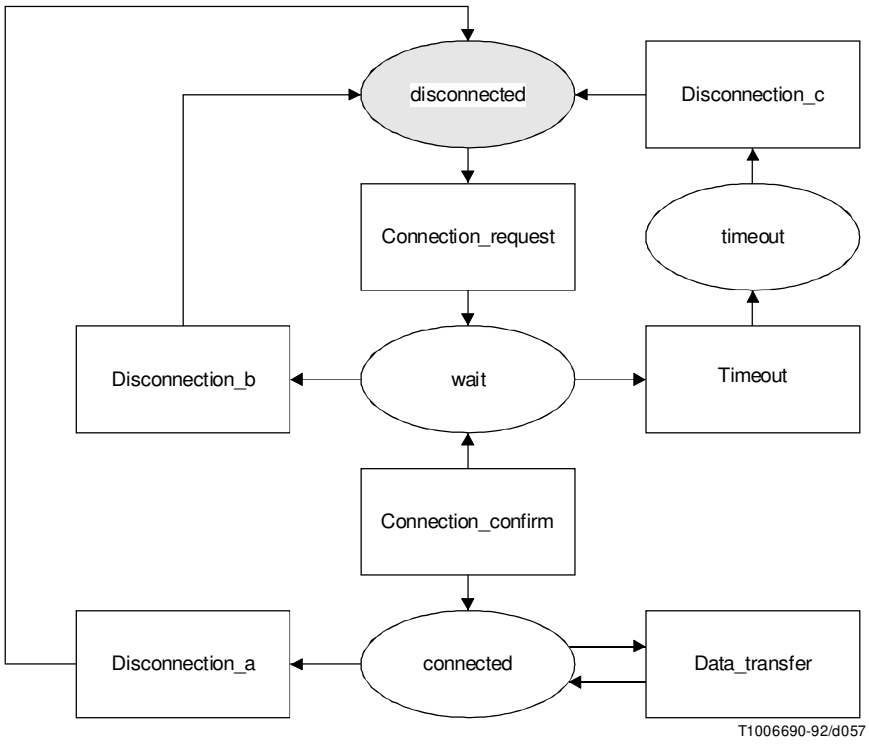


FIGURE I.6-2/Z.100

A MSC overview diagram

Step 4 – Splitting up the MSC overview diagram

– The MSC overview diagram is split up at the node assigned to the initial state. Starting from this node, the MSC overview diagram is split up further at nodes from which cyclic subtraces emerge. For each edge connected to such a node, a corresponding copy of this node is attached after decomposition.

Step 5

– The maximal traces within the obtained fragments of the MSC overview diagram form normalized MSCs.

We obtain five normalized MSCs from Figure I.6-3, of which *Data_transfer* and *Disconnection_a* are the same as in Example I.6-5. The new normalized MSCs are presented in Examples I.6-6a to I.6-6c.

In order to obtain an overview about all possible system traces, the relation/connection between the derived normalized MSCs has to be shown. This is provided in Figure I.6-4, where the normalized MSCs again are represented as transitions within a MSC overview diagram.

A comparison of Figures I.6-2 and I.6-4 demonstrates the structuring capability of normalized MSCs. Of course, the usefulness of normalized MSCs is even more evident in case of more realistic examples containing extensive system traces.

I.6.3 A method for system development based on MSC, SDL and functional decomposition

Since the main importance of MSCs lies in the stage of requirement definition, whereas SDL in industrial practice is used primarily for system design, it seems to be obvious to derive the SDL specification from a set of MSCs. However, such an approach is automatically performable in practice only up to a certain level. The SDL specification has to be refined afterwards by hand. In addition, such a strict chronology ("waterfall" life-cycle) presents a simplified view of system design. SDL and MSC should be used in parallel within different stages of system development, supplementing each other and being correlated in many ways. In the following, a more sophisticated methodology, supporting top down design based on refinement concepts, is presented.

A telecommunication system is described according to three different but interrelated points of view: *functional*, *behavioural* and *architectural*. The functional aspect may be modelled by a hierarchy of functions, the behavioural aspect by message sequence charts and the architectural aspect by SDL. It should be noted that this schematic assignment only expresses the main emphasis, i.e. in each kind of description one aspect is presented most clearly.

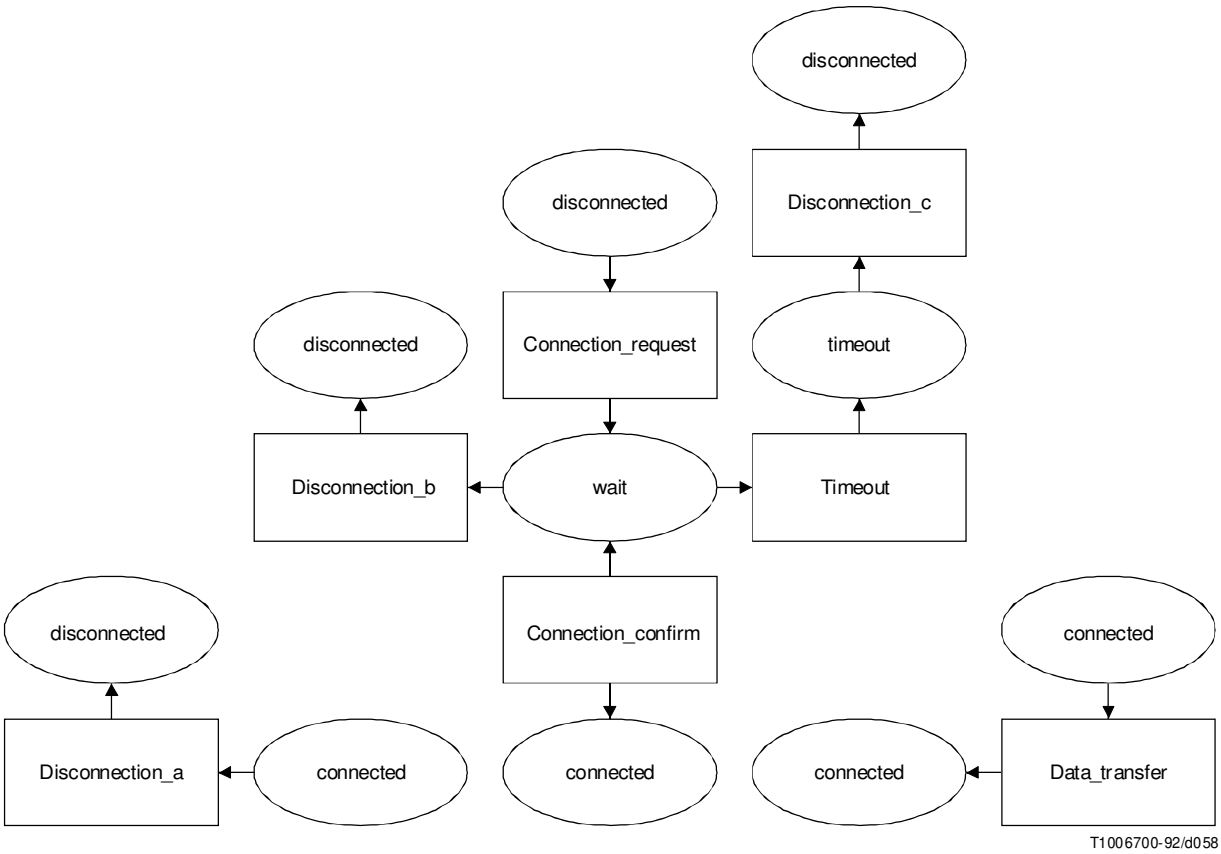
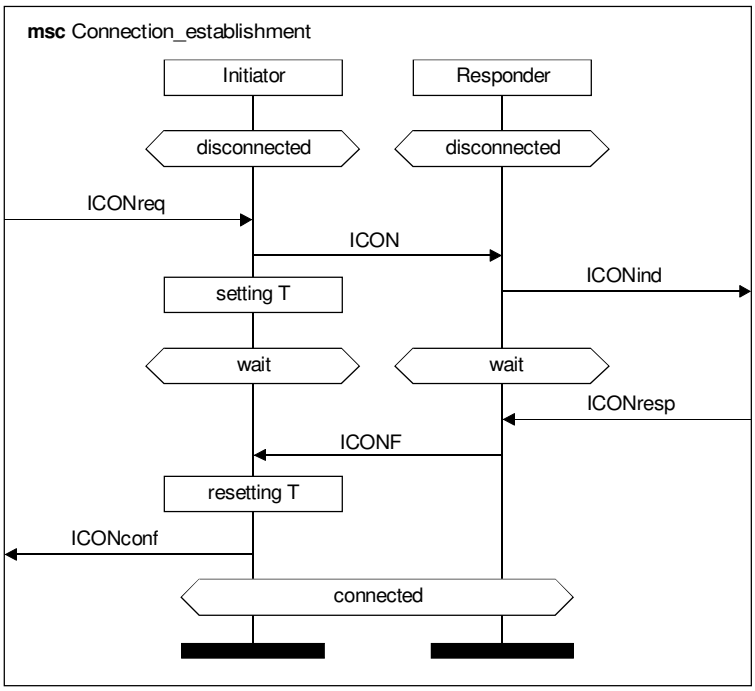


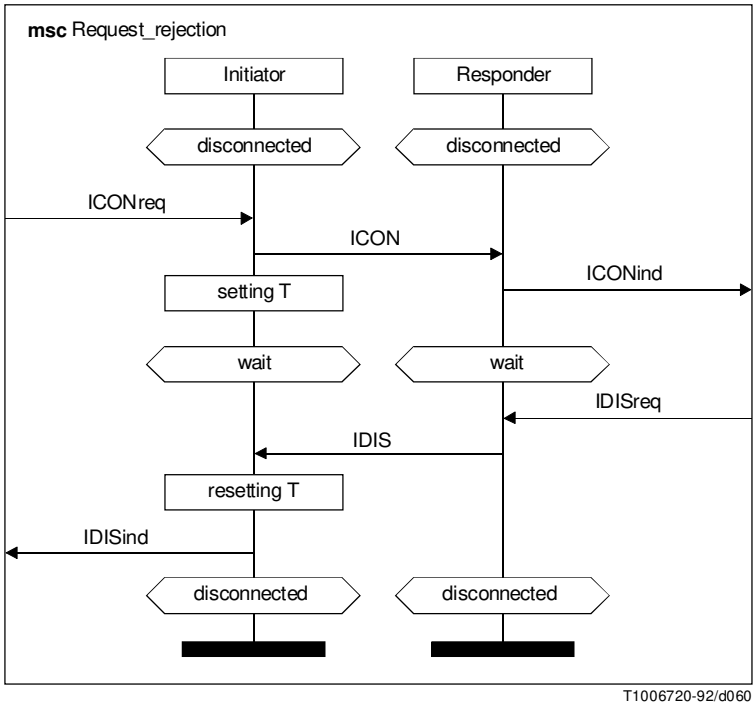
FIGURE I.6-3/Z.100

Splitting up the MSC overview diagram



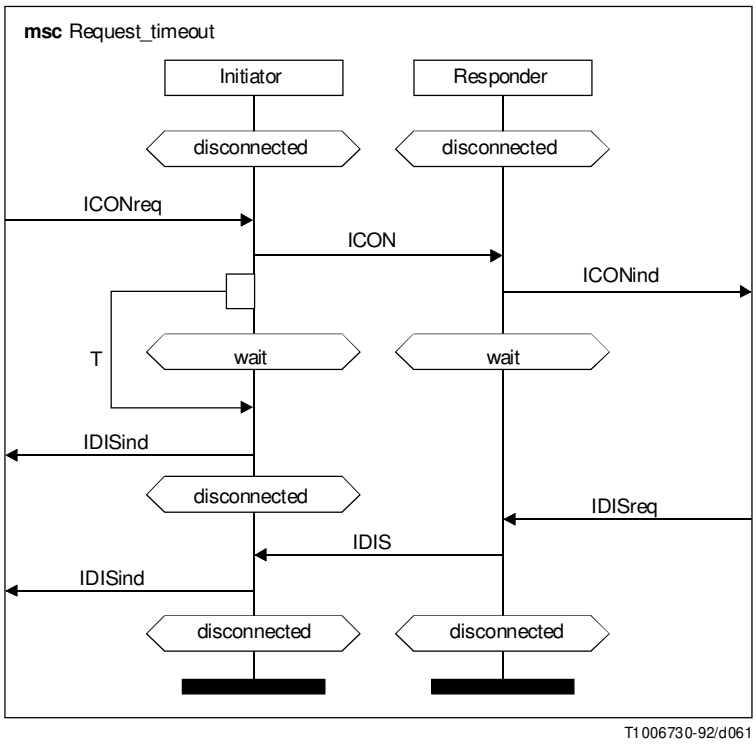
EXAMPLE I.6-6a

Normalized MSC



EXAMPLE I.6-6b

Normalized MSC



EXAMPLE I.6-6c

Normalized MSC

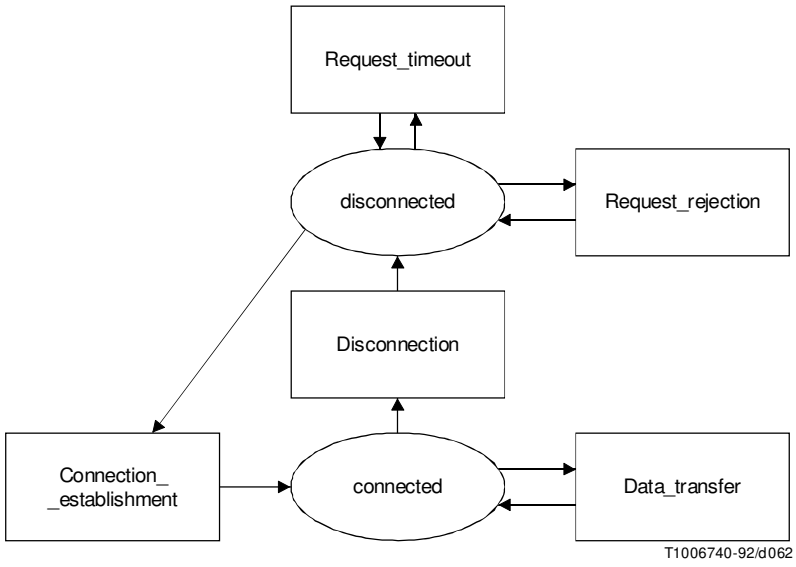


FIGURE I.6-4/Z.100

Overview diagram for normalized MSCs

I.6.3.1 Functional decomposition

Functions are used for the functional system specification, and we can distinguish between

- *Global functions*, each of them is decomposed into a hierarchy of lower level functions;
- *Abstract functions*, which represent intermediate nodes on different levels in the function hierarchy;
- *Terminal functions*, which form the leaves in the function hierarchy, and are thus not further decomposed. They represent very precise functions and may be described by a hierarchy of message sequence charts.

In order to express the temporal relationship between functions, a composition operator is attached to each global and abstract function, indicating how subfunctions are temporally ordered between each other. There are six operators:

- **is** – the resulting function is a subfunction;
- **and** – the resulting function is the sequence of subfunctions temporally ordered from left to right;
- **parallel** – there is no temporal ordering between the subfunctions;
- **or** – the resulting function is one of the subfunctions;
- **repeat** – the resulting function is the repetition of its subfunction;
- **exception** – the resulting function is its subfunction when an error is encountered.

Figure I.6-5 shows an example of functional decomposition. The *normal call* function corresponds to the sequence *dialing*, *conversation* and *disconnection*. Note that the symbols used for the operators are not part of SDL.

The formalism is used to express what the system must perform. SDL will describe how the system is to perform it. The functions defined are thus implemented by the SDL description.

I.6.3.2 SDL specification

By means of SDL the complete system is specified. One important aspect of SDL is the ability to describe a hierarchical decomposition representing the architecture of the system. Each SDL node in this hierarchy (system, block, process) is called in the following an *entity*.

I.6.3.3 Message sequence charts

A set of message sequence charts (MSC) is attached to each terminal function. This set again is structured as a hierarchy of MSCs. The MSC hierarchy must correspond to the SDL hierarchy. Each MSC describes how a terminal service is provided at a given decomposition level of the SDL system architecture. An MSC can be considered as the projection of a terminal service onto a decomposition level of the SDL hierarchy.

I.6.3.4 Consistency rules between descriptions

The presented methodology requires that the three kinds of description are kept consistent during all stages of system design.

Figure I.6-6 shows a global view of the relationships between the three kinds of description by means of a simple example, and illustrates the consistency rules. The following rules must hold:

- MSC decomposition must be consistent with SDL decomposition.
- All instances and messages in an MSC must be visible in the corresponding entity in the SDL description.
- A MSC must be consistent with the SDL description on the corresponding decomposition level.

One MSC-instance shows a set of input and output events which are totally linear ordered. Therefore, a MSC-instance can be interpreted as a sequence of events. This sequence is called a MSC-trace.

An SDL process specification can be represented as a reachability graph: the nodes are process states and the labelled edges represent input events. An SDL-trace of a process is a sequence of events in the reachability graph.

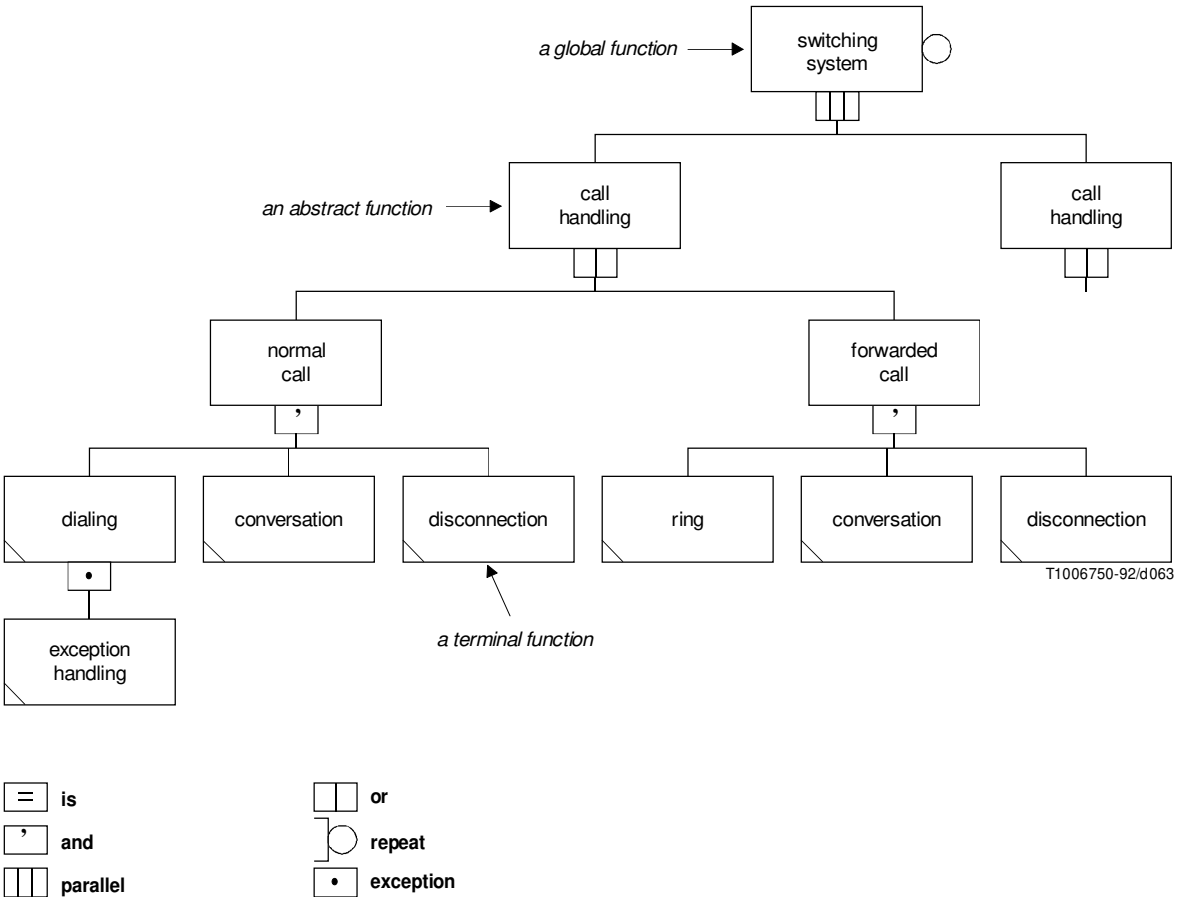


FIGURE I.6-5/Z.100

Example of a functional decomposition

Now, the definition of consistency is reduced to a graph theoretical problem. A MSC-instance is consistent with an SDL process specification, if there is an SDL-trace which contains the MSC-trace and shows no additional input and output events (see Figure I.6-7).

This consistency rule is not sufficient for all kinds of MSCs. One might drop information about the signal flow between processes, e.g. in case of signal overtaking. The definition of traces and a corresponding consistency rule may be generalized to cover all kinds of MSCs. In practice, however, an automatic consistency check is feasible, because of state explosion, only for restricted MSC-instances.

I.7 Derivation of implementations from SDL specifications

The problem addressed in this clause is how to map abstract systems specified with SDL into real systems composed from hardware and software components. Real world components normally differ from abstract components in structure as well as in behaviour. Therefore adaptations are needed both on the abstract and the concrete level in order ensure that the SDL specification faithfully defines the functionality of the real system. Documentation in addition to pure SDL is needed to define the concrete system and its relationships to the abstract SDL system. This clause outlines the major steps in deriving implementations, and how the implementation may be documented on the architectural level. The step-wise guidelines are summarized at the end of the clause.

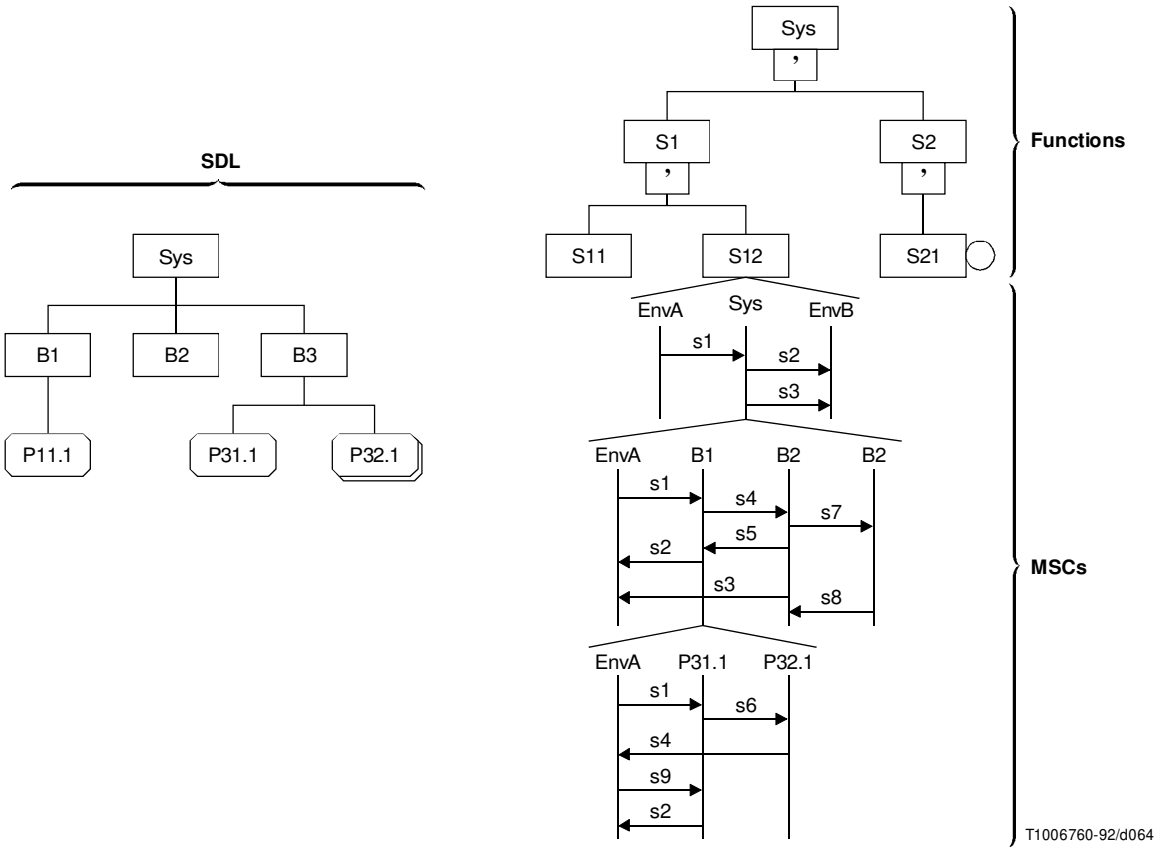


FIGURE I.6-6/Z.100

The three views of a system

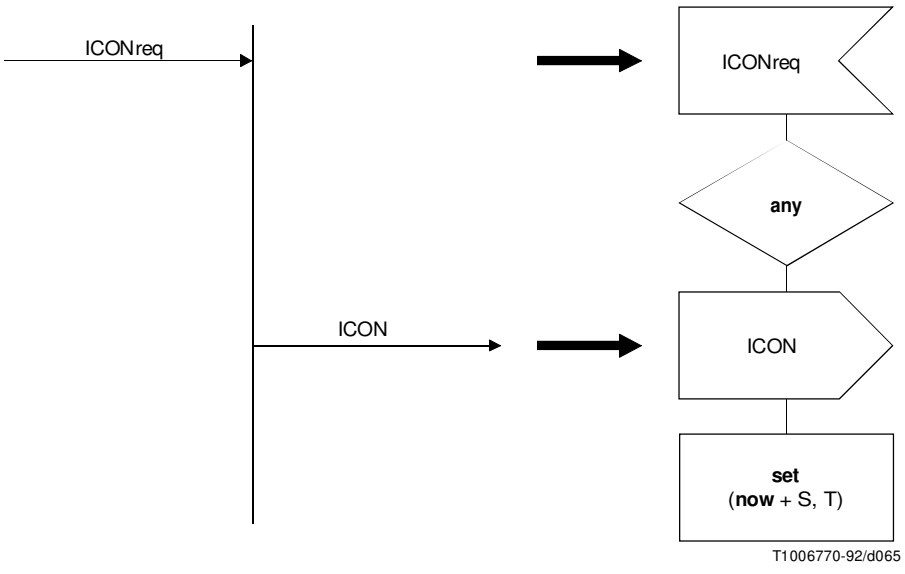


FIGURE I.6-7/Z.100

Consistency between SDL-trace and MSC-trace

I.7.1 Introduction

During the life time of a system, SDL specifications will be used for at least three purposes:

- At an early stage, to specify and to validate the functionality (behaviour) required by the user environment (high level specification).
- Then, to provide a firm basis for implementation design, i.e. designing the optimum implementation (low level specification).
- After implementation design, to describe (document) the complete functional properties of the system as implemented (implementation oriented specification).

SDL is based on concepts well suited for the first purpose above: to specify the observable behaviour of systems in a clear and unambiguous way. For this purpose, the external behaviour should be emphasized and irrelevant internal design details should be discarded.

SDL is also well suited for the second purpose: to be a basis for implementation design. SDL has the nice property of combining implementation independence with implementability (except for infinite abstract data types). For this purpose, no premature design decisions should be embedded in the SDL specifications, but so called non-functional requirements, i.e properties the implementation shall have in addition to those expressed in the SDL specifications, may be given as additional guidance.

If the real system is functionally equivalent to the SDL system, the third purpose above is achieved too. This is well worth aiming for. Not only because it saves documentation effort, but also because the SDL specifications will be useful during system testing, operation and maintenance. When designers and programmers find the SDL specifications useful in their daily work, they will be motivated to keep them up to date and avoid changes on the implementation level that degenerate the documentation. For this purpose, the SDL specifications must normally be somewhat adapted to reflect functional properties that are specific to the underlying implementation.

There are two major aspects involved in designing the implementation of an SDL system:

- The forward aspect of selecting among implementation alternatives being functionally equivalent to the SDL system. Normally, there will be several such alternatives from which the designer should select an optimum implementation with respect to non-functional requirements.
- The feedback aspect of adapting the SDL specification in the case when the selected implementation is not functionally equivalent. There are important differences between the abstract world of SDL and the real world that sometimes will show up in the observable behaviour of the system. In such cases the complete SDL specification of the system should be adapted to maintain functional equivalence.

Figure I.7-1 illustrates the forward aspect. The SDL specification and the non-functional requirements together are used to select and specify an implementation. The implementation specification resulting from this activity defines the mapping from the SDL specification to the real system implementation. It is orthogonal to the SDL specification and therefore shown as a separate box in Figure I.7-1. The separation is not the key point here, but the orthogonal nature of the information. In practice the implementation specification might be embedded as annotations to the SDL specification. A few observations can be made in relation to Figure I.7-1:

- One and the same SDL specification fulfils in this case all three purposes of SDL specifications mentioned above.
- Alternative implementations may be derived from the same SDL specification by providing alternative implementation specifications.
- If the implementation activity is automated, the implementation will be kept consistent with the specification at all times (up to the point of translation correctness).

In many practical situations, the somewhat idealized picture of Figure I.7-1 will be modified by the feedback aspect. If the system is to be realized by a distributed computer network, for instance, some of the SDL channels will be implemented using the network protocols. These protocols are needed in a distributed implementation, but not in a centralized solution. Since parts of a distributed system can go down while other parts remain operational, error handling is different in a distributed system compared to a centralized system. Therefore, in order to define the complete functionality actually implemented, the SDL specification of a distributed system may differ from that of a centralized system.

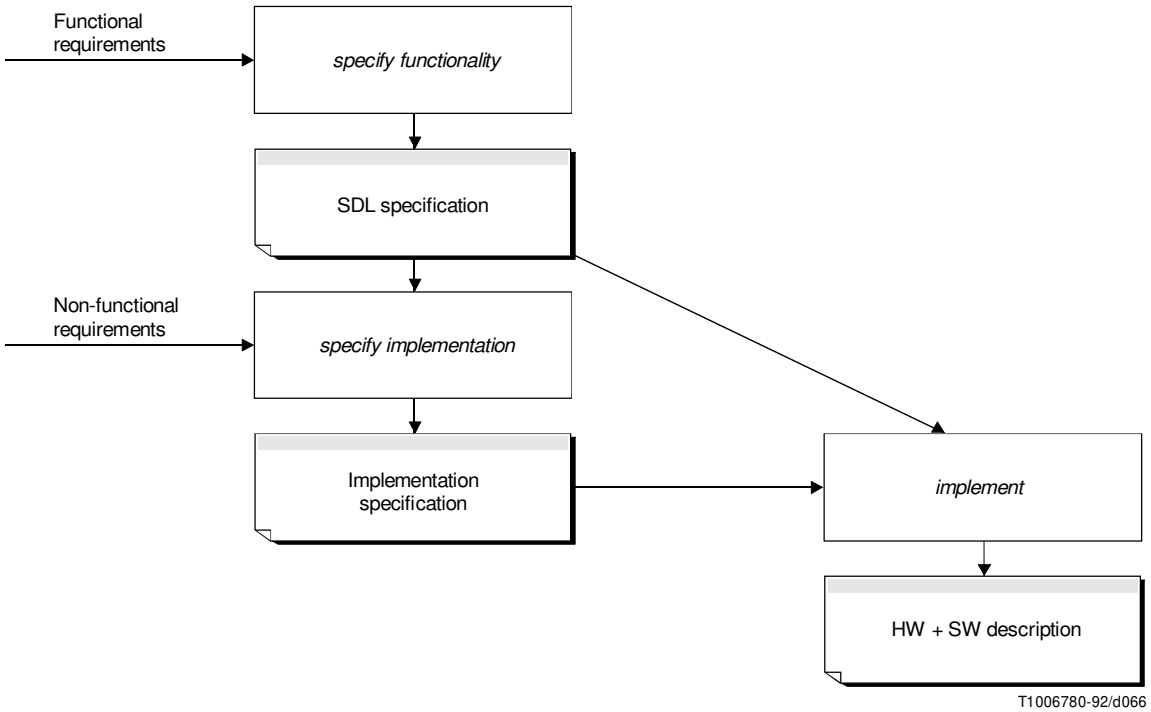


FIGURE I.7-1/Z.100

The forward aspect; non-functional requirements are used to select a functionally equivalent implementation

Figure I.7-2 illustrate the interplay between the forward and the feedback aspect. As in Figure I.7-1, the non-functional properties are used to derive an implementation specification. Knowledge about the implementation is then used to refine and restructure the SDL specification to a complete refined low level SDL specification. But even this latter SDL specification will leave room for alternative, functionally equivalent, implementations. Therefore, the implementation specification is still needed to direct the implementation step just as in Figure I.7-1.

The following observations apply to Figure I.7-2:

- Two SDL specifications, a high level specification and a low level specification are needed to fulfil all three purposes of SDL.
- For a given high level SDL specification, several low level SDL specifications may be derived depending on the implementation specification.
- The feedback aspect and the forward aspect can be separated and combined in a structured fashion.

When a new system is developed from scratch, it is desirable that the SDL specification is sufficiently implementation independent to fulfil the second purpose (mentioned in the beginning of this section). This can mean considerable feedback when the implementation is chosen[4]. When an existing system is extended or enhanced, however, more is known about the implementation, so one may aim more directly at the low level SDL specification and avoid feedback.

---

4) But it does not mean that everything in the specification is changed. Often, the refinement and restructuring needed can be confined to limited parts of the system.

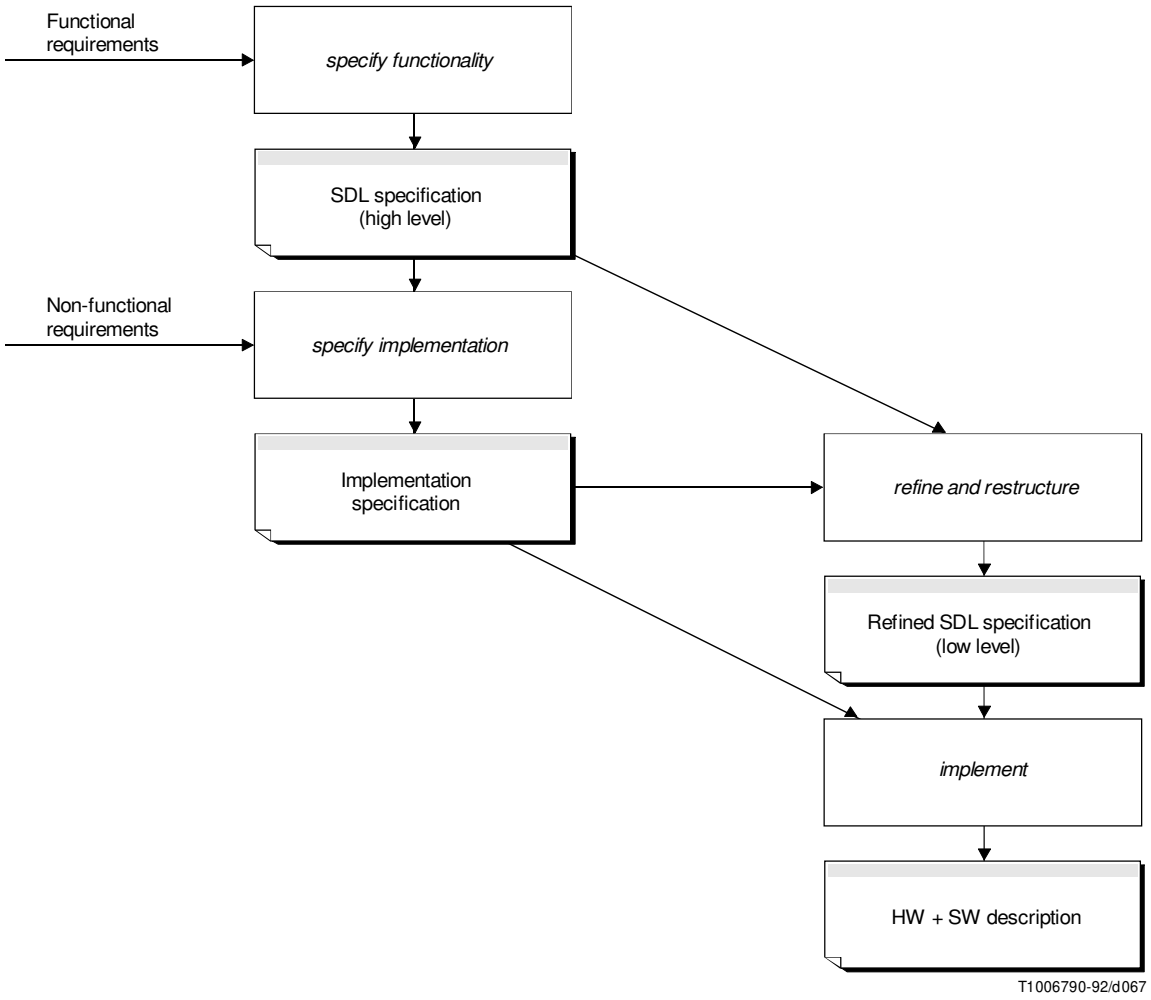


FIGURE I.7-2/Z.100

The forward and the feedback aspect of implementation design

Three questions may now be asked:

- What are the essential differences between SDL systems and real systems that need consideration?
- How should the implementation specification be expressed: as annotations to the low level SDL specification, or separately from the SDL specification?
- What are the main implementation alternatives to choose from?

These questions will be discussed in the remainder of this subclause.

I.7.2 Differences between real systems and SDL systems

A good designer must be well aware of the differences between the real and the abstract world. There are two main categories of such differences:

- Fundamental differences in the nature of components. Physical components are rather imperfect compared to the more ideal properties of SDL components. They develop errors over time, they are subject to noise, and they need time to perform their processing tasks.
- Conceptual differences in the functioning of components. In both worlds there are concepts for concurrency, communication, sequential behaviour and data, but they are not necessarily the same.

I.7.2.1 Fundamental differences

Processing time

An SDL system is not limited by processing resources. Consequently, the balance between the traffic load offered to the system and its processing capacity needs not be considered. One simply assumes that the system is fast enough to process the load it is offered.

The real world is very much different on this point. Each signal transfer, and each transition of a process will take some time and require some processing resources. Therefore one of the major issues in implementation design is to balance the processing capacity of the implementation against the offered traffic load.

A related issue is how fast the system must respond to certain inputs. Again the SDL system has no problems, but the implementation may be highly pressed to meet response time requirements.

Due to these differences, the focal point of implementation design is quite different from that of the functional specification. The challenge is to find implementations for the SDL concepts that are sufficiently fast to meet the traffic load and response time requirements without destroying the validity of the SDL specifications.

Since SDL specifications clearly specify the external and internal interactions needed to perform given functions, they provide an excellent basis for estimating the processing capacity needed to meet load and timing requirements. This is elaborated in I.7.4.3.

Errors and noise

SDL systems may suffer from specification errors, but the abstract world of SDL does not suffer from physical errors. It is simply assumed that processes and channels always operate according to their specification. It is not assumed that processes will stop from time to time, or that channels will distort the content of signals. But in the real world, such things happen. From time to time, errors will manifest themselves as faults in the operation of channels and processes.

Hardware errors, noise, and physical damage are inevitable, since they are caused by physical processes being entirely outside the realm of logic. In addition, logical errors will normally be introduced in the implementation that were not in the SDL specification.

The mechanisms causing errors and noise are outside the scope of the SDL specifications. But the effect of errors and noise will often need to be handled explicitly in the SDL specification. One must consider what a process should do if it never gets a response from another process, or if it gets an erroneous response. Is it possible that a channel will go down, or that noise may distort the signals? What should be the reaction to a channel going down? What if a process starts to produce wild signals? These are the kind of questions to ask, and to find answers to.

To some extent, the answers depend on the physical distribution of SDL processes in the real system and the physical distances that SDL channels must cover. Physically separate processes and channels may fail independently of each other. Channels covering long physical distances are subject to more noise and errors than channels implemented in software within one computer.

A positive effect of physical separation is that errors are isolated. Errors in one unit need not affect the other units in the system, provided that erroneous information is not propagated to them or that they are able to protect themselves. Thus physical separation may improve the error handling. But this will normally not come for free. Errors need to be detected and isolated to allow the operational parts to continue operation with the error present. Proper handling of this aspect can be quite complex and will normally require additional functionality in the SDL specification.

An SDL specification does not tell anything about the physical distance covered by a channel. In reality, however, there may (or may not) be large physical distances. This means that transmission equipment and protocols are needed to implement the channel reliably. Thus, physical distance may introduce new functions needed to support the implementation of channels.

Finite resources

All resources in a real system are finite. There may be a maximum number of processes the operating system can handle, or a maximum number of buffers for sending messages. The word length is restricted, as is the memory space. Even primitive data like integers are finite.

SDL, on the other hand, has an unbounded queue in the input port of each process, and allows infinite data to be specified. Hence, the designer must find ways to implement potentially infinite SDL systems using finite resources. One way is to restrict the use of SDL such that all values are sure to be bounded. Another is to deal with resource limitations in the implementation, preferably in a manner transparent to the SDL level. In cases where transparency cannot be achieved, one must either accept deviation from the SDL semantics, or explicitly handle the limitations in the SDL specification.

I.7.2.2 Conceptual differences

Concurrency

The model of concurrency used in SDL assumes that processes behave independently and asynchronously. There is no relative ordering of operations in different processes except the ordering implied by the sending and reception of signals.

This permits SDL processes either to be implemented truly in parallel on separate hardware units, or in quasi-parallel on shared hardware.

Physical objects in the real world behave truly in parallel. This means that operations within different objects proceed in parallel to each other at the speed of the performing hardware. We may distinguish between two cases:

- *Synchronous operation,* where the operations of parallel objects are performed at the same time. This is mostly used on the electronic circuit level where operations may be controlled by common clock signals. In relation to the SDL semantics this may be seen as a special case.

- *Asynchronous operation*, where the operations of parallel objects are performed independently and possibly at different times. Unless we have detailed knowledge of the processing speeds we cannot know the exact ordering of operations. This corresponds well with the SDL semantics.

Quasi-parallelism means that only one process will be active at the time and that the active process will block the operation of other processes as long as it is allowed to remain active. This will affect the response times of the blocked processes. Additional support will be needed to perform scheduling and multiplexing of the quasi-parallel processes on top of the sequential machine. Normally, this is handled by an operating system, which can be seen as a layer that implements a quasi-parallel virtual machine on top of the physical machine.

Communication

Very basically, there are two different classes of communication needs:

- to communicate a sequence of symbols, or values, in a given order;

- to communicate a symbol, or value, continuously for the time it is valid.

In the first case, the sequential ordering is important. In the second case, the sequence does not matter, only the current value at each instant in time.

When reading a book, the sequence of letters, words and sentences is essential to the interpretation. When arriving at a traffic light, the current colour will determine whether one shall stop or not. The history of previous changes in colour does not matter. Continuous symbols may be read over and over again, whereas symbols in a sequence normally is read only once.

SDL signals belong to the symbol sequence category. They are read only once and then consumed at the receiving end. They are well suited to the representation of event sequences.

The view/reveal construct in SDL belongs to the continuous value category. The export/import construct looks similar, but is a shorthand representation for an underlying protocol of signals enabling the receiver to keep track of a continuous value.

The two communication forms are dual in the sense that one form may be used to implement the other. We consider first the need to communicate a continuous value. The most direct implementation is to use a communication medium that will transmit the value continuously, such as a shared variable in software, or an electrical connection in hardware. But one may alternatively use a sequence oriented medium, such as a message queue, to transmit a sequence of symbols representing the sequence of changes (events) in the continuous value. For this scheme to work, the sender must derive events, and the receiver must integrate events to regenerate the continuous value. This principle is applied in the SDL export/import mechanism.

What about the need to communicate a symbol sequence? The most direct implementation is to use a sequence oriented communication medium such as a message queue. But one may alternatively use a continuous value medium. In this case, the sequence has to be represented by the changes (events) in the continuous value. For this scheme to work, the sender must integrate a sequence of events to form the continuous value and the receiver must derive the events from changes in the continuous value.

Although it is possible to implement either form of communication by means of the other, there is a penalty associated with a change in paradigm.

The "natural" implementation medium for SDL signals will be symbol sequence oriented. But this will not always be the best form in the real system. Sometimes, the SDL signals have to be implemented by means of continuous values.

Input from a keypad may serve as an example. The output signal from each button is basically a continuous "1" when pushed, and a continuous "0" when not pushed. But the system need to know the sequence of key strokes, and not the instant values. Thus, the value changes (events) need to be detected and converted to symbols representing complete key strokes. Event detection like this is often needed at the interfaces of a real-time system. It may either be performed in software or in hardware.

Visual signals on a display screen are another example. The user wants information presented as continuous values, and not as messages flickering across the screen. Hence, the SDL signal has to be converted to a continuous value on the screen.

On the other hand, when export/import are used in SDL, it may be better to use continuous values in the implementation rather than the symbol sequence protocol assumed in SDL[5)].

Channels crossing the hardware-software boundary need special attention. An atomic channel, represented by a line in the graphical representation may turn out to be a mixture of physical lines, electronic equipment and software in the real system. The communication and synchronization primitives used at the hardware side will often differ from those used on the software side of the boundary, see the next subclauses. Conversion from one form to another will be necessary. This is often a time critical task needing careful optimization.

Synchronization

Consider two SDL processes that communicate. The sending process may send a signal at any time because it will be buffered in the input port of the receiving process. The receiving process may then consume the signal at a later time.

This is a buffered communication scheme in which the sender may produce infinitely many signals without waiting for the receiver to consume them. It is often referred to as *asynchronous communication*.

Asynchronous communication may be contrasted with so-called *synchronous communication* in which the sending operation and the consuming operation occur at the same time. This is necessary when there is no buffer between the processes.

The act of aligning the operations of different concurrent processes in relation to each other is generally called *synchronization*. Synchronization is necessary not only to achieve correctness in communication, but also to control the access to shared resources in the physical system.

---

5) This changes the SDL semantics slightly, so one should be careful. But normally it will be acceptable to the user, and far more efficient.

Synchronization of interactions can be classified in the following categories:

a) *Synchronous interaction,* where the processes perform interaction operations at the same time. There are two sub-categories:

   1) *Time dependent* – In this case the transmission medium itself is synchronous as for instance the channels in a PCM system. The sender and the receiver has to keep in step with the common timing of the transmission medium. This puts real time constraints on the sender as well as on the receiver.

   2) *Time independent*, or explicitly synchronized – In this case an explicit, time independent, synchronization takes place between the processes whereby they are locked together during interaction. Processes may have to wait for the interaction to be enabled. Once it is enabled, the interaction take place by operations that are simultaneously performed by both processes. LOTOS communication is in this category. This is a sequence oriented scheme.

b) *Asynchronous interaction,* where the processes not necessarily perform interaction operations at the same time. Still the operations have to be aligned in some relative order in order to ensure correct interaction. For continuous value communication the main requirement is that operations are *mutually exclusive*. For value sequence communication there are two sub-categories:

   1) *Time dependent* – The transmission medium itself is asynchronous, with no explicit synchronization mechanism on the medium. This scheme depends on the relative speed of the receiver compared to the sender. The sender must produce output at a rate the receiver is able to follow. A typical example is the decadic pulse dialling on old telephone subscriber lines. This scheme is much used on physical communication links (and is a notorious cause of real time problems).

   2) *Time independent* or explicitly synchronized – In this case, there is a buffer and an explicit time independent synchronization between the interacting processes. The buffer is normally a FIFO queue. The sender puts (a sequence of) values (signals) into the buffer, and the receiver removes the (sequence of) values at some later time. The buffer capacity determines how many values the sender may produce ahead of the receiver. SDL signal communication is in this category. It is much used in software systems, but is less common in hardware. Synchronous interaction may be seen as a special case where the buffer capacity is zero.

The SDL semantics assume asynchronous interaction with infinite buffer capacity. In practice, however, the buffer capacity must be bounded in one way or another.

Sometimes the application is such that the number of signals the producer is able to generate ahead of the consumer always will be limited. In other cases, the limitation depends on the consumer being fast enough to prevent the queues from growing too long. In general, one must deal with the buffer overflow problem by delaying the producer when the buffer reaches its maximum capacity.

In consequence, output from an SDL process may have to be delayed until the receiving buffer is ready. This deviation from the SDL semantics can hardly be avoided in a finite implementation. Careful engineering is needed to reduce to a minimum the practical problems this may cause.

One will often find mechanisms that differ from the SDL mechanisms at the physical interfaces to the system. It is quite typical to find time dependent interaction on physical channels for instance. This implies that time critical event monitoring and event generation will be necessary.

A designer will be faced on one hand by the synchronization primitives available in the real system, and on the other hand by the synchronization implied by the SDL specification. Additional functionality will often be needed to glue the various forms together.

Data

SDL data is based on the notion of abstract data types, where operations may be defined by means of equations. An implementation will normally need concrete data types where the operations are defined operationally. Therefore, the designer may need to transform the abstract data types of SDL into more concrete data types suitable for implementation.

I.7.3 Implementation specifications

It follows from the implementation independence that one and the same SDL specification may be reused in several different implementations. Also, that the physical structure of those implementations may vary considerably. In one instance, each SDL process may be implemented on a separate physical chip, in another, they may all be software running on the same computer. The way this is done is something that needs to be documented.

I.7.3.1 Rationale

Implementation specifications define the mapping from an abstract SDL system to a concrete system made up of hardware and software components. Clearly, an SDL system can be restructured and refined up to a point where it reflects much of the design. But still, the SDL system will be abstract and possible to implement in different ways. Therefore, something is needed in addition to pure SDL to define the implementation.

How should the implementation specification be expressed?

One possibility is to add implementation specifications as annotations to SDL specifications. The advantage of that solution is that all information describing the system can be found in one place. The disadvantage is that the SDL specifications are tied to particular implementation designs. By describing the implementation separately, it will be easier to reuse SDL specifications in systems having different implementations. It will also be easier to put the particular aspects of implementation design into focus during the implementation design activity.

The following aspects need consideration:

- the overall structure of the real system in terms of hardware and software components;
- the non-functional properties of the real components and the resulting system. In particular, its performance in terms of traffic handling capacity, response times and error handling;
- the mapping from the abstract SDL system into components of the real system.

It may be an advantage to specify and analyse these aspects somewhat disconnected from the functional structure of SDL systems. The main structuring criteria for real systems are related to performance, cost and physical properties, whereas the criteria for SDL specifications are clarity and completeness of behaviour. These criteria are so different by nature that they will not always yield similar structures. Moreover, there are aspects of real systems that are complex enough by themselves to justify a separate specification.

In order to discuss aspects of implementation design independently of SDL specifications without going into the details of particular implementations, a notation for implementation specifications will be informally introduced in the next subclause.

I.7.3.2 Notation for implementation specifications

The notation for implementation specifications focuses on the structures of hardware and software, and the mappings between specifications on various levels of abstraction. Note that this notation is not normative.

It is a quite general notation for block diagrams. In many respects it is syntactically similar to SDL block interaction diagrams, but there are important differences in the meaning.

The boxes and arrows of SDL block interaction diagrams represent abstract blocks and channels with well defined semantics. They may be understood and analyzed in their own terms independently of the realization.

The purpose of the implementation specification is to define the mapping from SDL specifications to the realization. For this purpose, it does not need a semantics of its own in the sense that SDL has a semantics. Its meaning comes from what it represents in the real world. For the real world itself we have other formalisms, such as programming languages and hardware description languages with well defined semantics. Therefore, the scope of the implementation specification can be limited to a syntactic mapping.

By representing the structure of the realization using a graphical notation, we gain overview and insight into the structure of the physical system. A hardware structure example is depicted in Figure I.7-3.

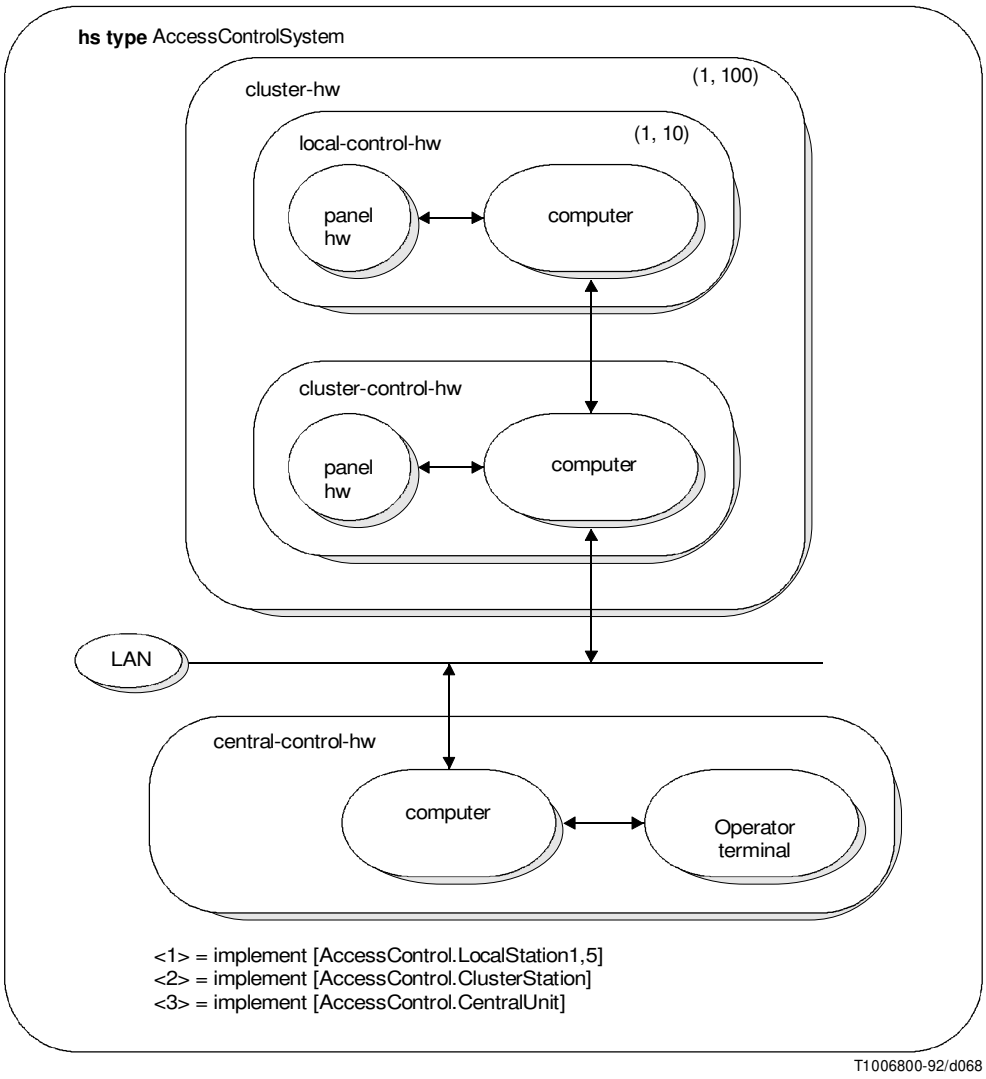


FIGURE I.7-3/Z.100

A hardware structure example

Boxes represent concrete hardware units such as computers and circuit boards. Arrows represent physical connections, such as cables. With this notation, the hardware specification may be decomposed to provide gradual approach to details. The idea is to use this notation to define the overall structure, and then refer to special hardware notations such as circuit diagrams for the details. In that manner, a well structured hardware specification can be made. See the table of hardware diagram symbols at the end of the subclause.

Arrowheads indicate the direction of signals flowing through a connection. Two-way connections are possible.

The notation for software structures is based on the same principles as for the hardware structure notation, but the boxes have different shapes. This is used to distinguish characteristically different types of software units (see Figure I.7-4).

The triple sided box represents the *software process,* a software unit containing at least one non-terminating program. Such programs will be executed as quasi-parallel processes under an operating system. The triple sided boxes therefore represent concurrent units. Units that only contains terminating programs such as procedures are represented by the double-sided box, and pure data are represented by a single-sided box.

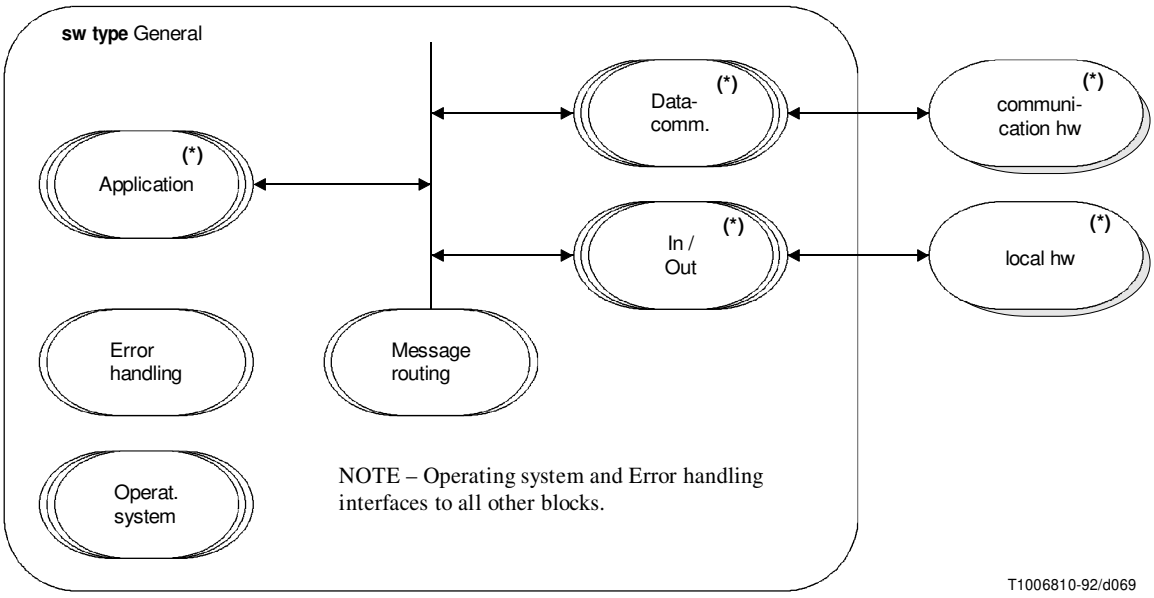


FIGURE I.7-4/Z.100

A software structure example

The software structure depicted in Figure I.7-4 is quite general. It contains a number of software processes, some of which do application functions while some do input or output, and some do inter-computer communication. These processes are scheduled by an operating system and communicate by passing messages to each other through a message routing procedure.

The software structure notation is used to give an overview and to provide a gradual approach to software system details. This notation combines data structures and program structures in a unified notation. It allows network structured relations to be represented, and can express concurrency.

Other notations such as pseudo-code or source code may be used to define the software system in all detail.

Because the relations are hard to see in a linear program text, it is useful to represent them graphically. Since activation, or control flow, is quite clear from the program text, it is considered the least important relation to be represented in the software structure. Emphasis should be on data-flows and references.

The arrows used to represent data-flows, references, and activations between software units are differently shaped, see the symbol overview at the end of this subclause.

Both hardware and software structure diagrams may be decomposed hierarchically. They have similar concepts for types and instances. Both kind of diagrams are organized like the SDL graphical representation with a frame symbol representing the boundary between the specified entity and its environment.

In the upper left corner inside the frame symbol, the type of diagram and the name of the specified entity shall be placed:

hs type <name> specification of a hardware structure type

ss type <name> specification of a software structure type

hs <name> specification of a hardware structure instance

ss <name> specification of a software structure instance

Boxes and arrows may be collected in sets or arrays. The size of a set or array is indicated by a minimum and a maximum number of elements enclosed in parentheses:

(<min>,<max>)	at least *<min>*, at most *<max>*
(<min>,)	at least *<min>*, no upper bound
(+)	at least one, no upper bound
(*)	any number

A box is identified by a type name, and an optional instance name:

lu:LocalUnit	instance *lu*, type *LocalUnit*
CentralUnit	type *CentralUnit*

This syntax is similar to SDL.

Links to the environment are indicated by extending arrows to the frame symbol, or beyond. Entities and links in the environment may be represented outside the frame symbol and linked to entities inside the frame.

Implementation specifications specify the real system that is an implementation of the SDL system (see Figure I.7-5).

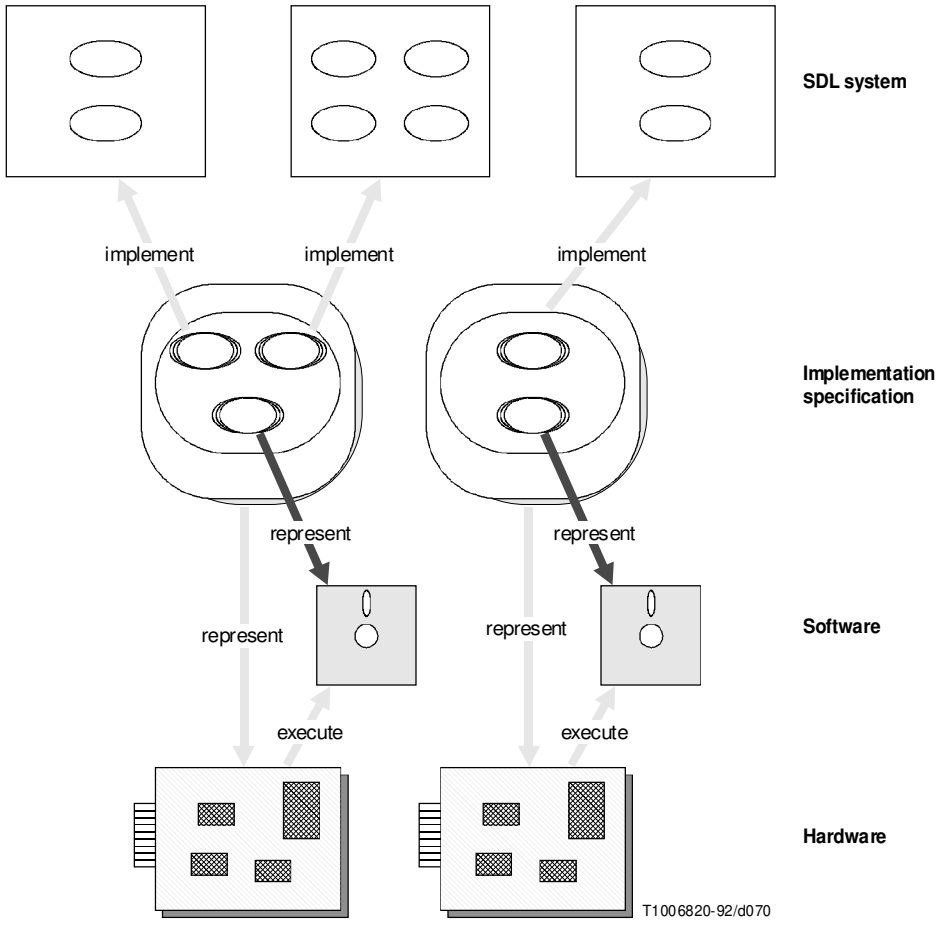


FIGURE I.7-5/Z.100

Implementation specifications specify the real system

In order to specify the mapping from the SDL system to the real system, it is necessary to add mapping information to the hardware and software structure diagrams. There are several mappings to take into account:

- source code *is translated* source specification;
- object code *is compiled* source code;
- executable code *is loaded* object code;
- physical hardware *executes* executable code;
- physical hardware + executable code *implements* source specification.

To fully control and document the design and implementation, all levels must be considered. For most practical documentation purposes, some levels may be be omitted.

Boxes and arrows in the implementation block diagrams are related on one hand to the SDL expressions, and on the other hand to the real world objects they represent. This is specified by means of *mapping expressions* in the diagram as illustrated in Figure I.7-3.

In order to save space in boxes and on arrows, mapping expressions may be referenced:

<1> mapping reference

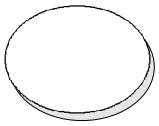
<1> = implement [**system** AccessControl.LocalStation] referenced mapping expression

The following mapping types are used (*node identifier* identifies the related entity):

- Implement [node identifier]
- Implemented-by [node identifier]
- Execute [node identifier] (implies that the software is loaded)
- Executed-by [node identifier]

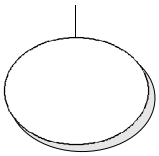
I.7.3.3 Symbol summary

Hardware structure diagram symbols



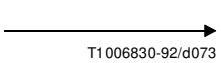
Hardware block

General symbol for a hardware block. May be decomposed recursively.



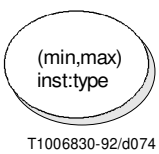
Switch

General symbol for a hardware block that performs switching and routing. The symbol may be tilted in any direction, and the single line may be stretched to be attached to many hardware links



Connection (optional arrows)

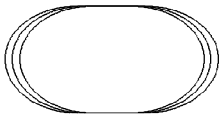
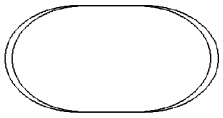
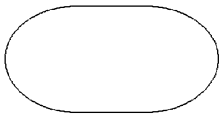
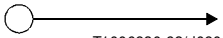
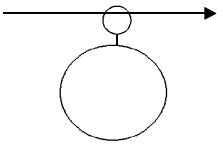
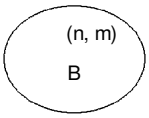
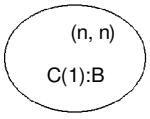
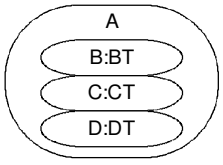
The arrow of the connection line is optional. The arrows show the direction of signals. One may have arrows at both ends. On the lowest level, connections are physical links.



Block set

The hardware blocks may be grouped in a set of blocks of the same type. Blocks shall have a type, and may have an instance identifier.

Software structure diagram symbols

 T1006830-92/d075	Software process block	A software unit containing at least one nonterminating program. May contain procedures and data.
 T1006830-92/d076	Procedure block	A software unit containing at least one terminating program but no non-terminating program. May contain data.
 T1006830-92/d077	Data block	A pure data element or a group of data elements. May not contain programs.
T1006830-92/d078	Data flow	Arrow denotes direction of flow.
T1006830-92/d079	Activation	Activation or call. Can not connect to data blocks.
 T1006830-92/d080	Reference	Reference, or pointer.
 T1006830-92/d081	Data flow by reference to message buffer	The line denotes data flow and the referenced square denotes the message buffer.
 T1006830-92/d082	Set of data elements of type B	A set of min *n*, max *m* elements of type B.
 T1006830-92/d083	Indexed array of data elements	An array of *n* elements called C(1) – C(n) of type B.
 T1006830-92/d084	Structure of data elements; record	A composite data structure (A) consisting of elements of different types. Each data element has a type and an instance identifier.

I.7.4 Trade-off between hardware and software

I.7.4.1 Introduction

The main emphasis in these guide lines are on software design. But neither software nor hardware can be designed completely independently of each other. The ability to meet time constraints, for instance, depends both on the speed of the computer and the amount of software needed to carry out a given function. In fact, there is a close interplay and many trade-offs to be made between hardware and software.

Once a suitable hardware structure has been selected, the consequences for software design must be considered. Peripheral hardware, for instance, has consequences for the input-output software. If the functions are distributed on many computers, we will need additional software for inter-computer communication and distributed error handling. Therefore, before all the software can be designed, it is necessary to know the overall hardware structure.

Hardware should not be designed without considering the software structure either. Therefore, coordination and trade-off between hardware and software design is necessary. The first step in implementation design is to perform this trade-off and design the overall architecture of the hardware and software taking the non-functional requirements into account. The major considerations are:

- physical distribution and physical interfaces;
- time constraints versus processing capacity;
- error handling requirements, i.e. detection, isolation and recovery from errors;
- security against unauthorized access to information;
- extendibility, operation and maintenance of the hardware;
- cost to develop, cost to produce, cost to maintain;
- use of existing components.

I.7.4.2 Physical distribution and interfaces

The SDL specification should not prematurely assume an internal physical partitioning of the system. But sometimes the physical location of interfaces implies a physical distribution of the system. The subscribers to a telematics system, for instance, will be physically distributed. Consequently, at least the user interfaces will be physically distributed.

SDL signals are defined independently of physical distances. One is therefore free to localize processes physically apart. But there will always be a certain transmission delay and cost associated with signal transfer over distances. One should therefore look for channels carrying a low signal traffic without strict timing constraints. Such channels may sometimes be found at the external system interfaces, but more often they will be found internally in the system.

This generalizes to a rule saying that we should distribute along the channels with few interactions and relaxed timing constraints (low bandwidth). We should keep strongly coupled processes together. This will often mean that a fair bit of processing should be performed physically close to the external interfaces.

In consequence, the implementation may partition the SDL system into physically distinct blocks and channels. Additional functionality is likely to be needed to support the distribution, e.g. signal transfer protocols, error handling.

I.7.4.3 Traffic load and response time

The goal is to find hardware and software that can meet the traffic load and response time requirements at a minimum cost. For this purpose, we need to estimate processing times and then calculate the processing load represented by the SDL application and its response times.

Simple mean value calculations may be performed as follows:

a) For each SDL process, P, calculate an average transition time t_p. This figure depends on the size of the software and the execution speed of the hardware and will have to be estimated. One may for instance calculate the mean number of operations o_p per transition (sending signals, timer operations, data operations) and then estimate a mean number of instructions i_p per operation. If the execution speed of the hardware is S instructions per second, then $t_p = i_p * o_p * S$.

b) Calculate the average number of transitions n_p each SDL process shall perform per second at peak load. This figure can be found by counting the number of transitions needed to carry out a given function, e.g. handle a telephone call, multiplied by the number of such functions the process shall perform per second.

c) Calculate the normalized, average peak load for each process:

$$l_p = n_p * t_p$$

This is a measure for how much processing time this process will need per second, i.e. the fraction of the processing capacity needed by this process (Erlang measure).

d) Calculate a corresponding load figure for each channel C and signal route R :

$$l_c = n_c * t_c$$

$$l_r = n_r * t_r$$

Where n_c and n_r are the number of signals per second, and t_c and t_r the processing times per signal transfer.

e) Calculate the mean load of the system as the sum of the channel, signal route and process loads. If this figure is higher than one, the average load is higher than the processing capacity of a single computer. In that case the capacity must be increased either by optimizing the software, speeding up the hardware or distributing the system. As a rule of thumb, the average load on a single computer should not exceed 0.3 in order to give room for statistical peak loads.

Note that the real load will vary statistically and have peak values considerably higher than the mean values we have been looking at. The system may therefore be overloaded for periods of time even though the average load is handled with good margin. A strategy for load control should therefore be part of the implementation design.

If the system may be run on a single computer, this should be preferred (unless other considerations demand something else). If it cannot be run on a single computer for performance reasons, it must be distributed. This will add communication overhead that must be included in revised load figures.

In addition to the load calculations above, it is necessary to check that the system will meet real-time constraints. The number of transitions and signal transfers per time-critical transaction must be found from the SDL diagrams, and the corresponding processing times calculated. Again, it is necessary to make allowance for statistical variations.

The hardware/software interface needs special consideration. It is not unusual that the larger part of a computer's capacity is spent doing input-output. Much can therefore be achieved by carefully designing the input-output interface. A special class of timing constraints originates from channels assuming time dependent synchronization, see I.7.2.2.

I.7.4.4 Reliability, safety and security

Requirements to reliability may impact the hardware structure in several ways:

– Fault tolerance means redundancy: at least two hardware units, and facilities for error detection, diagnostics and switch-over are needed to implement fault tolerance.

– Fault sectioning means to distribute the functions over separate hardware units in a way that limits the number of SDL processes that may be blocked by a single hardware error.

– Fail-safeness means that the system always must fail to a safe output state where it does no harm to its environment. Some sort of supervisory hardware will normally be needed.

Security against unauthorized access to or modification of information may also demand some special measures in the hardware.

I.7.4.5 Modularity and reuse

Hardware design, more than software design, must take production cost into consideration. This will depend both on the complexity of the hardware itself and the production volume. A high volume will mean lower unit cost. One therefore needs to minimize the number of different designs in order to take advantage of a high production volume. One will often be able to find a generic hardware design that can be used to implement a wide range of SDL systems.

I.7.4.6 Hardware architecture

From the considerations above, the overall hardware architecture needed to implement the SDL system should be defined. This may be documented by means of hardware structure diagrams, as illustrated in Figure I.7-3. The emphasis at this stage is the overall hardware structure in terms of computers, peripheral equipment, and communication channels.

The protocols, the signal formats and the synchronization schemes used on all physical interconnections should be defined, as this is important input to the software design.

Finally, the allocation of SDL processes to physical units should be documented. Once this is done, we know the functionality as well as the physical environment of the software in the system.

I.7.5 Software architecture design

I.7.5.1 Design principles

There is considerable semantic differences between SDL and most (if not all) programming languages:

a) **Concurrency** – Sequential programming languages like C and PASCAL give no support to the concurrency of SDL. Some languages like CHILL and ADA support concurrency, but do it differently from SDL.

b) **Time** – Very few programming languages support time. SDL-like time is not directly supported by any language, except possibly by CHILL.

c) **Communication** – SDL-like signal communication is not supported by any language.

d) **Sequential behaviour** – An SDL process graph specifies state-transition behaviour in the fashion of an extended finite state machine. Programming languages specify action sequences.

e) **Data** – SDL data are abstract and possibly infinite. The implementation in a programming language have to be operational and finite.

A usual way to overcome such differences is to adapt the underlying machine and programming language to the SDL semantics by means of support software. Three levels of support are in common use (see Figure I.7-6):

a) None – SDL concepts are mapped directly to concepts supported by a sequential programming language.

b) Basic facilities to support concurrency, time and communication provided by a real time operating system – This also includes the run-time support provided for concurrent languages like CHILL and ADA.

c) Additional support for SDL concepts on top of b).

Although the use of an operating system is the most common approach today, there are cases where the overhead this introduces is unacceptable either because of speed constraints, memory size constraints or cost.

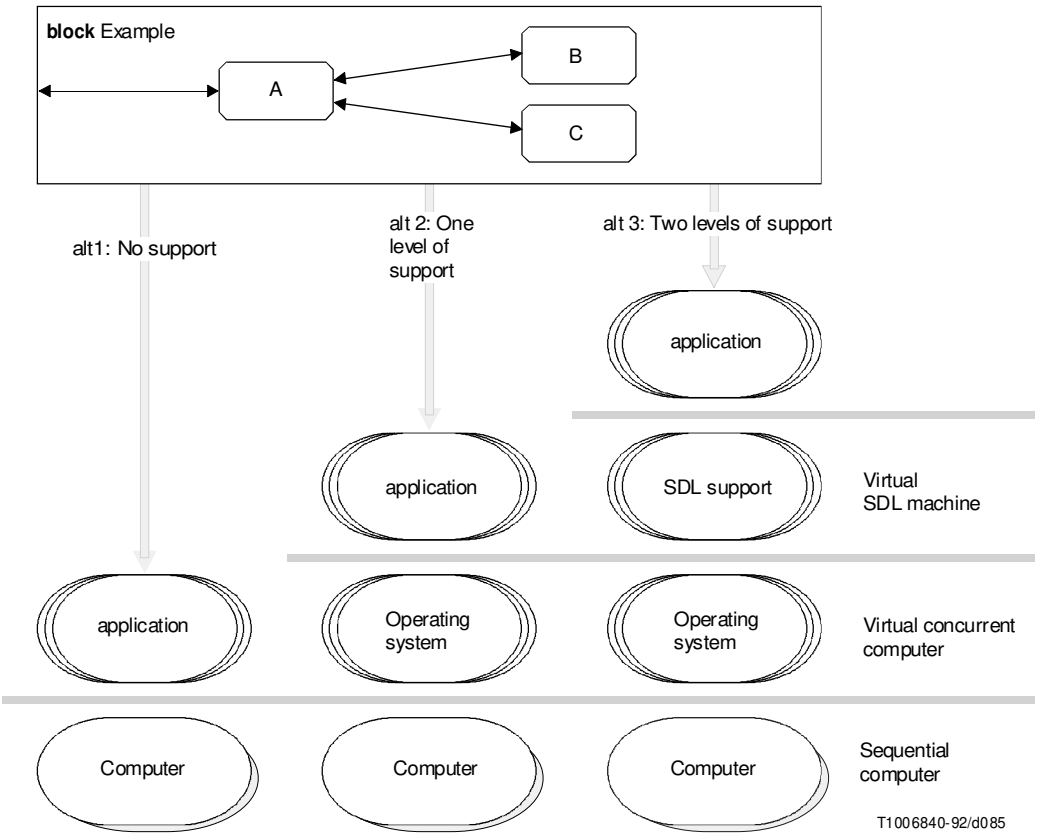


FIGURE I.7-6/Z.100

Alternative levels of support software

I.7.5.2 Concurrency and time

Levels of support

When several SDL processes are implemented on the same computer using a sequential programming language, such as C, the concurrency of SDL must be approximated by a sequential behaviour. This can in principle be achieved by transforming an SDL specification containing several processes into an equivalent specification containing only one, and then implementing this process as a sequential program. Due to the problem of "state explosion", and the lack of modularity in this approach, it is only practical in very restricted cases. Therefore, one will normally seek an implementation that retains the original process structure of the SDL specification. This implies that there must be some support for the scheduling of processes and their communication.

In a purely sequential language, there is no general support for concurrency. But procedure calls provide a combined communication and scheduling mechanism that can implement special cases of SDL communication and concurrency. (The asynchronous communication of SDL is then implemented by means of synchronous communication.) The lowest level of support is therefore to use procedure calls as the basic scheduling and communication mechanism. This approach is outlined in I.7.5.3.

In order to implement more general SDL systems, it is necessary to introduce a buffered asynchronous communication scheme. This can be achieved within the framework of a purely sequential program system using, for instance, a main program to schedule the activation of procedures implementing SDL processes that communicate through message buffers. The limitation of this approach lies in its ability to handle time and real time constraints.

A general purpose operating system that schedules concurrent processes according to priority, and supports SDL like communication and timing will provide the easiest and most general platform to implement the concurrency of SDL systems. The basic facilities needed from a real time operating system are:

- Multiplexing and scheduling of processes, i.e.
  1) context switching between processes;
  2) scheduling with pre-emptive and non pre-emptive priority to meet response time requirements;
  3) interrupt handling to provide passive waiting on external events.
- Synchronization of interactions, i.e.
  1) communication;
  2) access to shared resources.
- Time measurement.

These facilities are provided in one way or another by many commercial operating systems. They are also supported by concurrent programming languages such as CHILL and ADA. A small operating system that support these functions will be described in the following to provide an example and a frame of reference.

On top of the basic operating system, one will need facilities for SDL like timing and communication to provide general support for SDL.

An operating system example

The operating system sees the software system as a collection of *software processes* and general *semaphores.* It performs multiplexing and scheduling of the processes on the basis of external and internal events. These events are either external interrupts (including time interrupts) or internal operations on the semaphores (see Figure I.7-7).

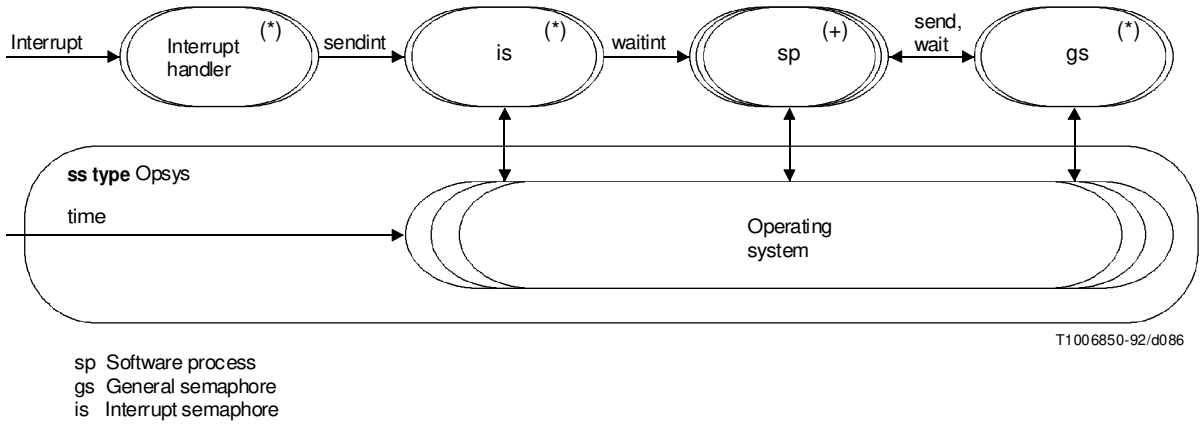


FIGURE I.7-7/Z.100

An operating system example

The semaphores administer buffers and waiting processes. Buffers are used both for message communication, and to represent shared resources allocated by a semaphore. The buffers are also shared resources themselves. Before a message may be sent, a free buffer must be allocated from a freepool. The general semaphore is a mechanism that can be used both to allocate free buffers, possibly representing some other resources, and to provide asynchronous communication.

Operations on the general semaphores are:

- **send**(semaphore, buffer);
- **wait**(semaphore, max-time) –> (buffer, time).

The *wait* operation can specify a maximum waiting time. If no buffer is made available before the time expires, the operation will return with a timeout indication. By waiting on a special semaphore, *suspend*, that never returns a buffer, the process may suspend itself for a specified time.

A special type of semaphore is used to signal interrupts and to wait on interrupts:

- **sendint**(semaphore);
- **waitint**(semaphore, max-time) –> (time).

The general semaphore can be seen as an abstract data-type with two operations, *send* and *wait*, implemented by procedure calls. It will use a data structure, e.g. a linked list, to hold a queue of buffers. It will also keep a queue of references to waiting software processes in the case when there are no buffers in the buffer queue (see Figure I.7-8).

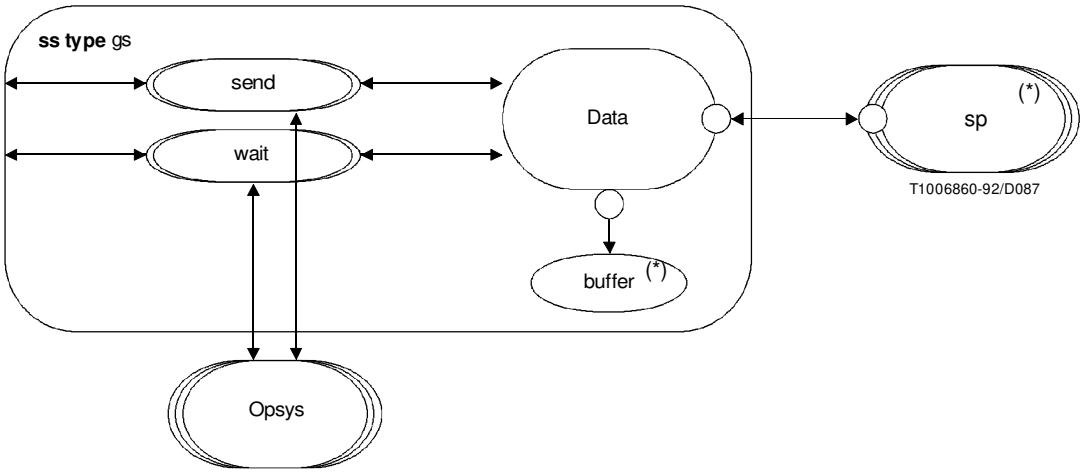


FIGURE I.7-8/Z.100

The general semaphore

This kind of semaphore is a generalization of the classic semaphores described in literature [17]. The messages/free buffers are put into the queue by a *send* operation, and fetched by a *wait* operation.

In order to implement SDL-like communication, the buffers may be encoded with SDL signals.

SDL timing is different from suspension of processes, since an SDL process may be active doing transitions, while a timer is running. To support this, the operating system provides an SDL-like timer facility:

- **starttimer**(time, timer-id, PId) –> message(PId, timer-id);
- **stoptimer**(timer-id, PId);
- **now** –> time.

The timer facility basically operates on a list of timers, with the corresponding processes waiting to receive a timeout message. By using *starttimer,* a process in reality orders the timer facility to make an entry in its list of timers and start counting and testing against the current time. When the specified time is reached, the timeout message is sent and the entry is taken out of the list.

The *stoptimer* will in a similar way order the entry to be taken out of the list. What will happen if a *stoptimer* is issued, just after timeout is sent to the process? The process will then be in a new state in which it will not expect the timeout, while in the timer list there will be no counter to be taken out. The best solution is to let *stoptimer* remove the timeout message from wherever it is in this case.

Each software process will have a process description that the operating system uses during scheduling and context switching. Some information in the process description are shown in Figure I.7-9

The scheduler will keep a list of processes that is ready to run, and always start the one with the highest priority. (In case of equal priorities, it may choose to alternate.) Whenever a *send* or *wait* operation is performed, some process may become ready to run, and/or the running process may have to wait. Each semaphore operation therefore may cause a new process to be activated and the running process to be stopped. Interrupts may have the same effect through the use of *sendint* and *waitint* operations. In addition, time interrupts may cause timeouts.

In addition to the basic set of operations described so far, operations to dynamically create and delete software processes and semaphores may be needed in some applications.

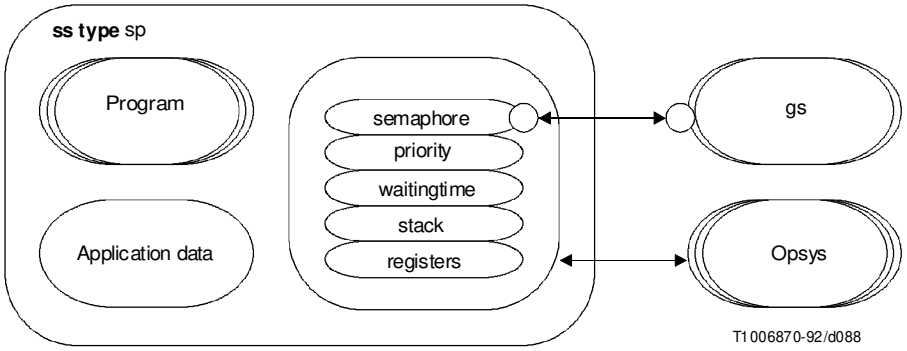


FIGURE I.7-9/Z.100

A software process

Priority

In SDL there is no priority among processes. But this does not mean that priority cannot be used in the implementation. Priority is needed for several reasons:

- there are external real time constraints that only can be met by giving high priority to certain processes;
- there are internal error situations that should be treated with the highest priority;
- there are situations where priority helps to simplify and speed up computation.

Normally, real time constraints must be met by giving high priority to the time critical processes. Therefore, one will normally have to treat time critical input/output at high priority, while the internal processing is allowed to proceed at a lower priority. This is one reason why input/output should be separated from the application processes.

In error situations one must react quickly in order to isolate the error, reduce the damage and give alarm. This means that some processing needs to be done immediately, and that the error should be reported as quickly as possible. In order to report fast, it is not sufficient that processes have high priority. We need priority on the message transfer as well. This can be achieved in the operating system either by giving each message a priority attribute, or by sending high priority messages through semaphores that are served with high priority.

SDL-services operate in quasi-parallel and communicate by means of priority signals. This can be implemented by giving equal priority to services, and let their internal signals have priority over external signals.

If internal messages within a software system have priority over external messages arriving from the environment, the internal processing order will become more deterministic. This helps to reduce the number of ways that processes actually will interleave. In turn, this reduces the probability that certain interaction errors will occur. More important, it sometimes helps to reduce the overhead and increase the speed of internal communications. Finally, load control is improved if service requests already accepted take priority over new service requests.

I.7.5.3 Communication

In the trade-off between hardware and software (see I.7.4) we started by looking at the physical interfaces to the system and worked our way towards an internal architecture. In software design we will do likewise by starting at the hardware-software interfaces and work towards an internal software structure.

The communication format used at the physical interfaces has a strong influence on the overall software structure. The external and internal communication needs also affect the way communication may be implemented within a software system. It also affects the needs for scheduling, preemption and synchronization.

Input/output

Conversions are often needed at the hardware software interface, not only to convert data from one format to another, but also to convert between continuous values and value sequences. One may, for instance, need to scan external variables at intervals in order to detect events and generate SDL-like messages for the internal communication.

The true concurrency and timing restrictions present at the software-hardware interface has to be matched by quasi-parallelism and priorities in the software system. High priority is normally needed on input/output-interactions in order to

- ensure that all input events are detected in the case of speed dependent interaction;
- make efficient use of slow input/output-channels; and
- reduce response times.

Interrupts are the basic means to provide pre-emptive priority and to enable passive waiting on external events. Passive waiting saves computing resources, but interrupts introduce a non deterministic execution order into a software system, which makes the conditions similar to truly parallel processing. Consequently, synchronization is necessary on the internal interactions between interrupted and interrupting software units (hence the interrupt semaphores).

Because of the priorities and speed needed to meet real-time and performance requirements, input/output must often be implemented by separate software processes as illustrated in Figure I.7-10.

This serves to hide the particularities of the input/output interface from the application part implementing the SDL processes. Implementation of SDL processes is discussed in I.7.5.4.

In Figure I.7-10 SDL-like communication is implemented by message transfer through semaphores. But this is not the only possible way.

Procedure calls

Procedure calls can be used to implement SDL signals directly. Each signal type may be represented as a procedure belonging to the receiving process. The signal “Open_door(here)” for instance, can be implemented by the procedure OPEN_DOOR(HERE). This implies that the receiving SDL process is implemented by a set of procedures and data.

As an example, consider three SDL processes *P1*, *P2* and *P3* represented in Figure I.7-11.

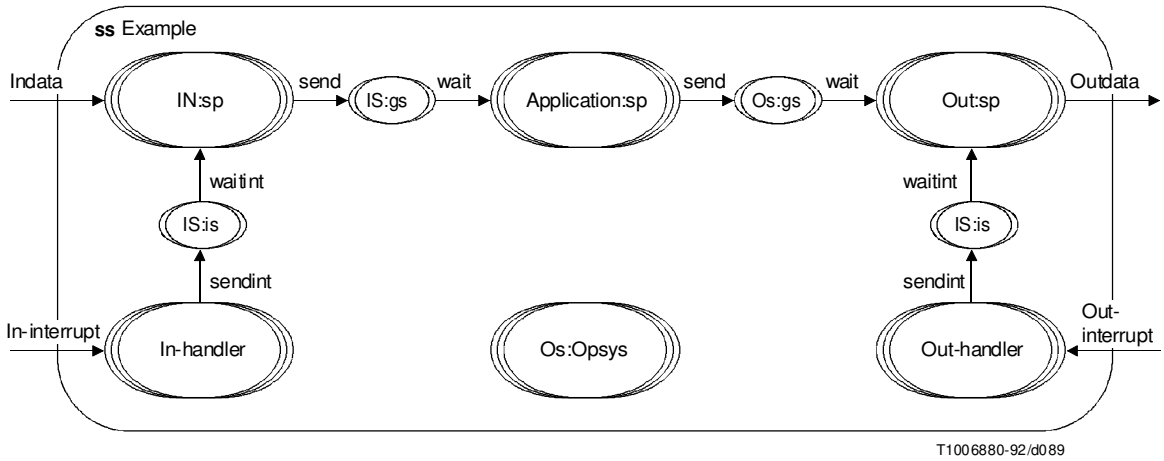


FIGURE I.7-10/Z.100

Implementing input/output by separate software processes

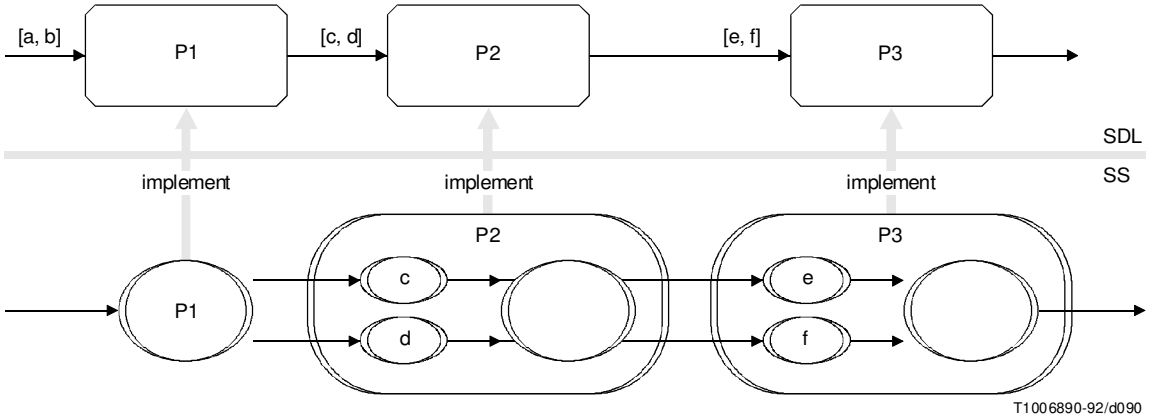


FIGURE I.7-11/Z.100

Signals implemented as procedures

For each process, there may be one procedure for each input signal, or there may be one common procedure with the signal type encoded as a parameter. In either case, the activation of processes will follow the procedure calls. When a signal is sent, the receiver will take pre-emptive priority over the sender and finish its transitions before control is returned to the sender.

If this solution shall work, the SDL system must be structured and behave like a procedure call tree. There must be sufficient time between each input signal or timeout to do all the processing and waiting that must be carried out in response. There must be no need to give pre-emptive priority to processes apart from the priority that follows implicitly from the call tree. For each signal sent down the tree, not more than one reply signal must be returned (implemented as a procedure return value). In practice, all non-deterministic waiting for "new" input and timeouts should be performed by one process. The role of the other processes is to perform the secondary processing and output that follows from a new input.

Procedure calls can be seen as special case of synchronous communication, where the scheduling is such that the receiver takes preemptive priority over the sender. This puts a rather severe restriction on the use of SDL. Hence, this simple and fast solution will not work in all cases. SDL communication is more general and flexible than procedure calls.

Buffered communication

The SDL synchronization is infinitely elastic, meaning that the sender may be infinitely many signals ahead of the receiver. As we have stated earlier, this situation cannot occur in reality. In practice, the queue must be limited, and in the case of a full queue, the sender will either have to wait or signals will be lost. Careful engineering is needed on this point to ensure a smooth load control without severe performance degradation in overflow situations.

The communication scheme depicted in Figure I.7-12 uses the general semaphores introduced in I.7.5.2. SDL signals are encoded in buffers and passed as messages through semaphore *S*. Free buffers are managed by another semaphore *F*. The scheme is quite general and has been used to implement many real time systems specified with SDL. Two aspects need consideration:

– *Load control* – The circulating buffers provide an opportunity to perform a simple form of load control. In overload situations the message queue between the sender and the receiver will grow until all free buffers are used. This prevents the sender from generating more messages before the receiver has managed to process some of the messages already in the queue. By carefully dimensioning the freepools, one can control the internal load in a software system.

– *Deadlock possibilities* – When several processes compete for the same resources, deadlock may be possible. Consider an overload situation where the freepool in Figure I.7-12 is empty. If the receiver now needs an additional buffer from the freepool, the system may deadlock with both processes waiting for an empty buffer. One should therefore prefer to use separate freepools to avoid this problem.

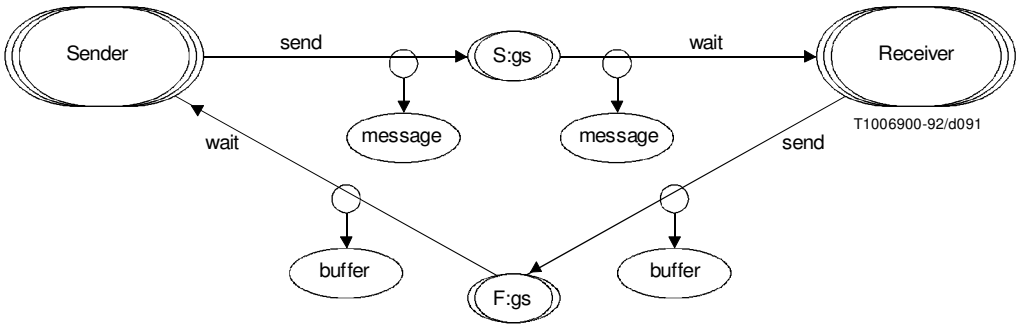


FIGURE I.7-12/Z.100

Communication by message buffers

In Figure I.7-12, messages are moved by transferring pointers and the buffer size is determined by the pool of free buffers. A different scheme is to keep the buffers inside the communication semaphore and copy the messages. When overflow occurs, the send operation must delay the sender.

The drawback with buffered communication is that it is slower, in most cases, than direct procedure calls, and that it is not supported by many programming languages (CHILL is an exception). One therefore has to acquire the additional software needed to support buffered communication, and to have computing resources for the additional processing. The primitives for asynchronous communication provided by many real-time operating systems will often be sufficient.

As explained in I.7.2.2, there are two classes of information one wants to communicate: continuous values, and value sequences.

Communication through a single buffer, i.e. a shared variable, is the most direct way to convey continuous values. The producer may set the value when it should be changed, and the users may read it when needed (see Figure I.7-13). Reading a current value can only be done when the user is active and executes a read operation. There should be no waiting involved, because the user wants to know the value at the reading instant.

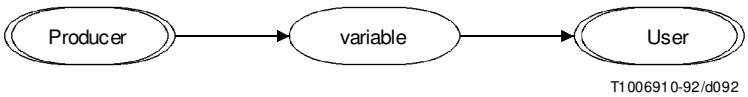


FIGURE I.7-13/Z.100

Communication through a shared variable

In case the producer and user are concurrent processes, the write and read operations must be mutually exclusive. This may be achieved in several ways:

a) Ensuring that the read and write operations are atomic with respect to each other:

   1) use primitive instructions;

   2) turn off interrupts;

   3) schedule such that sender and receiver never interrupt each other.

b) Controlling the access to the shared variable by means of a resource allocator. The producer and the receiver must ask the allocator to get the access right before they start an operation on the shared variable, and return the access right after the operation is finished. This can be achieved by *wait* and *send* operations as illustrated in Figure I.7-14.

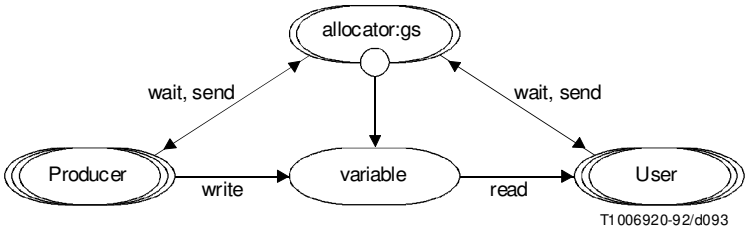


FIGURE I.7-14/Z.100

Communication through a shared variable controlled by a resource allocator

Routing

When an SDL process sends a signal, this must somehow be routed to the receiver. SDL signals identify their sender and receiver by PId values and the signals are routed by the blocks on the basis of *who* the receiver is, not *where* it is, except possibly when the route is specified by the **via** clause. At the abstract level, this is sufficient, but in the realization we must know the physical location of the receiver in order to route the signal correctly from the sender to the receiving process.

The physical process location is determined by the physical computer, the software process inside the computer, and possibly the local address inside the software process.

It should, however, be a goal to keep the software as independent as possible of the physical location of processes. A process should know as little as possible about the path a signal is routed through to get to a destination. Ideally, it should only know *who* the receiver is, not *where* it is.

To some extent, this is solved by the concept of PId variables. The data structure of a process will contain PId variables representing the processes it can send signals to. These variables are bound either at process creation time or dynamically during process behaviour. Thanks to the concept of abstract data types, the actual representation of PId values can be hidden from the process code.

But the implementation of the PId data-type will depend on how PId values are represented. How the PId values are generated and represented, is not defined in SDL. This is left to the designer to decide. One of the major design questions is therefore how PId values shall be represented and allocated.

One common solution is to represent PId values (signal addresses) by an identifier for the process type and another for the process instance. But this will be a logical and not a physical address. We will therefore need some support to map the logical addresses into physical locations.

In Figure I.7-4, there is a separate software block, called *Message routing*, that performs the actual routing on the software process level. Its purpose is to hide the physical addresses from the application processes, allowing messages to be routed on the basis of logical destination addresses. Hence, an SDL process need not know where other processes are located. It is sufficient to know their logical identifiers. Moreover, the implementation of the PId data-type is independent of the physical address structure. The routing system will use the logical identifier and an address map to select a semaphore through which the message is sent (see Figure I.7-15). Thus, knowledge of physical routing is centralized to the address map.

If there is no need to hide the physical addresses, a separate routing system is not needed.

One advantage of this approach is that it is simple to re-route messages in cases of on-line system extensions.

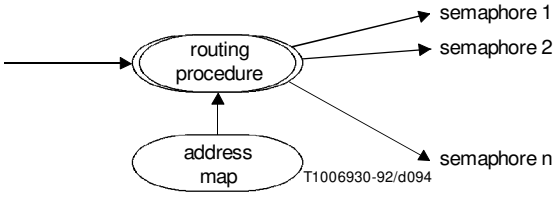


FIGURE I.7-15/Z.100

The routing procedure

I.7.5.4 Sequential behaviour

In this subclause we first consider the implementation of finite state machines (FSMs) in general. Thereafter we look at the special features of SDL, i.e. services, procedures, decisions, and dynamic process creation.

The FSM behaviour of SDL processes may be implemented in many ways. One is to encode the control part directly as **if-then-else** expressions or **case** expressions in the programming language. Another is to encode the control part in state transition tables. In between those, there is a whole range of intermediate solutions.

Direct code implementation of finite state machines

In this approach we may distinguish between two ways of representing the state:

– by a position in the program text;

– by a value stored in a variable.

Let us first consider the program position case. The principle is to encode the state transition diagram directly in the programming language. Since the diagram is a good definition of the logic, a direct mapping should be sought. Consequently, one should not try to make the diagram into a **goto**-less program, but use **goto** to get to the next state.

A typical implementation is the following:

```
begin
    state-1: wait(input);
        case input of
            signal-a:       call Action-1; goto state-3;
            signal-b:       call Action-2; goto state-5;
            else:           call Action-3; goto state-1;
        end case;
    state-2: wait(input);
        case input of
            signal-c:...
...
end
```

Because of the execution speed required, this may be the preferred solution for input/output processes. If decisions may affect the next state, the necessary "goto state" statements must be added to the "Action" procedures. SDL procedures may here be supported in a straightforward manner by using procedures in the programming language.

Since the state is represented by the program position, this technique implies that each SDL process is implemented as a separate software process, e.g. a CHILL process. This may be too space consuming if the number of processes is high.

Alternatively, the state may be stored in a variable. This will typically lead to a program that waits for input in only one place:

```
repeat forever
begin
    wait (input);
    case state of
        state-1:
            case input of
              signal-a:      call Action-1; state:= state-3;
              signal-b:      call Action-2; state:= state-5;
              else:          call Action-3; state:= state-1;
            end case;
        state-2:
            case input of
              signal-c:...
            ...
        end case;
    end;
```

If decisions may change the next state, the next state assignment must be moved inside the Action procedures above. SDL procedures can easily be supported here too as long as one can accept to wait for input inside the procedures.

This approach may be extended to handle many SDL processes inside a software process simply by introducing an array of states indexed by the process identifier. Multiplexing many SDL processes onto one software process may help to reduce the space and time overhead associated with the operating system. Internal communication between SDL processes running in the same software process can be considerably faster than communication involving synchronization and context switching between software processes.

A disadvantage is that the implementation of general SDL procedures becomes a bit more difficult. The problem is to keep track of return addresses when the SDL procedures contain states. If the program is shared between many process instances, this has to be handled explicitly using a stack of return addresses.

In the case of communication by procedure calls, the state has to be stored in a variable. In that approach, there will be a procedure corresponding to each input signal to the process. Each procedure must first test on the state, and then perform the corresponding transition. General SDL procedures need special attention in this approach too.

Table driven implementation of finite state machines

Since the process model of SDL is based on the extended finite state machine (EFSM) model, all SDL processes share the common characteristics of extended finite state machines.

Such machines are well suited for table driven implementation. Their behaviour may conveniently be defined in a state transition table, which is equivalent to the graphical form used in state transition diagrams. Such a table can be implemented by a two dimensional array indexed by the current state and the input signal, where each element specify a next state and an action to be taken for each combination of current state and input (see Figure I.7-16).

	Signal	
State	Signal 1	Signal 2
State 1	State 2 / Action 1	State 3 / Action 3
State 2	State 3 / Action 5	State 1 / Action 2
State 3	State 2 / Action 3	State 1 / Action 2

FIGURE I.7-16/Z.100

A state transition table

One may easily design a general program that uses such a table to determine the next state and the action, given a current state and an input signal. This is the general idea behind table driven implementation of FSMs.

Normally, there will be many open cells because one does not expect all input signals to arrive in all states. It is therefore better to implement the table in a more space efficient way.

One way to design table driven implementation of extended finite state machines is outlined by the software structure diagram in Figure I.7-17. It contains a general program called *FSM-support* which interprets a data structure *ST-table* representing the state transition table of the finite state machine.

The software described in Figure I.7-17 provides a concurrent implementation platform for processes described as extended finite state machines. Scheduling is performed on the basis of input messages. Each message contains an address that identifies the receiver process. The *FSM-Support* program will use the address to activate the process by selecting the appropriate *Process-Type* and *Process-Data*. The *Process-Data* holds the current state and the optional data objects that an extended finite state machine may have in addition to the state (i.e. the *extension* of the pure FSM). The input message carries a signal type name, which is used to select a transition. For each input message, the addressed process is allowed to perform one transition. It is therefore a quasi-concurrent implementation, where the scheduling of processes is determined by the address in the input message.

In short, *FSM* exploits the fact that the support program may be reentrant for all FSM-processes, and the *ST-table* is reentrant for all processes of the same type. If there are many instances of the same process type, there must be one *Process-Data* item for each, but the *ST-table* and *Actions* may be shared. If there are many process types, each must have its own *ST-table* and corresponding *Actions*.

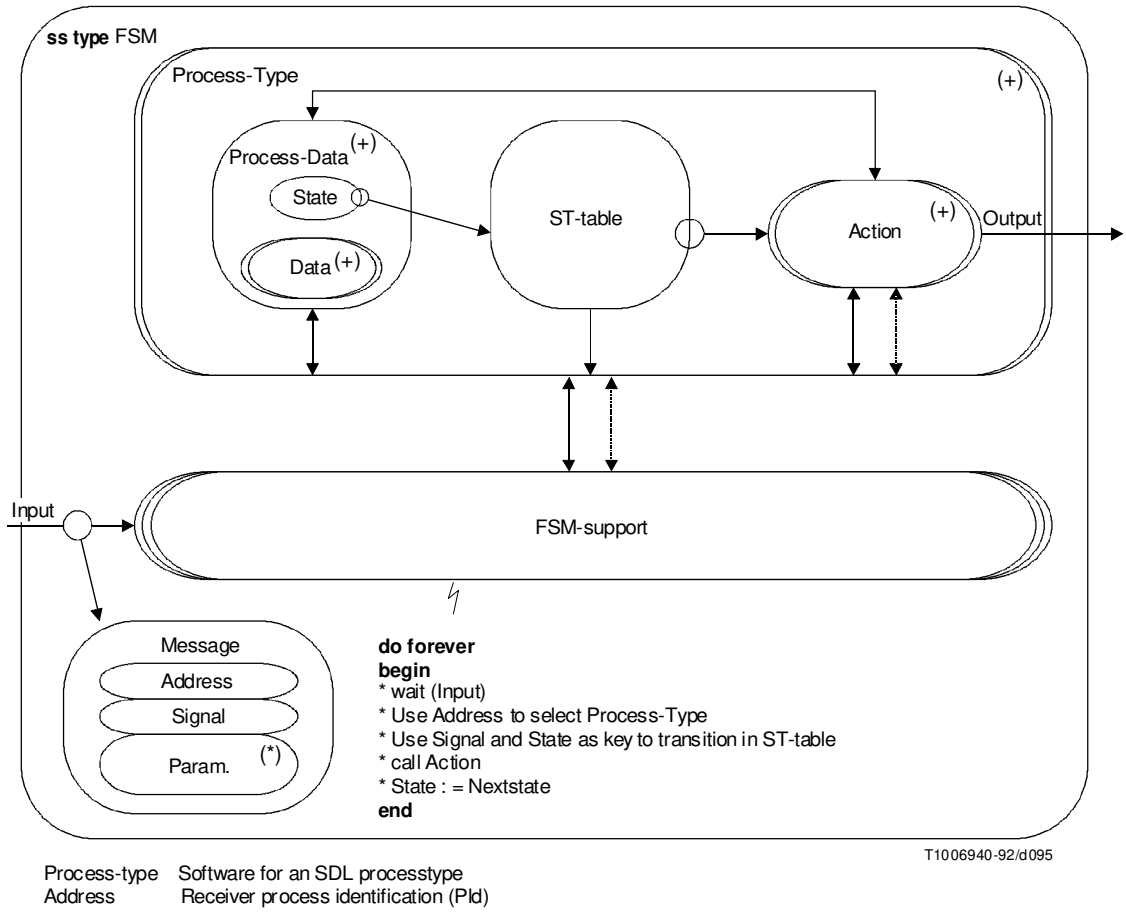


FIGURE I.7-17/Z.100

Support software for extended finite state machines

In the *ST-table*, each state is represented by a record that contains a (variable) number of transition records, one for each transition from the state (see Figure I.7-18). Each transition record specifies the input signals that may cause the transition, the corresponding action and the next state. See the example in Figure I.7-19.

The *FSM-support* program will wait for input messages, and respond to the messages in the order they arrive. For each message it will access the state record in the *ST-table* referred to by the current state in *Process-Data*, and search there for the transition record that corresponds to the input message. If a transition record with the same signal name as the input message is found, this transition is selected.

If not, the transition for unspecified reception, marked with a "*", is selected (see Figure I.7-18). The *FSM-support* will then perform the *Action* procedure referenced in the selected transition record, and, finally, assign the next State value to the current State. This ends the transition, and the *FSM-support* will wait for a new input message.

When a new message arrives, the next transition will be performed in the same manner.

Actions may, in principle, either be encoded as interpreted code or directly executable code in the programming language. We shall use the latter approach here. Hence, interpretation is limited to the state transition table which represent the control part. The action part is then performed by calling the action procedure referred to in the transition record.

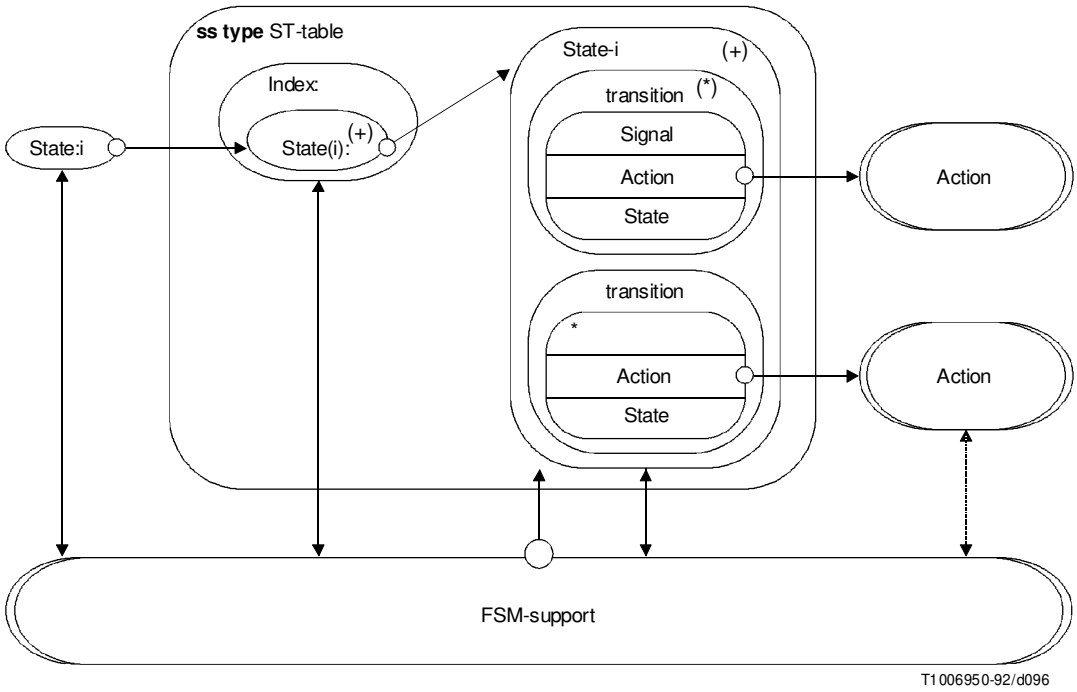


FIGURE I.7-18/Z.100

ST-table definition

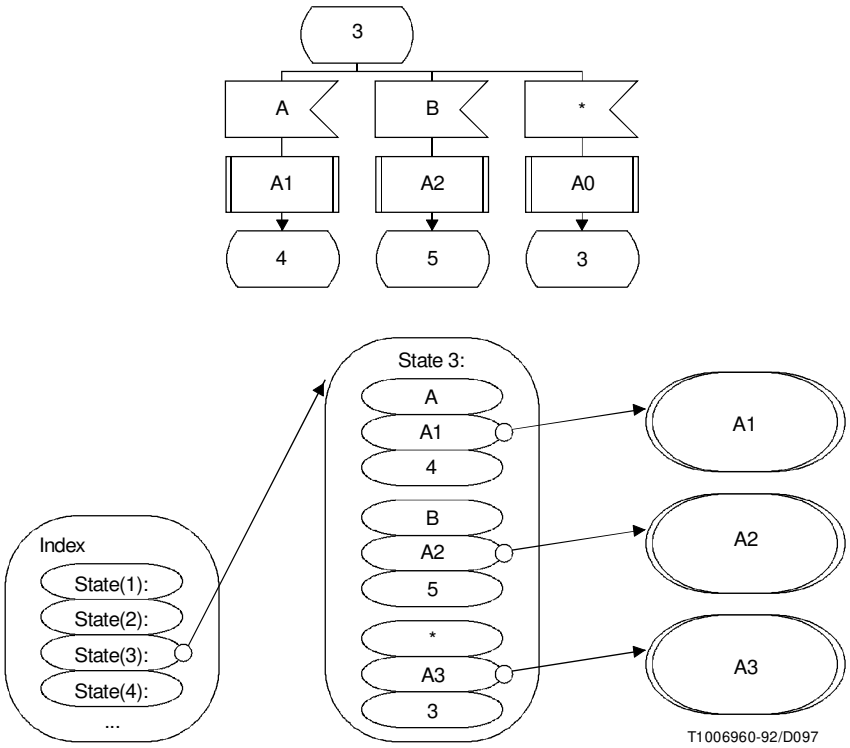


FIGURE I.7-19/Z.100

Example showing how one state is encoded in the ST-table

Normally, a number of distinct operations shall be carried out during a transition, i.e. sending output signals, starting and stopping timers, assigning values to data objects. The *Action* procedures perform all these operations. Each operation will normally be implemented as a procedure. Hence, the *Action* procedure mainly consists of procedure calls to operation procedures. In addition, there may be simple assignments and expressions corresponding to the data operations specified, e.g.:

```
procedure action-1
begin
    send-signal(open_door, here);
    send-signal(door_bell, here);
    starttimer(now+waiting_time, floor_delay, self);
    table:= cancel_res(table, moving, here);
end;
```

It can be seen that the operation procedures may be designed to constitute a language level which is very close to the SDL specification[6]. This simplifies the effort needed to generate and to read action code considerably. In general, the procedures will correspond to the operations that an extended finite state machine may perform:

- sending signals to other processes or external equipment;
- operations on local data; and
- setting, resetting of timers.

The format of signals need to be standardized for each particular implementation. It is not sufficient that the signal interfaces are correct at the functional design level; they must be compatible at the concrete level as well. This means that communicating processes must encode signals in the same way.

Messages from one process to another running on the same instance of *FSM-support*, may be given priority over external messages by sending such messages through an internal queue. This will speed up internal communication since one avoids external synchronization and calls to the operating system (**wait**, **send** operations). Messages to processes outside are passed on to external communication semaphores, possibly through a routing procedure.

The table driven solution has several virtues:

a) **It facilitates implementation and documentation** – The coding of control structures, which normally is difficult and error prone, is done easily and reliably in the *ST-table*. The transition actions are usually structured into small and well defined procedures. A typical size is 10 lines of code, mainly calls to common routines for signal sending, timing, and data operations. It is easy to see the relationship between the functional design and its software implementation in *ST-tables*. So easy that the encoding may be done by non-programmers. An easy to follow cookbook can be made.

   The name of the *Action* procedure may be entered as a comment in the SDL process graph in order to provide an easy reference from the SDL specification to the actual code.

   If the code is reasonably well structured and commented, the SDL process graph together with the source code listing is sufficient documentation. It is normally the case that people really use the SDL diagrams as the main documentation, and only go down into the code when errors have to be debugged. Thus, the approach stimulates people to keep the functional design updated.

b) **It enhances modularity and eases maintenance** – The standard *FSM-support* program itself can be used in many systems with different applications. Thus, it is a highly reusable module. It is also more easy to reuse processes in new applications, when they are implemented in a standard way.

---

6) This is not particular for the table driven approach. Direct code implementations may use a similar approach to the action code.

Moreover, modifications are easy to make and their consequences possible to control. Normal practice is to first modify the process graph (*ST-table*) and then add the *Action* procedures that are needed. The trick is to avoid side effects if an *Action* procedure is called from several transitions. One should either keep a backward reference from *Action* procedures to transitions, or use a separate *Action* procedure for each transition to achieve this.

c) **It is reliable** – The standard *FSM-support* program will be thoroughly tested in many applications and is therefore likely to be very reliable. The application is also likely to be reliable since it is directly derived from, a presumably correct, functional design.

d) **It is robust** – Robustness is partly due to the message communication mechanism, and partly due to the table driven approach. When signals are passed as messages, the receiver may always check the input before it is used. Thus *FSM-support* may check incoming messages for consistency before they are accepted. Moreover, the "*" transition in the *ST-tables* ensures that all non expected inputs may be received, and handled in a proper way.

e) **It supports testing** – The *FSM-support* may be implemented in a test version that allows the tester to simulate input messages, and to trace the transition Actions and output messages that are generated. Thus a process may be conveniently tested in an environment that looks exactly like the real one to the process itself.

Admittedly, the *FSM-support* program introduces overhead. Searching for transition records may take some time. To reduce this overhead, one should arrange the records in the order of the most frequent signals.

When all aspects of an application are taken into consideration, it often turns out that the time overhead is negligible. When implementing several processes by the same *FSM-support*, there may even be a net gain due to saved context switching in the operating system.

The *FSM-support* is a well defined candidate for speed optimization, if needed. One may alternatively encode the table as a two dimensional array, where the state and the input signal are used as indexes. This will give fast access, but may be too space consuming.

Since the *ST-table* encodes control structures in a space efficient way, one will save space in larger applications.

What are the limitations of this approach? Since other processes are blocked during a transition, the *Actions* should not wait on external events or perform very time consuming operations. For the same reason, processes implemented on the same instance of *FSM-support* should have equal priority. (They cannot interrupt each other.)

Implementation of SDL features

SDL has some features which are different from the ordinary finite state machines, i.e. decisions, save, procedures, services and dynamic process creation. These may all be implemented on top of the FSM-implementations described so far.

Decisions in the SDL specification can be encoded as states when the FSM-support technique is used. This is not strictly necessary, but it makes the Action procedures independent of particular states.

When there is a decision that affects the next state, treat it as if it was a state. Do the question operation before the decision state is entered, and send the answer as an input signal to be received in the decision state. In that manner, the branching on internal values and external signals are treated similarly. In order to speed up the decision making, and to finish decisions before new inputs, decision signals should be treated internally in *FSM-support*, that is, given priority over external signals. When a decision signal has been issued, the corresponding transition should be performed immediately. When a transition contains a decision that doesn't affect the next state, it should be encoded as an internal decision in the procedure of the transition *Action*.

The save mechanism implies that each process has a logical queue of saved signals. This queue may be implemented as part of the process data. A save symbol may then be encoded as a transition to the current state (next state := current state), where the action is to put the signal in the save queue. Whenever a process makes a transition to a new state, the signals in the save queue are moved back to the input queue and treated as normal input signals. Hence, save may be implemented by adding a save queue to the data of each process, and by moving input from this queue to the normal input queue at the end of each transition to a new state. Since this may be rather time consuming, save should be avoided in time critical applications.

The idea of a save queue is to keep some signals away until it is time to treat them. This can be implemented in other ways too, for instance using separate signal queues. The reader should think over the purpose of using save before implementation takes place, and look for solutions that avoid time consuming activities.

Procedures in SDL may contain states. In those cases, the procedure introduces a level structure into the process graphs. This is easily implemented in the direct code approach when SDL procedures can be directly mapped to programming language procedures.

But in the cases where several SDL process instances are mapped on one software process instance, this is not possible. In order to implement this case, each process will need a stack to store states, task addresses and data local to the procedures. When a procedure is called, the current context (state, task address and data) is pushed onto the stack, and the procedure is entered. Upon return, the opposite operation takes place. This is more simple to implement if procedure calls always occur at the end of a transition, because the return will be to a state and not to an arbitrary task address.

Services in SDL are basically finite state machines that execute in quasi-parallel, and use priority signals to communicate with each other. They are easily implemented within one software process using either the table driven or the direct code approach. In order to identify the service to receive a given input signal, the signal type name may have to be considered in addition to the PId.

In the case when there is one software process instance per SDL process instance, dynamic process creation require the support of the operating system. Within one software process, dynamic process creation amounts to the allocation and initialization of a record from free memory that can store the process data.

I.7.5.5 Data

Now we come to the non-trivial problem of implementing SDL data. SDL data have three aspects:

- the sorts, defined in terms of operators and equations;
- the instantiation of the sorts as SDL process variables; and
- the instantiation of operators specified in SDL process transition expressions.

In SDL specifications, sorts help to focus on functionality and to remove irrelevant details of the implementation. In the software implementation, the notion of abstract data-type helps to increase modularity and reuse. At this level, the main aspect is *encapsulation* and *information hiding* [18].

The idea is to encapsulate data in modules that only provide a set of well defined operators to the environment. This means that data structures cannot be accessed directly, only by invoking the operators, often implemented as procedures. In this way the internal data structures are encapsulated by the operator procedures and hidden from the environment (see Figure I.7-20).

The instantiation of operators specified in the SDL process graphs will therefore be implemented by calls to the operator procedures. This means that the transition *Actions* are independent of the particular data structure used to implement the data types.

Even if data are informally defined at the SDL level, the notion of abstract data-type is useful because it helps to achieve independence and modularity.

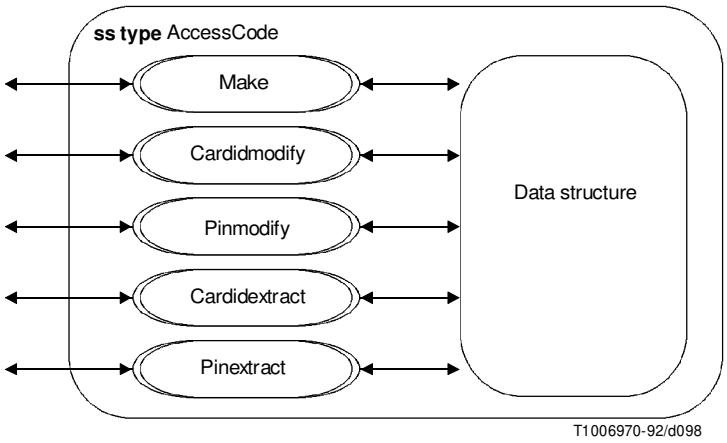


FIGURE I.7-20/Z.100

The implementation of a sort is an encapsulated module

The first step in software design for a sort is to specify its signature (the operator interface). Consider, as an example, the SDL sort *AccessCode*:

```
newtype AccessCode
operators
    Make             : Integer, Integer          ->     AccessCode;
    cardidModify     : AccessCode, Integer       ->     AccessCode;
    pinModify        : AccessCode, Integer       ->     AccessCode;
    cardidExtract    : AccessCode                ->     Integer;
    pinExtract             : AccessCode                  ->     Integer;
axioms
/* we have omitted them here */
endnewtype AccessCode;
```

The corresponding operator interface in terms of procedure definition headings may be:

```
Make(integer, integer)
Cardidmodify(accesscode, integer)
Pinmodify(accesscode, integer)
Cardidextract(accesscode)
Pinextract(accesscode)
```

The next step is to choose an appropriate data structure. The choice is very important since it will have a major influence on the algorithms used to implement the operators and hence on the performance. The choice will depend on non-functional design constraints. It will be possible to have several implementations corresponding to different design constraints, all reachable through identical operator interfaces.

In the formal SDL specification, the meaning of the operators may be defined by means of equations or operator specifications. However, there is as yet no general and straightforward way to turn an axiomatic definition into an efficient implementation. Consequently, we have to be pragmatic about the design of data structures and algorithms. We have a much simpler case when operator specifications are used, since an operator specification is similar to a value returning procedure (in fact, it is defined by a transformation to such a procedure).

First, we must consider alternative data structures, and select one that meets the performance requirements, e.g. a linked list. Next, we design a procedure corresponding to each operator.

The design approach of this example has been to define a procedure for each operator of the sort. Alternatively, one might consider to use a macro, or even a simple statement, if the sort is directly supported in the programming language. This will be the case for many of the predefined sorts of SDL. In real time systems, many data structures and algorithms are simple, so one may come a long way by using the predefined sorts.

In complex cases, more effort will be needed to do this part of the design. For sorts with complex data structures, it is useful to represent the logical data structure in conceptual data descriptions, e.g. entity relationship descriptions, as part of functional requirements or functional design.

The next step would then be to map the conceptual description into SDL data. Since the operator interface of data objects hides the internal data structure from the environment, a database management system may be used without the environment knowing it. In fact, the notion of abstract data-type can be successfully used for very elaborate data structures.

Abstract data types are supported by SDL, but the notion is not particular for SDL. In fact, the above guide-lines will be useful in any software design, regardless of specification method. They help to decompose a design into modules with low coupling and high cohesion.

I.7.5.6 Overall software architecture design

In previous subclauses we have looked at some principal design problems and solutions. In this subclause we shall see how they can be used in an overall approach to software architecture design. The goal is to find and document a software structure that will implement the SDL system and satisfy the non-functional requirements. The input information to software architecture design are

- the SDL specification;
- the hardware architecture design; and
- the non-functional requirements.

From the SDL specification we know the functional interfaces and the functionality of the software system. From the hardware architecture design we know the physical interfaces. From the the non-functional requirements we know time and performance restrictions.

The first step is to look at the internal and the external communication, and design a solution for each interface.

On one hand, we have the problems of the physical interfaces, on the other hand we have the internal communication needs. Start by considering the input/output interfaces and their requirements to response times, time measurement, event detection, priorities and synchronization. Then consider the requirements to internal communication between SDL processes. Is time critical? Can procedure calls be used? Should buffered communication be used?

At this stage, a decision can be made concerning the need to use of an operating system.

The next step is to link the interface modules to the internal process modules. This will lead to an overall software structure, such as the one shown in Figure I.7-4 or I.7-10.

The best way to implement the extended finite state machines defined with SDL depends on the design constraints. If the speed constraints allow, one should use the table driven implementation. This is a compact, flexible, and reliable solution. If processing speed is critical, one should consider to use the direct code implementation, and to implement signals by procedures.

The data part should be implemented in modules corresponding to sorts.

One may implement each SDL process in one software process, but this is necessary only when pre-emptive priority is needed. Another solution is to run several SDL processes on top of a single software process, e.g. by using the table driven solution outlined in I.7.5.4.

Take another look at the estimated mean execution times, see I.7.4.3. The estimates may now be improved. Re-calculate the mean peak load of the computer. If this figure is above the load limit, reconsider the design, i.e. either

- allocate SDL processes differently to computers;
- use a faster computer; or
- optimize the software for high execution speed.

If the load figures look fine, proceed by checking real time constraints against the maximum execution times needed to respond as required. When everything looks fine, proceed with the design of each SDL process, and support software.

Software processes that interacts by buffered communication provide a message interface to their environment and hide the internal structure. They are therefore relatively independent modules, convenient to handle and easy to integrate.

I.7.6 Hardware architecture design

We have already, in I.7.4, discussed the hardware architectural design. Hardware, in contrast to software, is fundamentally different from SDL. Hardware is physical. As such, it develops errors over time, needs time to perform tasks and is subject to noise.

There are conceptual similarities between SDL and hardware. Concurrency, for instance, comes naturally with hardware, and there is a long tradition for specifying and implementing hardware by means of finite state machines.

There exists hardware description languages, such as VHDL, that resemble programming languages. Therefore, up to a point, hardware architecture design is similar to software architecture design. It is a matter of mapping the SDL specification into a corresponding description in the hardware description language. This can be achieved by automatic translation tools in the same manner as for software. But the hardware designer is faced with a different set of constraints that makes the task different from software design.

Concurrency and time

On the circuit level, hardware components will often operate synchronously and interact synchronously using continuous values. These mechanisms must then be used to implement the asynchronous operation and asynchronous interaction with value sequences existing in the SDL system.

Therefore, either the SDL system must be restricted in a way that comply with the hardware mechanisms, or the hardware must provide the mechanisms used in SDL.

In principle it is possible to implement most SDL mechanisms in hardware. We can build asynchronous components corresponding to SDL processes communicating asynchronously through SDL like channels.

For technical and economical reasons, however, it will normally not be practical to implement general SDL systems in hardware. Therefore, it is more common to restrict the use of SDL in a way suitable for hardware implementation. This may mean to restrict the communication structure such that synchronous communication, without save, can be used.

In a synchronous system, the clock signal gives a natural time reference. Time measurements can then be implemented by counting clock ticks.

Sequential behaviour

There is no fundamental problem in implementing finite state machines in hardware. In fact, finite state machines were used for hardware design before they were used to design software.

Provided that the use of data are restricted to the data types supported in the hardware description language, there is no fundamental problem in the implementation of tasks and decisions.

Procedures may cause problems. In particular, if dynamic data allocation is needed to support them. Therefore, the use of procedures should be restricted.

Dynamic process creation, in the context of hardware, is unfeasible, if it means to create a physical unit. But it may be simulated by activating and passivating processes built into the hardware from the beginning.

I.7.7 Stepwise guidelines to implementation design

The main steps of implementation design are:

- Step 1: Trade-off between hardware and software.
- Step 2: Hardware architecture design.
- Step 3: Software architecture design.
- Step 4: Restructuring and refinement of the SDL specification.
- Step 5: Detailed hardware and software design.

Note that these are the main design steps only. In practice, iterations may be needed. While the main design steps rely on high level decisions which are likely to remain manual, the detailed design and implementation are subject to gradual automation.

Be communication oriented. Find solutions for the external interfaces first, and then the internal. Go from there to the process implementation design. Use the principle of generality; prefer to use the most general and flexible techniques whenever possible.

Step 1 – Trade-off between hardware and software

- Analyze requirements on physical distribution of interfaces and services. Select a physical system structure to support this. Minimize the bandwidth needed over channels covering physical distances.
- Consider the implications of performance requirements. Calculate the mean processing loads for each SDL channel, signal route and process. Allocate processes to computers such that the mean load on a single computer not exceeds 0.3 Erlang of its total capacity.
- Consider the implications of real-time requirements. Calculate the response times for time critical functions and check that requirements will be satisfied. Use priority to ensure fast responses. Isolate time critical parts as much as possible.
- Consider the implications of reliability requirements. Consider the need for redundancy. Add redundant units and restructure the system until requirements can be met.
- Consider the implications of safety and security requirements.
- Consider the implications of cost-effective production.
- Consider the implications of cost-effective procedures for modifications.

Step 2 – Hardware architecture

- Describe the overall structure of computers and other hardware units.
- Describe the physical interconnections between the hardware units.
- Describe the signal formats, synchronization schemes and protocols to be used on the physical interconnections to the extent they are not already covered in the requirement specification or the functional design.

Step 3 – Software architecture

- Map the SDL specification as directly and reliably as possible into the software implementation.
- Look at the physical interfaces and design software modules that can take care of the physical layer, i.e. synchronization, event detection, timing, format conversion.
- Look at the internal interfaces and choose a suitable communication mechanism for each, i.e. procedure calls, message buffers, continuous values.
- Use the most general and flexible communication scheme for SDL signals, i.e. buffered communication, unless certain that direct procedure calls are needed and will work.
- Prefer to use a general operating system that supports concurrent processes and buffered communication, except when a simple sequential program structure is obviously sufficient.
- Select the implementation method for each SDL process. Use general support systems whenever possible to ease the implementation of application functions, and to increase reliability.

Step 4 – Restructuring and refinement of the SDL specification

- Restructure and refine the high level SDL specification into a low level SDL specification, if necessary, so that this reflects the concrete system structure.

All functional properties of the implementation should be reflected in the low level SDL specification. This serves to ensure equivalence.

The SDL specification should reflect the concrete system structure. This serves to simplify the mapping from the SDL specification to the implementation.

Additional functions will normally be needed to support the concrete system. These functions will depend on design decisions, and cannot be defined before the overall design is made. While some of these functions will be invisible to the users, others will be visible.

Some functions that normally are visible to the user are:

- error handling, e.g. error reports, unavailable services;
- operation and maintenance of the realization, e.g. blocking units, testing units;
- access to limited resources such as printers.

Some functions that should be invisible to the user are:

- multiplexing of computers and channels;
- synchronization and mutual exclusion;
- communication services;
- load control.

Note that the restructuring does not mean that everything has to be redefined. A majority of the processes from the high level SDL specification may remain unchanged. If they are defined as separate types, it is a simple matter to put instances into a new structural context together with some new processes.

Step 5 – Detailed hardware and software design

- Design completely each hardware and software unit identified in previous steps, using standardized components and technology as far as possible.

I.8 Formal approaches to validation, verification and testing

I.8.1 Introduction

This subclause gives an introduction to formal approaches to validation, verification and testing. This is a broad area, subject to intensive research activities. It is expected that more detailed guidelines can be provided in the near future.

According to commonly accepted terminology, *validation* is concerned with the establishment of a correspondence between the informal requirements on one hand and the specification and/or implementation of a system on the other. In other words, the objective of validation is to answer the question “Is the system the intended one, will it fulfil the expectations from its future users?”.

Verification is concerned with the establishment of the *correctness*, according to some correctness criteria, of the specification and/or implementation of a system. In other words, the objective of verification is to answer the question “Is the system correct?”. Of course, one of the expectations of the future users is that the system is correct, thus verification can be seen as part of validation.

Testing is concerned with the establishment of *conformance*, according to some conformance criteria, of an implementation with its associated specification. Conformance criteria are normally expressed in standards by means of conformance statements, covering among other things a minimum set of features to be supported by conforming implementations. Testing can be seen as a technique to be used in both verification and validation.

The required properties to be validated/verified are usually classified in *safety* properties and *liveliness* properties. Safety properties are required to be satisfied in every instance of time in the specified system, e.g. absence of deadlock. Liveliness properties are required to be satisfied in some instances of time, e.g. “after connect request, the system must respond with either connect indication or disconnect indication”.

Since SDL, like other standardized FDT's, is mainly concerned with the behaviour of systems, this section focuses on the validation and verification of the functional properties, i.e. the specified behaviour in terms of signal communication. The validation and verification of non-functional properties, e.g. real-time requirements, performance criteria, etc., are not further considered here. Moreover, the scope is further restricted to the derivation of test cases; the procedures for testing are not covered.

One advantage of using a formal specification language, such as SDL, is the possibility to perform more precise validation and formal verification of specifications. The specification can also be used to derive test cases in order to test implementations of the specification. This is a direct consequence of the formal semantics of SDL, since the semantics defines an unambiguous interpretation and enables formal reasoning about behavioural properties of specifications.

The formal semantic model defined for SDL is a *denotational* model, which defines the behaviour of an SDL system in terms of an abstract machine that interprets the SDL specification. In such a model, the behavioural properties of the specification, important for validation and verification, are not expressed explicitly; instead, emphasis is put on the behaviour of the abstract machine interpreting the specification. This makes the current semantical model for SDL not directly applicable in the context of validation and verification.

An alternative approach is to use an *operational* model, see [19], which defines the behaviour of the specified system in terms of its observable behaviour using *Labelled Transition Systems* notation. A model supporting this view is given in [20] and [21], where the behaviour of an SDL system is defined relative to where the events are observed. For example, the events regarded as observable in the environment of the system are input and output of signals to and from the system, the events regarded as observable at the block level are the input and output of signals to and from the observed block, etc. One of the major benefits of such a model is the possibility to develop computer based tools which supports the validation and verification of specifications. Such tools are usually based on the technique of *state exploration*, see [22], where the behaviour of the system is computed and represented in terms of a *reachability graph* or *Asynchronous Communication Tree*, see [23]. Informally, the process of computing the reachability graph of a system can be described as:

> *Given an initial state for the system, all possible events that may occur in this state are performed, which results in a new set of states for the system. The states are connected to the initial state with a labelled edge identifying the event performed. This process is then repeated for each new state, until no new states can be derived.*

However, all tools based on this technique have the problem of *state explosion* in common. The number of states in the reachability graph exceeds the capacity of the system performing the computation of the behaviour. The problem of state explosion is an inherent feature of the intricate behaviour of communicating systems, rather than a flaw in a particular technique. For example, an SDL system has generally an infinite behaviour, and hence it is not possible to compute the full behaviour of the system. Nevertheless, there exist a number of different techniques to reduce the effects of the state explosion:

- Techniques that *reduce* the number of states necessary to generate

  1) *partial ordering* of events instead of interleaving, see e.g. [24];

  2) *abstraction*, by collapsing semantically equivalent states in the reachability graph into one state, see e.g. [25] and [28];

  3) *partial analysis*, by generating only a part of the possible behaviour, see e.g. [26];

  4) *partitioning*, by dividing the system into independent subsystems which are analyzed separately, see e.g. [27].

- *Performance* techniques for efficient storage of the reachability graph, see e.g. [26].

The different techniques can be combined and used within one tool.

I.8.2 Validation and verification

Step 1 – Analysis

- Ensure that the specification is syntactically and semantically correct by using conventional SDL analysis tool.

Step 2 – Simulation

- Validate/verify the normal behaviour of the specification by means of a simulation tool.

The purpose of this step is mainly to check that the normal behaviour described in the requirements specification is correctly covered by the SDL specification. This is accomplished by executing the simulation tool according to the prescribed "normal usage", for example described by message sequence charts, while checking that it is consistent with the SDL specification. The simulation will reveal the major design flaws (if there are any) and will also as a by-product help to detect some errors that were not detected in **step 1**.

Simulation tools give a possibility to execute an SDL specification, while interactively investigating the behaviour of the specified SDL system. Signals can be sent to the system and the system responses can be checked. Usually, it is also possible to force the system into some predefined state, for example by changing variable values or creating process instances, and continue the simulation from this state.

Simulation tools for SDL are commercially available from different vendors and must be considered to form a well-known technique for the validation and verification of SDL specifications.

Step 3 – State space exploration

– Check certain specific properties of the specification by means of a state space exploration tool.

The purpose of this step is to check that a given SDL system specification has certain specific properties. These properties can be general properties like the absence of deadlock and other general error situations, or they can be system specific properties defined by the designer of the SDL system. The analysis with the state space exploration based tool will reveal subtle design errors, involving for instance unexpected signal races or infrequent timeouts, that are very difficult to find using simulation or manual inspection. This sometimes also includes making some modifications of or extensions to the SDL system specification in order to reduce the state space necessary to explore. In practice, this might include limiting the number of instances allowed for some of the process types, limiting the length of channel queues or making a more or less detailed specification of the environment of the system.

For small SDL systems it is possible to verify the absence of a given property, but in general it is not possible due to problems with state explosion. However, even for large and complex systems these types of tools have been found to be very useful as automatic analysis tools, that perform a more extensive analysis than is possible with simulation tools.

Several state exploration based tools exist and are in use, but they are currently not commercially available and the area must still be considered to be research oriented.

I.8.3 Conformance testing

The topic of *conformance testing* as defined in [29] involves testing both the capabilities and the behaviour of an implementation. The *static conformance requirements* define the minimum capabilities that have to be supported by a conforming implementation. It is the external behaviour of an implementation that is of interest , i.e. the behaviour of an implementation is defined relative to how the events it may perform is observed in the environment of the implementation. This means that even if both internal and external behaviour are described by a standard, it is only the requirements on the external behaviour that has to be met by the implementation. These requirements are referred to as *dynamic conformance requirements*, and are defined as a relation on the external behaviour between the implementation and the specification standard.

Currently, standards are usually given in natural language, in some cases accompanied by state tables, and the test suites are developed manually. This creates problems when developing test suites, since there is no possibility to formally verify that a test suite is specifying the same set of requirements on the implementation as the standard does.

It is believed that the introduction of formal languages such as SDL in specification standards is a way to overcome these problems. When formal specifications are available, possibilities will emerge for computer supported test suite derivation and/or test suite validation. Much research efforts have been spent in the area of computer supported test suite derivation during the past few years, and mainly two approaches can be identified:

– automatic derivation; and

– test derivation based on simulation.

I.8.3.1 Automatic test derivation

Automatic test derivation uses either techniques available in the area of hardware testing or techniques similar to the ones used in validation, i.e. state space exploration. The main characteristic of the approach is that a representation of the externally observable behaviour of the system is computed. From this representation, the test suite is then automatically generated, based on a description of which behaviour that should be tested. This information can be based on message sequence charts, see [30], if present, or on the informal requirements on the system. It should be noted that it is possible to specify requirements on implementations that are not testable. This should be taken into account when preparing specification standards. In order to get testable specifications, some restrictions are necessary on the use of SDL constructs.

Examples of techniques used to compute the observable behaviour are techniques based on state space exploration and techniques based on the concept of finite state machines. In the latter approach, the processes in the system are transformed into a set of finite state machines, which then are composed in order to compute the observable behaviour. Since the semantics of SDL are based on the concept of extended finite state machines, it is in general not possible to perform this transformation.

I.8.3.2 Simulator assisted test derivation

The use of a simulator can assist test derivation in different ways. First, the test designer can get familiar with the behaviour of the specification by simulating the specification. When knowledge is acquired of which patterns of the behaviour that are relevant to test, the simulator can assist the derivation of the test cases by allowing for the test designer to interactively walk-through the specification and define the test cases according to the behaviour of the specification. Several simulators for SDL are commercially available from different vendors. However, the interactive simulation of the specification may become a hard and time consuming task if the behaviour of the specification is complex.

The test derivation can also be partially automated by the use of a special purpose simulator. In this case, the test designer guides the test generation by interactively making decisions on which behaviour to generate tests for, but the actual test generation is performed automatically.

I.9 Auxiliary documents

Auxiliary documents contain information that can be extracted (manually or by means of a tool) from an SDL system specification. The purpose is to provide overview. However, these documents can also be created before the SDL system specification, and can then serve as a basis for this.

I.9.1 Communication and interface specification

I.9.1.1 Introduction

In many situations it is desired to specify an interface of an entity, as part of the requirements specification of that entity. This subsection contains a general discussion of the communication and interface specification, as a basis for subsequent subsections, where different approaches for this kind of specifications will be presented.

To start with, a distinction must be made between *communication* and (dynamic) *interface*. Communication is assumed to take place between two or more parties, and all parties involved have some influence on the resulting combined behaviour.

An interface, on the other hand, applies to one single entity[7)]. An entity can have more than one interface. An interface specification expresses the possible behaviour of the entity at the given interface, disregarding the behaviour at the other interfaces of the entity. The behaviour at a given interface is called possible, since it is influenced only by the entity. When this interface is connected to some other entity, then the communication between the two entities will be a subset of the possible behaviour at the interface.

When considering an interface, it is usual to assume that the entity is connected to the *environment* via the interface. The environment, however, does not impose any restriction on the behaviour of the entity, observed at the interface.

---

7) This terminology is in line with the modelling concepts used by CCITT/ISO for Open Distributed Processing (ODP). It is recognized that the term “interface” is also used in a different sense in other standard works.

In the context of SDL, such a situation is illustrated by a block *A* connected to the environment by channels *I1* and *I2*, or a block type *A* having gates *I1* and *I2* as interfaces (see Figure I.9-1). Note that a channel connecting two blocks is a communication medium and not an interface. The entity could also be a process type, but this possibility is not covered explicitly in the following.

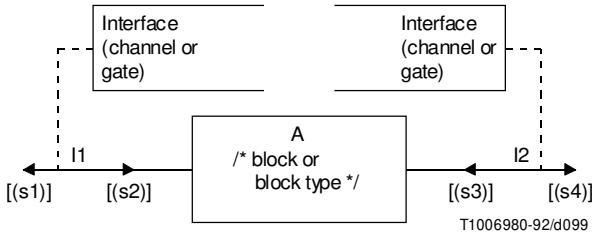


FIGURE I.9-1/Z.100

Interface of a block or block type

The specifications of the channels/gates (using the signal lists *s1*, *s2*, *s3* and *s4* together with the corresponding signal specifications which are not shown here) are also the specification of the *static* interface. For the specification of a *dynamic* interface or communication we can use :

- message sequence chart (MSC);
- process algebra; or
- channel substructure.

These three approaches will be described below. The *dynamic* interface or communication specification is applicable where a MSC can be used, e.g. in **step 2** and **step 5** in I.3.

Note that normally each interface is specified separately. This is one reason why the specification of the block or block type is not very useful in this context, since such a specification covers the *combined* behaviour satisfying simultaneously all interfaces of the block or block type.

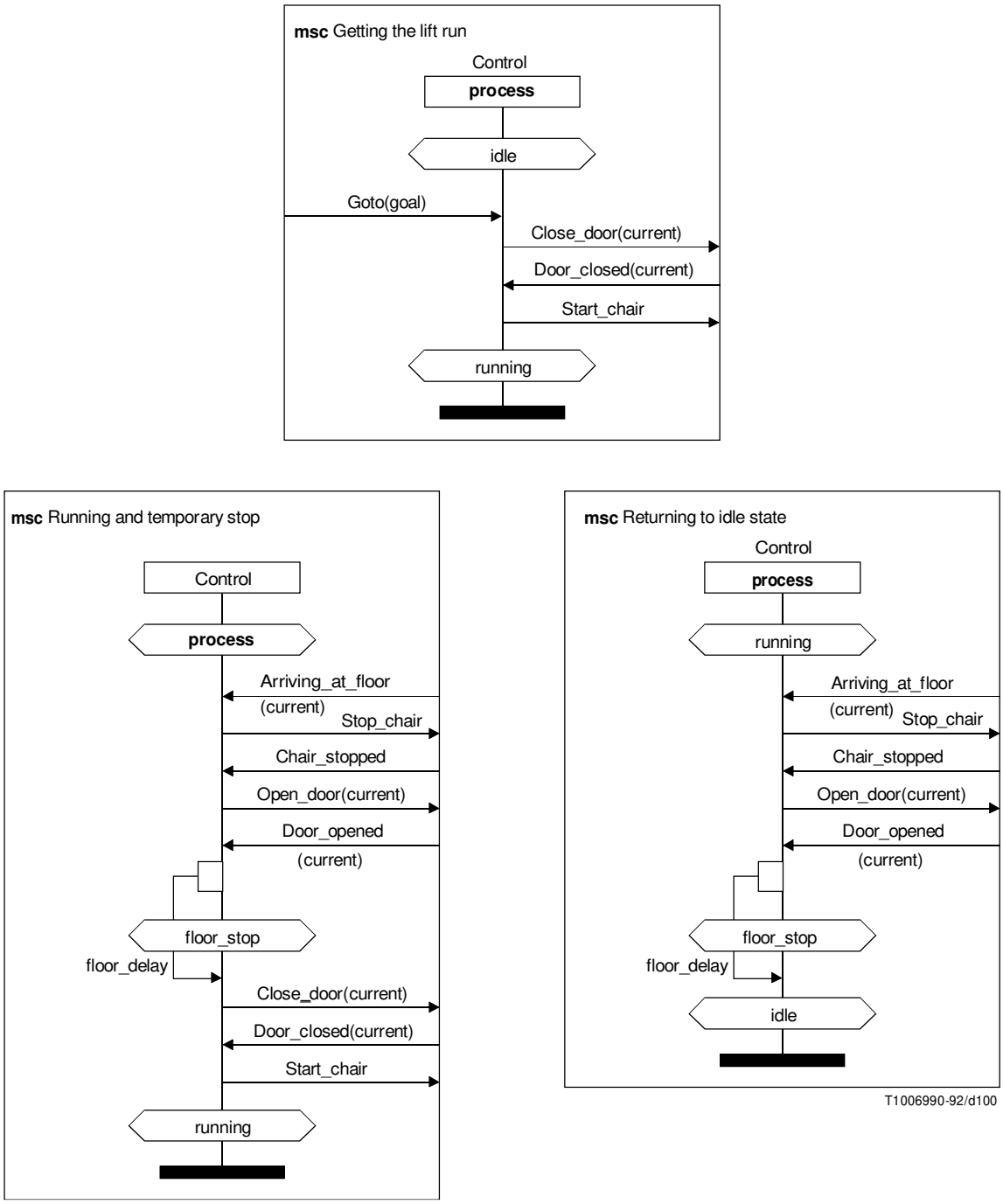
I.9.1.2 Using message sequence chart

The use of message sequence chart is covered in detail in I.6. As stated there, a MSC can be used for many purposes, including statement of requirements for SDL specifications, semi-formal specification of communication and interface specification.

The different uses of a MSC are correlated, e.g. an interface specification can be seen as part of a requirement specification. In fact, there is no difference between using a MSC for an interface specification and for the specification of a communication. This is because a MSC only covers a partial behaviour (one of many possible responses from an entity or the environment to a certain stimulus).

To ease the comparison of the above mentioned three approaches for the specification of a dynamic interface or communication, the use of MSC for the *Lift* example is shown here. Some MSCs have already been introduced in I.3, **step 5**, for the single-user case (*A lift ride* and *Lift request*).

The multi-user case can also be described by MSCs (see Example I.9-1). Since a MSC only describes one single-user case, it is advantageous to have a number of small MSCs that can be combined in different ways. The MSC *Running and temporary stop* can evidently be “added” to the MSC *Getting the lift run* a number of times before adding the MSC *Returning to idle state*.



EXAMPLE I.9-1

MSCs for multi-user case

I.9.1.3 Using process algebra

Introduction

In this subclause the use of a process algebra for communication and interface specification is outlined. The process algebra is based on LOTOS, and is denoted by LOTOS*. The main advantage of using LOTOS* for communication and interface specification compared with MSCs is a complete and more compact specification. It should be noted that the combined usage of LOTOS* and SDL, as outlined in this subsection, does not have a solid theoretical basis. Further, LOTOS* is not completely in conformance with the official language definition of LOTOS.

When using LOTOS* for communication and interface specification, no new documents are required. Instead, behaviour expressions in LOTOS* can be included into gate or channel specifications as comments. A LOTOS* behaviour expression defines through its operational semantics a set of traces, i.e. possible sequences of signals passing through the gate or channel.

The use of LOTOS* for communication and interface specification involves some special assumptions. The communication mechanism in SDL is asynchronous as opposed to the synchronous communication in LOTOS. Therefore, the synchronisation mechanisms of LOTOS are not to be employed. In addition, the communication can always be seen as taking place between two parties, the block and its environment. When only the observable behaviour is taken into account, these parties can be assumed to contain only one SDL process each. Therefore, no parallelism is present and it is enough to consider only the sequential part of LOTOS.

In interface specifications we have two kind of events; the input and output of an asynchronous signal. To express this in LOTOS*, we need value and variable declarations at some gate. Then, we have the following expressions corresponding to each other:

SDL	LOTOS*
input a;	? a
output a;	! a

Note that the gate name is omitted in a LOTOS* expression for practical reasons. Since a gate is not synchronised with anything else, nothing prevents us from taking the value and variable offerings at the gate as the consumption and sending of an asynchronous signal.

LOTOS* operators

LOTOS* offers several operators which can be used in communication and interface specifications. Some of them are straightforward to express in SDL, whereas some others offer a more compact representation than corresponding SDL constructs.

- **Action prefix**: Action prefix (;) can be used to prefix an existing behavioural expression with an action, e.g. *a;exp*. This means that the action *a* is executed followed immediately by the execution of the existing behavioural expression *exp*. In SDL, action prefix corresponds to the implicit ordering of actions in a transition.

- **Choice**: The choice operator ([ ]) means a nondeterministic choice between two behavioural expressions. It can be expressed in two different ways in SDL. If there is a choice in signals to be received, then this is taken care of by a state and the different inputs connected to it. In other cases, decision is employed.

- **Interleaving**: As was mentioned earlier, we need not consider synchronisation in interface specifications. This leaves us with only pure interleaving (| |), where the actions of the expressions it connects may interleave arbitrarily.

- **Enabling**: The enabling operator (>>) means a sequential ordering of two behavioural expressions in such a way that the right hand behavioural expression is executed only after the successful termination (with an **exi**t) of the left hand behavioural expression. It corresponds to an SDL nextstate, or to the normal ordering of actions in a transition.

- **Disabling**: The disabling operator ([>) means that the right hand behavioural expression, when executed, will interrupt immediately the left hand behavioural expression. Disabling can be expressed in SDL in a handy way by an asterisk state, followed by the transition which interrupts the normal behaviour.

– **Exit**: Means the successful termination of a behavioural expression, and corresponds to an SDL nextstate.

– **Stop**: has the same meaning as **stop** in SDL.

– **Recursion**: corresponds to entering an earlier state repeatedly.

Example

Consider the SDL system *Lift* in I.3.3. There are three interfaces to the block *Control*, one for each channel connected to it. These are represented by gates in LOTOS*. The interface specifications for *Chair* and *Floors* are shown in Example I.9-2.

```
Chair_interface :=
        ?Goto; Chair_interface
    [ ]
        ?Emergency_stop; ?Restart; Chair_interface
Floor_interface :=
        Open_door >> Floor_request >> Close_door >> Running
Running :=
        ?Floor_req; Running
    [ ]
        ?Arriving_at_floor; (Running [ ] Open_door >> Floor_stop)
Floor_stop :=
        ?Floor_req; Floor_stop
    [ ]
        i; (Close_door; Running [ ] Floor_request >> Close_door >> Running)
Floor_request :=
        ?Floor_req; (Floor_request [ ] exit)
Open_door :=
        !Stop_chair; ?Chair_stopped; !Open_door; ?Door_opened; Exit
Close_door :=
        !Close_door; ?Door_closed; !Start_chair; Exit
```

EXAMPLE I.9-2

LOTOS* interface specifications

Compare this with the process specification *Control* in I.3.3. As can be seen, repeated executions of transitions in a state are expressed in LOTOS* by recursive processes, such as *Running*.

As mentioned above, the interface specification can be used as a requirement specification for the SDL block. When specifying the block (and its processes), additional decisions are normally required concerning the detailed behaviour and the combination of behaviours at each interface. Examples of such decisions are: What happens when *Floor_req* or another *Goto* arrives while the lift is moving? What information should be conveyed by the signals and what information should be stored within the block? Nevertheless, a skeleton SDL process specification can be automatically extracted from an interface specification.

Semantics and conformance of specifications in LOTOS*

Here we will discuss briefly the relation of specifications expressed in LOTOS* to a full SDL specification.

To start with, a model for the dynamic semantics of a system consisting of SDL processes has to be constructed. For this purpose, asynchronous communication trees are used. This model is based on actor systems and their event diagrams [23]. The definition of the tree tells what are the possible sequences of observable signal inputs and outputs. The definition of an asynchronous communication tree of an SDL system and its environment provides the necessary semantical basis needed to formalise the conformance of an SDL-process specification to an interface specification.

The next step is to define a similar tree for interface specifications. This is more or less nothing but the operational semantics of LOTOS* and the labelled transition graphs based on it [31]. Intuitively, the paths of this transition graph represent the action sequences allowed by the corresponding behavioural expression in the same way as an asynchronous communication tree would.

The conformance of an SDL process specification to an interface specification can be then defined as a binary relation between the nodes of the asynchronous communication trees of the SDL process specification and the interface specification. Intuitively, there are three conditions to be fulfilled:

- Everything the SDL process can do must be allowed by the interface specification, which is the normal requirement for refinement.
- The SDL process must at any time be prepared to accept any signal which may be sent to the process according to the interface specification.
- The SDL process must carry out its task to the end, i.e. it may not stop any time it decides to.

One interesting point in the conformance is the distinction between the input and output of signals. It is intuitively clear that if, at some moment, several different signals may be sent to the process according to the interface specification, it must be able to deal with all of them. If some of them are discarded and do not lead to the response anticipated according to the interface specification, the detailed SDL specification corresponding to the interface specification can not be correct. However, if several response signals to one input signal are possible, it is the freedom of the specifier to choose which of them to use. In this respect, the requirements caused by input and output signals are dual.

I.9.1.4 Using channel substructure

Channel substructure is not an auxiliary document, but it can be used for communication and dynamic interface specification. This is shown here to make the picture more complete. For the channel *Floors* of the system *Lift* in I.3.3, the channel substructure *Floorsub* is introduced (see Example I.9-3).

The dynamic part of the interface specification is given by the process specification *Floor*. This should be equivalent to the LOTOS* process specification *Floor_interface* in Example I.9-2.

Timeout is expressed by a spontaneous transition, and unspecified decisions by nondeterministic decisions. The procedures *Open_door* and *Close_door* are specified in I.3.3 as part of the *Control* process specification.

I.9.2 Tree diagram

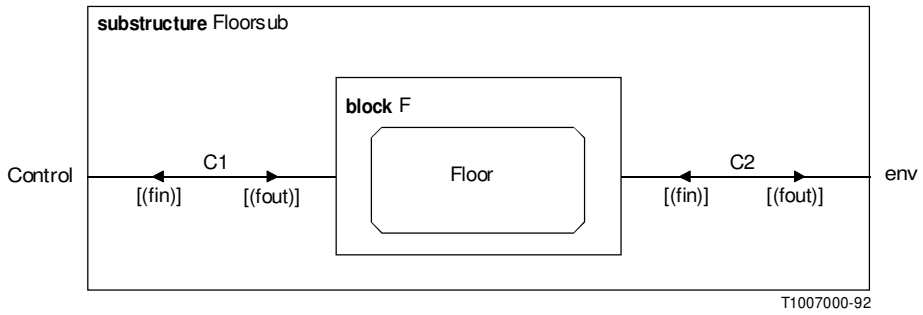
A tree diagram shows the components of an SDL system. The components (also called nodes in the sequel) form a hierarchical structure, with the system as the root, according to the containment relationships expressed by the syntax rules. The nodes of a tree diagram can be (in addition to the system-node):

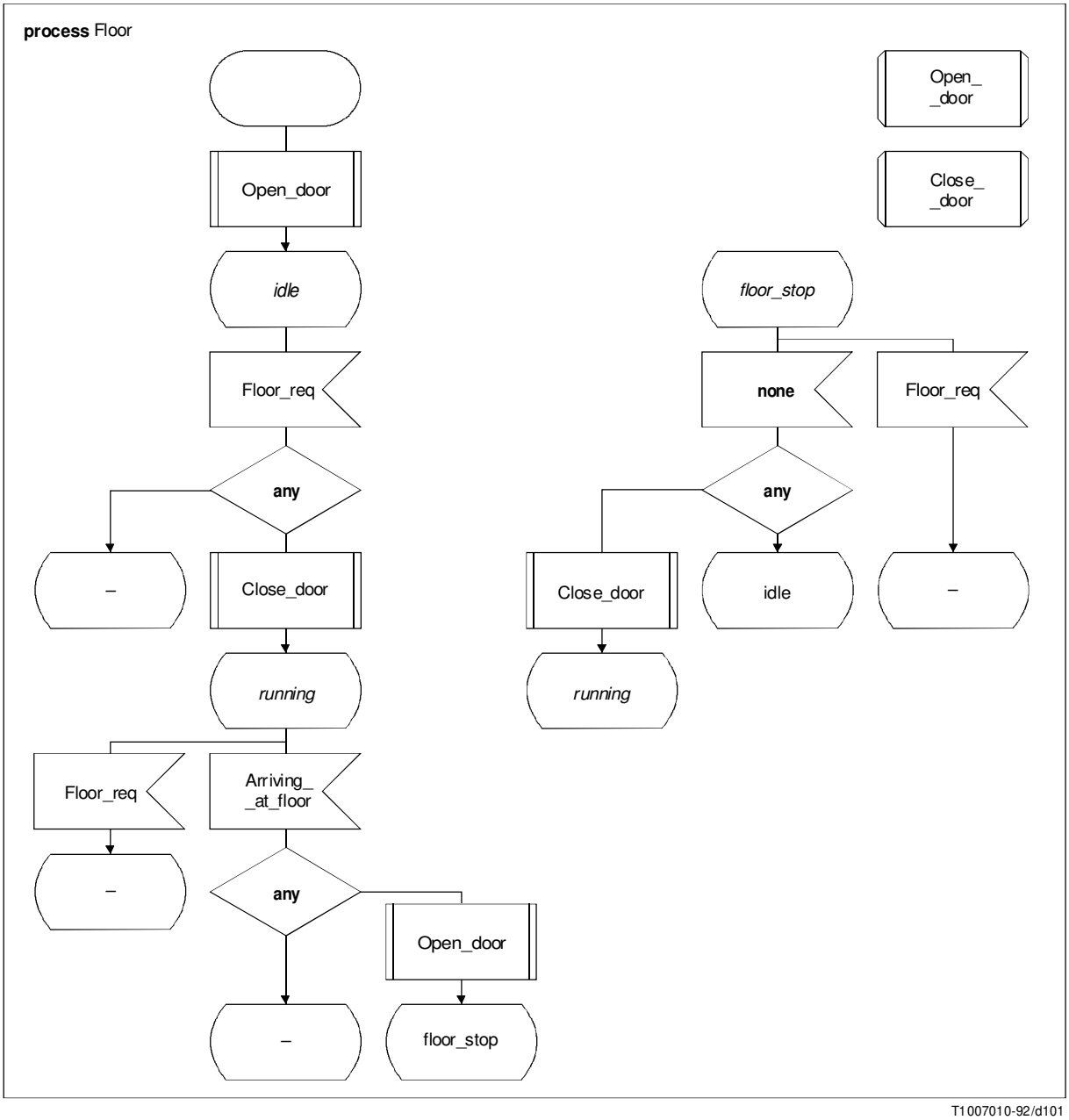
- blocks only, called *block-tree diagram* (see Figure I.9-2);
- blocks, processes and services, called *basic tree-diagram* (see Figure I.9-3);
- in addition to the nodes of *basic tree-diagram* also procedures and macros etc, called *general tree-diagram* (see Figure I.9-5);

Note that the different branches of a tree diagram need not have to contain the same number of nodes. All nodes represent definitions of entities. For the macro node a new symbol has been introduced

T1007020-92/d102

since no symbol for the macro definition is defined in this Recommendation. Note that this symbol is not normative.





EXAMPLE 1.9-3

Substructure for the channel *floors*

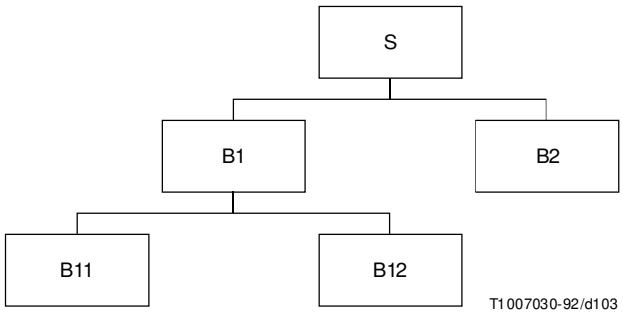


FIGURE I.9-2/Z.100

A block-tree diagram

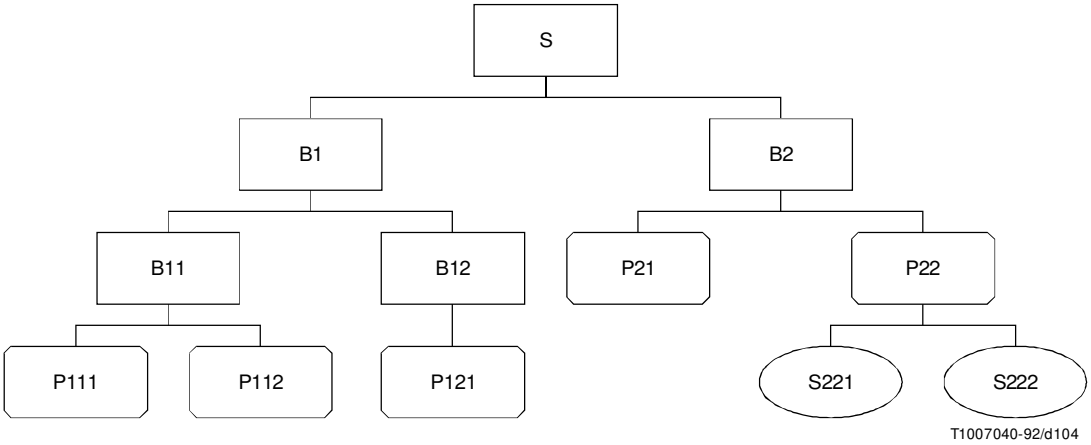


FIGURE I.9-3/Z.100

A basic tree-diagram

Macro nodes are always terminal nodes, and can be attached to any other node. A procedure node can be attached to any other node, except macro node.

The diagram should preferably be drawn with all the node symbols having a uniform size. This allows the nodes at the same level of partitioning to appear as a uniform level in the diagram.

A general tree-diagram can also contain channel substructure, as shown in Figure I.9-4. In this example, there is a bidirectional channel substructure *C1* between the blocks *B1* and *B2*. The channel substructure node is analogous to a block node, and can thus be the root of a similar tree of nodes.

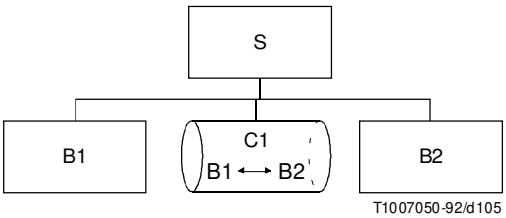


FIGURE I.9-4/Z.100

A general tree-diagram containing channel substructure

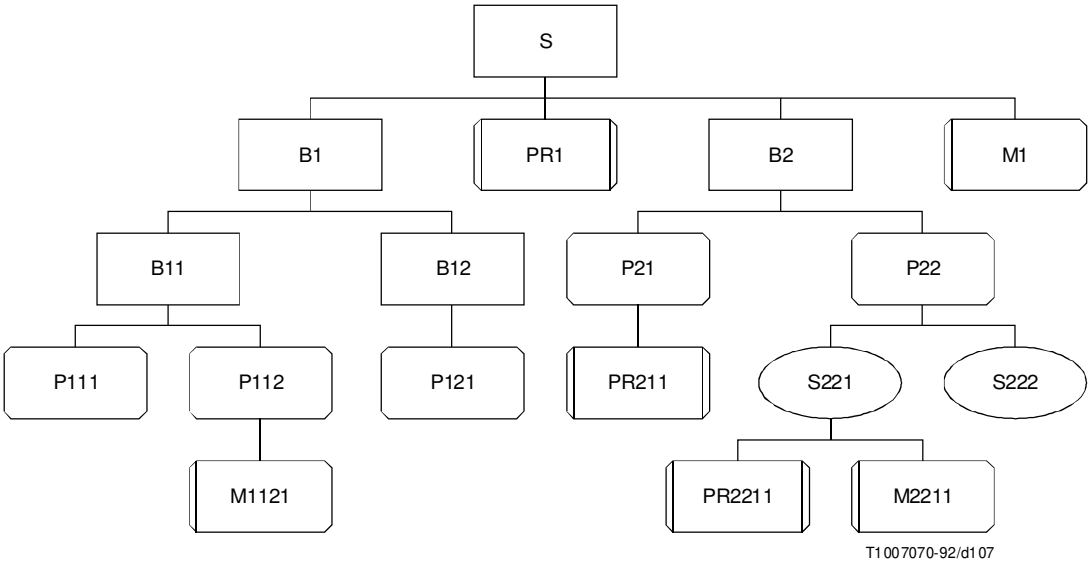
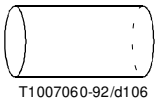


FIGURE I.9-5/Z.100

A general tree-diagram

For the channel substructure node a new symbol has been introduced



since no special symbol for the channel substructure definition is defined in this Recommendation. Note that this symbol is not normative.

It is often useful to partition a tree diagram into partial diagrams, for example if the diagram is so large that it requires more than one page. The splitting into several partial diagrams is done so that the roots of the additional diagrams appear as terminal nodes of the first diagram (see Figure I.9-6).

If it is not obvious that a terminal node of a diagram is further partitioned on other diagrams and/or where to find the continuation diagrams, references should be inserted using the comment symbol.

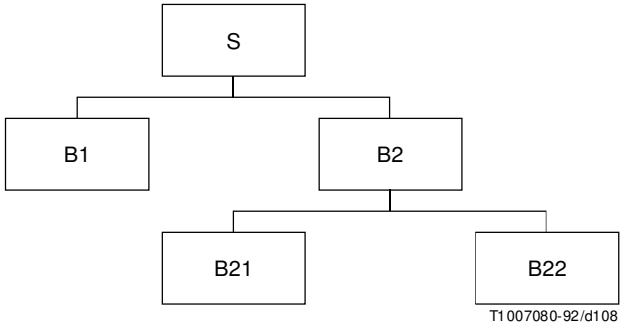


FIGURE I.9-6a/Z.100

Partitioning a tree diagram – Complete diagram

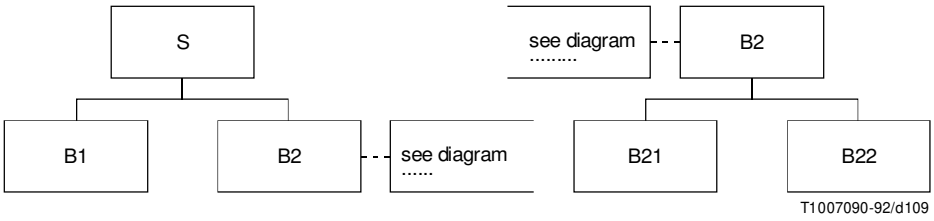


FIGURE I.9-6b/Z.100

Partitioning a tree diagram – Partial diagram

I.9.3 State-overview diagram

The intention with the state-overview diagram is to give an overview of the states of a process and show the possible transitions between them. The diagram gets very soon complicated when the number of states and transitions increases. Therefore, it is applicable only in simple cases, or when there are “unimportant” states or transitions that can 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 or an asterisk (*) notation.

To each of the directed arcs the name of the signal, or set of signals, causing the transition can be associated as well as signals sent during the transition. The list of sent signals is proceeded by the character /. Timer signals and setting of timers can be treaded as ordinary signals. An example of a state-overview diagram for the *Control* process of I.3.3 is given in Figure I.9-7.

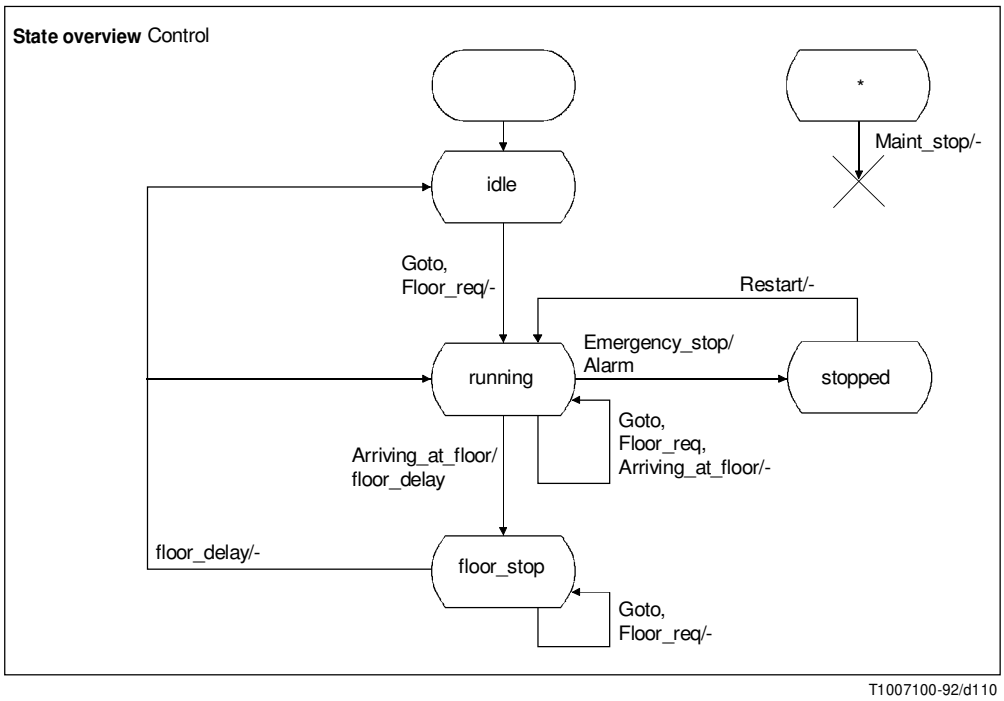


FIGURE I.9-7/Z.100

A state-overview diagram

I.9.4 Signal-state matrix

The signal-state matrix is an alternative to the state-overview diagram containing exactly the same information. It can be used, however, also when the number of states and transitions is big. In addition, it offers a possibility to check that all combinations of states and input signals are considered. See Figure I.9-8, which corresponds to the state-overview diagram in Figure I.9-7.

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 empty signal is used to show which is the initial state of the process.

The matrix may be partitioned into partial matrixes 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 partial matrix covers one aspect of the process behaviour.

I.10 Documentation

I.10.1 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.

signal state Control

state → / signal ↓	'start'	*idle*	*running*	*floor_stop*	*stopped*
	idle/-				
Goto		running/-	-/-	-/-	
Floor_req		running/-	-/-	-/-	
floor_delay				idle, running/-	
Arriving_ _at_floor			floor_stop/ floor delay; -/-		
Emergen_ cy_stop			stopped/ Alarm		
Restart					running/-
Maint_stop		'stop'/-	'stop'/-	'stop'/-	'stop'/-

FIGURE I.9-8/Z.100

A signal-state matrix

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 subclause is concerned with the logical organization of documents rather than their physical organization. This is left to the user's discretion. The similarity in requirements on 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 description can be made more readable and manageable. A documentation structure should provide both overview and details.

Properties of a document are:

- unique identification;
- revision designation;
- manageable size;
- is (generally) part of a documentation structure;
- is not part of another document (i.e. documents are not nested);
- is (generally) partitioned into pages.

SDL does not recommend certain documents or documentation structures. However, some language constructs are provided to aid the user in handling the documents. These are also covered in this subclause to make it more self-contained.

I.10.2 Language support for documentation

Nested specifications

An SDL specification contains, as do many other languages, a hierarchy of components that are arranged in a tree-like structure. This results in the system specification having a number of levels of hierarchy or abstraction. The traditional syntax structure of an SDL system specification is illustrated in Figure I.10-1. There nested specifications are shown, with the lower-order specifications contained within the next higher order of specification. This can be compared to a circuit diagram that is completely contained in a single sheet.

Although the nesting of specifications is certainly suitable from the tool-makers point of view and for very small specifications, it poses the following problems for the human user:

- it provides no overview;
- it provides no separation of abstraction levels;
- there is too much information in one place;
- it is difficult to map documents and specifications.

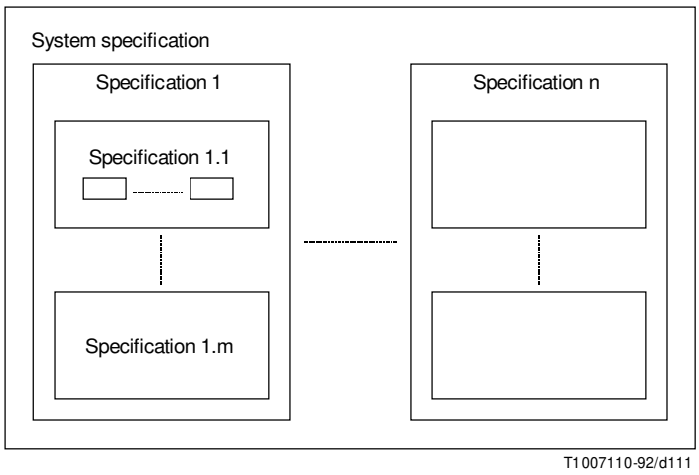


FIGURE I.10-1/Z.100

Traditional syntax structure of a system specification

Remote specifications

In SDL, these problems have been solved by introducing the **remote specification** construct. A remote specification is a specification that has been removed from its defining context to gain overview. It is similar to calling and defining a procedure, but is "called" from exactly one place (the defining context) using a **reference**. In other words, there is a one-to-one correspondence between the reference and the remote specification (see Figure I.10-2). This can be compared to a circuit diagram that is presented as a block diagram, where the blocks are described on separate pages as circuits with electronic components.

A system specification which makes use of remote specifications is a flat representation, as opposed to the hierarchical representation when nested specifications are used.

Mixing graphical and textual representation

By using remote specifications, the graphical and textual representation forms can be mixed at the discretion of the user of SDL (see Figure I.10-3).

It is a good practice to start with a system diagram to give overview. Specifications which require plenty of text should be given in the textual representation. This is illustrated e.g. in the lift example in I.3.

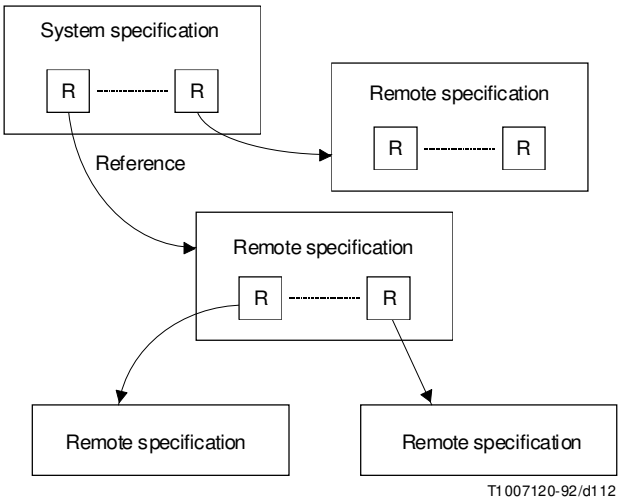


FIGURE I.10-2/Z.100

Remote specifications

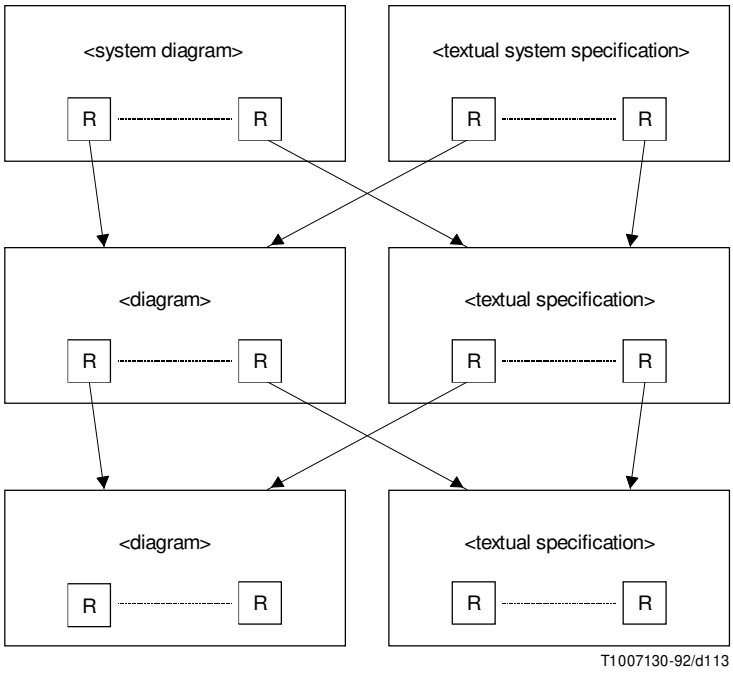


FIGURE I.10-3/Z.100

Mixing graphical and textual representation

I.10.3 Mapping of specifications to documents

Looking at a system specification as a set of remote specifications, a document can be seen as a container of one or more of these specifications:

<document> ::=

<system specification>

| {<remote specification> }+

Note that a remote specification can contain nested specifications.

If the system specification is small and nested, one single document is enough. If a flat representation is used, more documents can be used, e.g. one document per specification (diagram).

The normal case is probably a mixture of flat and nested representations. When deciding this mixture, the following rules are applicable:

- A specification should not be divided between several documents.
- If a specification is to be placed in a separate document, it must be a remote specification.
- When using the logical page concept to split a diagram into several diagram pages, the diagram pages should coincide with the physical pages of the document.
- If a diagram is more than one page, it must be a remote specification.
- A block, process and service interaction area must fit on one single page.

The graphical syntax provides the means to number each logical page of a diagram, although this is not absolutely necessary. The ordering of the pages normally does not matter, or can be derived anyway. The important thing is to have the heading repeated on each page in order to see what diagram the page belongs to.

In the textual representation, the language does not provide means to number pages, although this is crucial to avoid ambiguities. The reason is to follow the convention of programming languages, which rely on a page numbering scheme outside the language.

I.10.4 Documentation issues

I.10.4.1 A documentation structure

The set of documents covering a whole system must be structured so that the relationship of a document to other documents is clear. A document normally contains information pertaining to the whole system or to one of its components, and this must be also clear. This is generally achieved by attaching a document survey to each component, containing a reference to all other documents that belong to the component. The relation between the components is given by the system structure (see Figure I.10-4).

If the low-level SDL specification (see I.7) reflects the structure of the implemented system, then the description of the system structure can be generated automatically by a tool from the SDL system specification.

However, the SDL specification only covers a snapshot of the system, which may change in the course of time, possibly several times per year. For most real-life systems, some additional information must be provided for configuration and revision management, ordering, production etc. Also, the nature and the kinds of system components need further elaboration. These issues are discussed in subsequent subsections.

I.10.4.2 System components

A system component (including the system itself) is evolving through a number of well-defined revision stages. A new revision stage should be backward compatible with previous revision stages according to some criteria, which are part of the organization standard. The reason for a new revision stage is fault correction and/or introduction of new features. If backward compatibility cannot be achieved, then a new component (variant) must be created instead of a new revision stage. A system component should have a unique identity and designation for the revision stage.

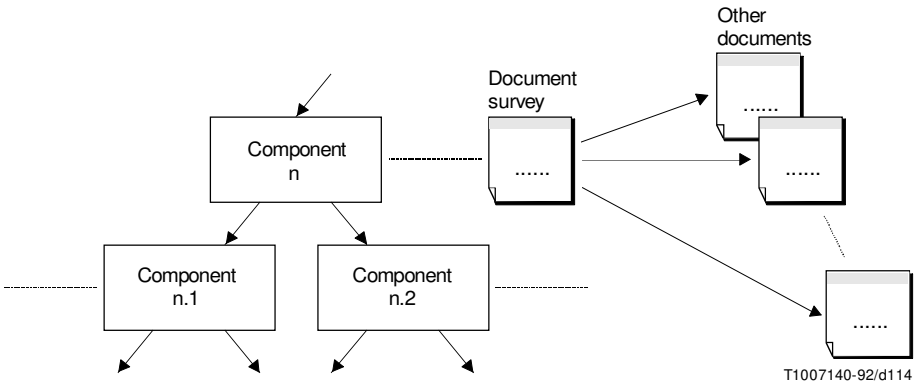


FIGURE I.10-4/Z.100

A documentation structure

The above general characteristics of a system component can be enforced more or less strictly dependant on how the component is used. From the usage point of view, we can distinguish between

- local components;
- standard components; and
- delivery components.

A local component could be a procedure used within one system by a development team. Clearly, the safe handling (identifying, documenting, changing, etc.) of a local component is not a major problem.

A standard component is generally available for all systems of an organization. Examples are hardware components and abstract data types (sorts). Clearly, standard components should not be changed frequently or at all.

A delivery component is used by client organizations, in different revision stages, possibly world-wide. Clearly, a delivery component must be handled following strict rules.

How should the SDL entities be treated in this respect? *Types* are natural candidates for standard components. *Blocks* should be used for delivery components, if these form part of a (low-level) SDL specification. There are, however, delivery components that cannot be covered in a sensible way in the SDL specification. All the other SDL entities should be considered local components.

SDL identifiers do not cover revision stage and are insufficient for delivery components. For standard and delivery components, some existing technique and tool for configuration and revision management should be used.

I.10.4.3 Kinds of document

The kinds of documents needed for a system are dependant on the purpose of the system and on the activities performed for it. In general, the following activities are relevant:

- ordering;
- production;
- operation;
- development;
- fault correction.

A general rule is that one piece of information should be contained in one document only, in order to avoid inconsistency due to changes. In cases this is impossible or not desirable, one of the documents should be considered the base document and the other documents derived documents.

The documents needed for *Development* form normally the basis for the other documents. An SDL specification covers structure and behaviour, and should form the kernel of this class of documents.

Documents for *Operation* contain mostly derived information. One special problem with these documents is how to customize them in order to cover those and only those features that have been delivered to the client. Some SDL constructs that can be useful here are *option*, *external synonym* and *subtyping* (specialization).

I.10.4.4 Document templates

For each document type a template should be provided, in order to enforce a uniform documentation style and also to ease the preparation of individual documents. A template specifies the type of information to be covered, table of contents, layout and possibly other documentation aspects that are considered necessary to standardize.

Different categories of components need different sets of document types. We consider here only the documentation of a delivery system component. The following document types are normally used, some are mandatory, others when appropriate:

- **Structure specification** – Lists contained components and possibly also their oldest revision stage that is required.
- **Revision information** – Is prepared for each revision stage and describes the differences compared with the previous revision stage.
- **Document survey** – Lists all the other documents that belong to the component, and indicates also their relevant revision stage.
- **Ordering information** – Describes the properties of the component from the ordering point of view, covering also optional features and the codes (values of external synonyms) for selecting these.
- **Description** – Provides a complete structure and possibly also behaviour description in SDL. Behaviour description may be covered only for components on the lowest system level (see I.3). The SDL specification can be given as an annex to a main document, which gives a short informal description as an introduction to the SDL specification.
- **Test specification** – Specifies the tests that the component must pass before it can be released.
- **Command description** – Describes the syntax and semantics of a command, given during *Operation*, that is performed mainly by the component.
- **Printout description** – Describes the syntax and semantics of a printout message that is generated by the component during *Operation*.
- **Installation manual** – Describes the installation procedure for the component.
- **Operating manual** – Describes how the component should be handled properly by a user/operator during *Operation*.

Bibliography

[1] *Basic Reference Model*, International Standard, ISO/IS 7498, 1984.

[2] *OSI Service conventions*, Technical Report ISO/TR 8509, 1987.

[3] CCITT Recommendation I.130 *Methods for the characterization of telecommunication services supported by an ISDN and network capabilities of an ISDN.*

[4] CCITT Recommendation Q.65 *Stage 2 of the method for the characterization of services supported by an ISDN.*

[5] CCITT Recommendation Z.120 *Message Sequence Chart*, Geneva, 1993.

[6] HARTMUT (B.): Fundamentals of algebraic specification, Volume 1, *EATCS Monographs on Theoretical Computer Science*, Springer-Verlag, 1985.

NACHUM: Termination of rewriting, *Journal on Symbolic Computation*, (3), pp 69-116, 1987.

HARALD: A completion procedure for conditional equations, *Proc. 1st Int. Workshop on Conditional Term Rewriting*, LNCS 308, pp. 62-83, Orsay, 1988, Springer-Verlag.

[7] CHONG-YI: Process periods and system reconstruction, *Lecture Notes in Computer Science 222, Advances in Petri Nets*, (G. Rozenberg ed.) Springer Verlag 1986.

[8] ENCONTRE, DELBOULBE, GAVAUD, LEBLANC, BOUSSALEM: Combining services, Message Sequence Chart and SDL: Formalism, method, tools, [10].

[9] FÆRGEMAND, MARQUES *Editors*: SDL '89 The language at work, *Proceedings of the 4th SDL Forum*, North-Holland, Amsterdam, 429 p. 1989.

[10] FÆRGEMAND, REED *Editors*: SDL '91 Evolving methods, *Proceedings of the 5th SDL Forum, North-Holland*, Amsterdam, 1991.

[11] GRABOWSKI (R.): Putting extended sequence charts to practice, [9].

[12] HOGREFE: OSI formal specification case study: the INRES protocol and service*, Technical Report*, University of Berne, April 1991.

[13] KRISTOFFERSEN: Message Sequence Charts and SDL specification consistency check, [10].

[14] NAHM: Consistency analysis of Message Sequence Charts and SDL systems, [10].

[15] REISIG: Petri Nets, *EATC Monographs on Theoretical Computer Sciences*, Vol. 4, Springer Publ. Comp., 1985.

[16] GRAUBMANN, GRABOWSKI (Rudolph): Towards an SDL design methodology using sequence chart segments, [10].

[17] HOARE: Communicating sequential processes, *Prentice-Hall International*, 1985.

[18] PARNAS: On the criteria to be used to decompose systems into modules, *Comm. ACM*, vol. 15 1972.

[19] PLOTKIN: A structural approach to denotational semantics, *Report DAIM-FN-19*, Computer Science Dept, Århus University, Denmark 1981.

[20] A common semantics representation for SDL and TTCN, *European Telecommunications Standards Institute (ETSI) Technical Report*, 1992.

[21] GODSKESSEN: A compositional operational semantics for Basic SDL, [10].

[22] WEST: General technique for communications protocol validation, *IBM J. Res. Develop*., 22(4), pp 393-404.

[23] AGHA: *ACTORS:* A model of concurrent computation in distributed systems. *The MIT Press, Cambridge*, Massachusetts, 1986.

[24] VALMARI: A stubborn attack on state explosion, *Proc. Computer-Aided Verification*, New Brunswick, New Jersey, 1990.

[25] GRAF (S.): Compositional minimization of finite state processes, *Proc. Computer-Aided Verification*, 1990.

[26] HOLZMANN: On limits and possibilities of automated protocol analysis, *Protocol Specification, Testing and Verification VII*. Elsevier Science Publishers B V (North Holland), 1987.

[27] YOUNG, TAYLOR, FORESTER, BRODBECK: Integrated cuncurrence analysis in a software development environment, *Proc. SCM SIGSOFT ´89 Third Symp on Software Testing, Analysis, and Verification*, Key West, Florida, 1989.

[28] EK, ELLSBERGER: A dynamic analysis tool for SDL, [10].

[29] Information Technology - Open Systems Interconnection - Conformance testing methodology and framework, International Standard IS 9646, 1991.

[30] HOGREFE: Conformance testing based on formal methods, *Proc. FORTE 90 3'rd Int Conf on Formal Description Techniques*, Madrid, 1990.

[31] BOLOGNESI, BRINKSMA: Introduction to the ISO Specification Language LOTOS, *Computer Networks and ISDN Systems*, 14, 1987, pp 25--59.

[32] BRAEK, HASNES, HAUGEN: Engineering real-time systems with an object-oriented methodology based on SDL, *SISU Project Report 1992*, Norwegian Computing Center, P O Box 114 Blindern, N-0314 Oslo, Norway.

[33] HAUGEN, MÖLLER-PEDERSEN: Tutorial on object-oriented SDL, *SISU Project Report 91002*, Norwegian Computing Center, P O Box 114 Blindern, N-0314 Oslo, Norway.

Appendix II

SDL Bibliography

(This appendix forms an integral part of this Recommendation)

(Helsinki, 1993)

[1] BELINA (editor): *SDL Newsletter* Telia Research AB, P O Box 85, S-201 20 Malmö, Sweden.
/* Published approximately once a year, free of charge, under the auspices of the CCITT */.

[2] BELINA, HOGREFE, SARMA: *SDL with Applications from Protocol Specification*, Prentice-Hall International (UK), 270 p., (1991). A German version will be published by Carl Hanser Verlag in 1992.
/* A textbook, on SDL, which can also be used as a reference. Can also be used as an introduction to the protocol specification area */.

[3] BELINA, HOGREFE: *The CCITT-Specification and Description Language SDL*, Computer Networks and ISDN System, 16, pp.311-341, (1988/89), North-Holland, Amsterdam.
/* A tutorial paper */.

[4] BRAEK, HASNES, HAUGEN: *Engineering real-time systems with an object-oriented methodology based on SDL*, SISU Project Report 1992, 320 p, Norwegian Computing Center, P O Box 114, N-0314 Oslo 3, Norway.
/* A textbook on methodology including techniques for implementation and desing. Updated for SDL92 */.

[5] CCITT Manual: *Guidelines for the application of Estelle, LOTOS and SDL*, ITU as above, 350 p. (1988); it is also available as an ISO document DTR 10167 (December 1989).
/* Contains a collection of examples, each specified in the three languages so as to aid subjective assessment and comparison of these. Contains also motivations for using formal specification languages */.

[6] FÆRGEMAND, MARQUES (editors): *SDL '89 The language at work*, Proceedings of the Fourth SDL Forum, North-Holland, Amsterdam, 429 p., (1989).

[7] FÆRGEMAND, REED (editors): S*DL '91 Evolving Methods*, Proceedings of the 5th SDL Forum, North-Holland, Amsterdam, 524 p., (1991).

[8] HAUGEN, MÖLLER-PEDERSEN: *Tutorial on object-oriented SDL*, SISU Project Report 91002, Norwegian Computing Center, P O Box 114, N-0314 Oslo 3, Norway.

[9] HOGREFE, SARMA: *The application of SDL to ISDN and OSI*, A paper in the Proceeding of the Seventh International Conference on Software, Engineering for Telecommunication Switching Systems (1989).

[10] HOGREFE: *OSI formal specification case study: the INRES protocol and service,* Technical Report, University of Berne, April 1991.

[11] HOGREFE: *Protocol and Service Specification with SDL: the X.25 case study*, Bericht Nr. FBI-HH-B-134/88, Universität Hamburg, (1988).

[12] SARACCO, SMITH, REED: *Telecommunications system engineering using SDL*, North-Holland, Amsterdam, 631 p., (1989).
/* A textbook on SDL. Contains also guidelines for the stated application area, with plenty of examples*/.

[13] SARACCO, TILANUS (editors): *SDL '87 State of the art and future trends*, Proceedings of the Third SDL Forum, North-Holland, Amsterdam, 463 p., (1987).

[14] TURNER (editor): *Using Formal Description Techniques An Introduction to Estelle, LOTOS and SDL*, will be published by John Wiley & Sons in 1992.
/* Is based on item 5*/.