International Telecommunication Union

ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

Series Z
Supplement 1
(04/2015)

SERIES Z: LANGUAGES AND GENERAL SOFTWARE ASPECTS FOR TELECOMMUNICATION SYSTEMS

ITU-T Z.100-series — SDL+ methodology: Use of ITU System Design Languages

ITU-T Z-series Recommendations – Supplement 1

ITU-T Z-SERIES RECOMMENDATIONS

LANGUAGES AND GENERAL SOFTWARE ASPECTS FOR TELECOMMUNICATION SYSTEMS

FORMAL DESCRIPTION TECHNIQUES (FDT)	
Specification and Description Language (SDL)	Z.100–Z.109
Application of formal description techniques	Z.110–Z.119
Message Sequence Chart (MSC)	Z.120–Z.129
User Requirements Notation (URN)	Z.150–Z.159
Testing and Test Control Notation (TTCN)	Z.160–Z.179
PROGRAMMING LANGUAGES	
CHILL: The ITU-T high level language	Z.200–Z.209
MAN-MACHINE LANGUAGE	
General principles	Z.300–Z.309
Basic syntax and dialogue procedures	Z.310–Z.319
Extended MML for visual display terminals	Z.320–Z.329
Specification of the man-machine interface	Z.330–Z.349
Data-oriented human-machine interfaces	Z.350–Z.359
Human-machine interfaces for the management of telecommunications networks	Z.360–Z.379
QUALITY	
Quality of telecommunication software	Z.400–Z.409
Quality aspects of protocol-related Recommendations	Z.450–Z.459
METHODS	
Methods for validation and testing	Z.500–Z.519
MIDDLEWARE	
Processing environment architectures	Z.600–Z.609

For further details, please refer to the list of ITU-T Recommendations.

Supplement 1 to ITU-T Z-series Recommendations

ITU-T Z.100-series — SDL+ methodology: Use of ITU System Design Languages

Summary

This Supplement is published as a Supplement to the ITU-T Z.100 Series for ITU System Design Languages (that is, Recommendations ITU-T Z.100 to Z.107, Z.109, Z.110, Z.120, Z.121, Z.150, Z.151 and Z.160 to Z.170). This Supplement 1 to the ITU-T Z.100 series of Recommendations outlines a methodology (called SDL+) for the use of these languages in combination, in particular where the ITU Specification and Description Language is used.

This Supplement covers the following topics:

1) terminology definitions and background material on the use and application of the ITU Specification and Description Language and related ITU System Design Languages;
2) an overview of the methodology (clause 4);
3) more detailed descriptions of:
   a) analysis of the requirements (clause 5)
   b) draft design and investigation of the application (clause 6)
   c) formalization of the application into a model of how an application behaves (clause 7)
   d) implementation issues (clause 8)
   e) validation (clause 9).
4) relationship with other languages and techniques;
5) an elaboration of the methodology for service specification.

This Supplement is not exhaustive. The intention is that users of SDL+ incorporate it in their overall methodologies, and tailor it for their application systems and specific needs. In particular, this Supplement does not cover the issues of derivation of an implementation from the specification, or the testing of systems in detail. In the case of testing, it is expected that this will be covered by a methodology dealing with the generation of tests for standards or products based on SDL+.

The methodology is a framework to be elaborated for the context of actual use. This Supplement has two parts. In Part I the framework is explained as an overview in clause 4, and with a more detailed description of the main parts in clauses 5, 6 and 7. The general framework is then specialized for the case of services specification in Part II, starting at clause 12. The parts from clauses 5, 6 and 7 are elaborated in clauses 13, 14 and 15. In this specialization some choices of approach have been made. Many of the details given in clauses 13, 14 and 15 can be used when the general framework is elaborated for other contexts and other choices.

History

Edition	Recommendation	Approval	Study Group	Unique ID[*]
1.0	ITU-T Z Suppl. 1	1997-05-06	10	11.1002/1000/4297
2.0	ITU-T Z Suppl. 1	2015-04-17	17	11.1002/1000/12447

[*] To access the Recommendation, type the URL http://handle.itu.int/ in the address field of your web browser, followed by the Recommendation's unique ID. For example, http://handle.itu.int/11.1002/1000/11830-en.

Table of Contents

Page

Supplement 1 to ITU-T Z-series Recommendations

ITU-T Z.100-series – SDL+ methodology: Use of ITU System Design Languages

1 Context

This Supplement is intended for those persons responsible for methodology in areas where the Recommendation ITU-T Z.100-series are used.

The objective of this Supplement is to provide a methodology for the effective use of the ITU-T Z.100-series Language Recommendations: the Specification and Description Language (SDL-2010) [b-ITU-T Z.100], [b-ITU-T Z.101], [b-ITU-T Z.102], [b-ITU-T Z.103], [b-ITU-T Z.104], [b-ITU-T Z.105], [b-ITU-T Z.106], [b-ITU-T Z.107]; and the Message Sequence Chart notation [b-ITU-T Z.120]. The application of this methodology should lead to a well-defined specification of the target application in SDL-2010 combined optionally with ASN.1 [b-ITU-T X.680], [b-ITU-T X.681], [b-ITU-T X.682] and [b-ITU-T X.683] and Message Sequence Chart notation [b-ITU-T Z.120].

When [b-ITU-T Z.105] is used, SDL-2010 is used to define behaviour and ASN.1 is used to define data. It is also possible to use SDL-2010 to define data as in [b-ITU-T Z.104]. This Supplement covers the use of SDL-2010 without ASN.1 as well as combined with ASN.1 as in [b-ITU-T Z.105].

Message Sequence Chart diagrams are used to describe sequences of events and normally do not cover every possibility. Message Sequence Chart diagrams are used to describe what happens for particular sequences of stimuli, whereas SDL-2010 defines how an application system behaves towards all stimuli for every possible status. Message Sequence Chart diagrams without SDL-2010 would not usually give a full description of system behaviour. In the methodology described in this Supplement, SDL-2010 would not normally be used without Message Sequence Chart diagrams. The Supplement does not describe a general methodology for the use of the Message Sequence Chart notation either alone or with languages other than SDL-2010 (with or without ASN.1).

Since the methodology assumes the use of SDL-2010 with ASN.1 and Message Sequence Chart notation, the term "SDL+" is used throughout the Supplement to mean "SDL-2010 with ASN.1 and Message Sequence Chart notation".

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 methodology in this Supplement is therefore incomplete and will need to be elaborated to produce the actual methodology for a particular context. This Supplement is to provide assistance to users of SDL+ and to promote unified usage and tool support of the language.

2 Definitions

This Supplement defines the following terms:

2.1 **activity**: A specific procedure in an engineering process, supported by one or several methods.

2.2 **agent**: An object that models how an entity behaves in the environment and application system.

2.3 **actor**: A person (or organization) in the environment of a system that has one or more roles of interaction with a system. In the context of this Supplement, an actor that interacts with the activities in the engineering system is called an *engineer*.

2.4 **analysis**: The stepwise activity for exploring collected requirements, organizing the information they contain, and recording this information in a format that reflects this organization (see clause 5).

2.5 **application concept**: A concept pertaining to a given application domain and the system that is being specified.

2.6 **application domain**: The field of activity in which the application system under specification will operate.

2.7 **application system**: (abbreviated to *system* where the qualifying term can be derived from context) the complex set (of parts) that is used to provide functionality and other characteristics. In the context of this Supplement there is usually no distinction between a system specification, a system description and a real system instance.

2.8 **ASN.1**: Abstract Syntax Notation One [b-ITU-T X.680], [b-ITU-T X.681], [b-ITU-T X.682] and [b-ITU-T X.683].

2.9 **association**: A dependency relationship between two or more classes (or two or more objects) in the Unified Modeling Language (see clause 13.2 and [b-OMG UML]), and shown as an annotated line between class (or object) symbols.

2.10 **behaviour**: Sequence of actions with stimulus and response aspects performed by a system that may change its state.

2.11 **class**: A description of the model of the system defining a set of objects that have similar properties (same structure and same behaviour). A class represents an application concept.

2.12 **classified information**: A model with the collected requirements structured and defined according to concepts that have been named and defined.

2.13 **collected requirements**: A set of requirements for the application collected as a result of the requirements collection activity.

2.14 **computational view**: A view of a system and its environment that focuses on decomposition of the system to allow for distribution (see "computational" viewpoint in [b-ITU-T X.903], clause 4.1).

2.15 **design view**: A view of a system and its environment that focuses on the functions required to support that system (see "engineering" viewpoint in in [b-ITU-T X.903], clause 4.1).

2.16 **draft design**: The stepwise activity for engineering partial or partly informal specifications from different points of view and at different levels of detail to investigate engineering design choices (see clause 6).

2.17 **draft designs**: Outline designs for the system under consideration using one or more models that describe the system partially.

2.18 **documentation**: The activity for selecting, cataloguing and storing the information about a product.

2.19 **enterprise view**: A view of a system and its environment that focuses on the purpose, scope and policies for that system (see "enterprise" viewpoint in [b-ITU-T X.903], clause 4.1).

2.20 **formalization**: The stepwise activity in which a formal SDL+ description of a system is produced (see clause 7).

2.21 **formal SDL+ description**: An SDL+ description that conforms to Recommendations for SDL-2010 [b-ITU-T Z.100], [b-ITU-T Z.101], [b-ITU-T Z.102], [b-ITU-T Z.103], [b-ITU-T Z.104], [b-ITU-T Z.105], [b-ITU-T Z.106] and [b-ITU-T Z.107], Message Sequence Chart notation [b-ITU-T Z.120] and ASN.1 [b-ITU-T X.680], is consistent and unambiguous without any SDL-2010 "informal text", and describes the system under consideration in a precise way.

2.22 formal validation: Systematic investigation of a specification (or implementation) to determine whether or not it possesses certain desirable properties and includes: checking the syntactic and semantic correctness of the specification (implementation), and checking that the known requirements of the specified system are expressed by the specification (implementation).

2.23 guideline: Advice to the user of this methodology on identifying decisions to be made, which can also provide reasons and direction for making decisions.

2.24 informal SDL+ description: An SDL+ description output (from Draft Design) that deviates from one or more criteria of formal descriptions, such as: for SDL-2010, deviations of ambiguity, use of informal text, and non-conformance with Recommendations for SDL-2010 [b-ITU-T Z.100], [b-ITU-T Z.101], [b-ITU-T Z.102], [b-ITU-T Z.103], [b-ITU-T Z.104], [b-ITU-T Z.105], [b-ITU-T Z.106], [b-ITU-T Z.107].

NOTE – Message Sequence Chart descriptions are always assumed to be formal within the constraints of their function, which is to provide a message trace.

2.25 information view: a view of a system and its environment that focuses on the semantics of information and information processing activities in the system (see "information" viewpoint in [b-ITU-T X.903], clause 4.1).

2.26 instruction: An imperative statement that specifies what is to be performed as part of a step.

2.27 method: The combination of a notation with instructions, rules and guidelines for its use (see [b-Bræk] and [b-Olsen]). A *method* is a systematic way to achieve a particular goal. A method is descriptive.

2.28 methodology: An organized and coherent set of methods and principles used in a particular discipline. According to a 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 Supplement, 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).

2.29 model: A representation of something that demonstrates some of the properties and behaviour of that thing.

2.30 object: An individual element which is a member of a class and has a well-defined role in the system, possesses properties and can be managed and characterized by:

– its state (all its properties)

– how it behaves (the way it acts and reacts)

– its identity (in which it is different from the others).

2.31 object-orientation: A technique for partitioning the system under study into separate objects, grouping objects into classes and allowing these objects to interact with each other and with the environment to perform the functions of the whole system.

2.32 product: The application (or specification) and associated documentation that is to be produced.

2.33 requirement: An expression of the ideas to be embodied in the application under development.

NOTE – ISO/IEC Guide 2:1996, *Standardization and related activities – General vocabulary*, contains the definition: **requirement**: Provision that conveys criteria to be fulfilled.

2.34 requirements collection: The activity of initially evaluating captured requirements and handling questions about them as they arise during the development of a formal SDL-2010 description.

2.35 reuse library: Storage of application concepts that are available for usage at any moment in the development process of a system.

NOTE – The provision and operation of a reuse library is not covered by this Supplement.

2.36 rule: A statement of principles to prevent invalid, non-testable, or undesirable results. Conformance is expected for rules expressed with "shall" and is highly recommended for rules expressed with "should" (but exceptions exist).

2.37 service: A set of functions and facilities offered to a user by a provider.

NOTE – In this definition, the "user" and "provider" may be a pair such as application/application, human/computer, subscriber/organization (operator). The different types of service included in this definition are data transmission service and telecommunications service offered by an operating agency to its customers, and service offered by one layer in a layered protocol to other layers.

2.38 specification: Details and instructions describing the behaviours, data structures, design, materials, etc. of an implementation that conforms to an application.

NOTE – SDL-2010 specification: A specification using the Specification and Description Language defines how a system behaves in a stimulus/response fashion, assuming that both stimuli and responses are discrete and carry information.

2.39 step: A self-contained task included in an activity.

2.40 system: (see *application system*).

2.41 technique: A way that a method is realized.

2.42 technology view: A view of a system and its environment that focuses on the choice of technology in that system. (See "technology" viewpoint in [b-ITU-T X.903], clause 4.1).

2.43 testing: The combination of the two activities *Test specification* and *Test execution*.

2.44 test specification: An activity to produce a set of tests for particular purposes such as testing the conformance of an implementation to a formal SDL+ description or the interoperability of different products.

2.45 test execution: An activity to run the tests produced by Test Specification to produce a set of test results.

2.46 tool: A means to assist the accomplishment of a sequence of steps performed for a given purpose, usually (in this methodology) implemented as a software program.

2.47 type: An SDL-2010 **system type, block type, process type, procedure** or **signal** definition.

2.48 use sequence: A sequence of related transactions performed by a user of a system for a particular purpose.

2.49 validation: The process, with associated methods, procedures and tools, by which an evaluation is made that an application is (or a standard can be) fully implemented, conforms to the applicable standards and criteria for the application (standard), and satisfies the purpose expressed in the record of requirements on which the application (standard) is based; and in the case of a standard that an implementation that conforms to the standard has the functionality expressed in the record of requirements on which the standard is based.

2.50 validation model: A detailed version of a specification or implementation, possibly including parts of its environment that is used to perform formal validation.

3 The advantage of using SDL+

It is probably widely accepted that the key to 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 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* or as a model (for example in a standard) against which an *implementation* can be compared. A specification 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 formal specification language) is not intended to be run on a computer, though that may be possible and useful to determine errors in the specification. 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 SDL+ makes it possible to analyse, reason about and simulate alternative system solutions at different levels of abstraction, which in practice is impossible when using a programming language due to the cost and the time delay. SDL+ 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.

3.1 Understanding an SDL+ specification

The concepts in SDL+ are based on mathematical models and the theory of meaning. How to apply the models of SDL+ to an application is given below.

The application domain of a system is understood ultimately in terms of the concepts of a natural language. Individuals acquire understanding of these concepts through a long process of learning and real life experience. In the description of an application in a natural language, descriptions of phenomena are as 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 SDL+, the specification makes direct use of neither the application nor implementation concepts. Instead the SDL+ 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 3-1. This mapping can be done in different ways, one way is to choose names for the concepts introduced in the SDL+ that give good association to the application concepts; another way is to comment the SDL+. 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* 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, 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 consequently this always has limitations.

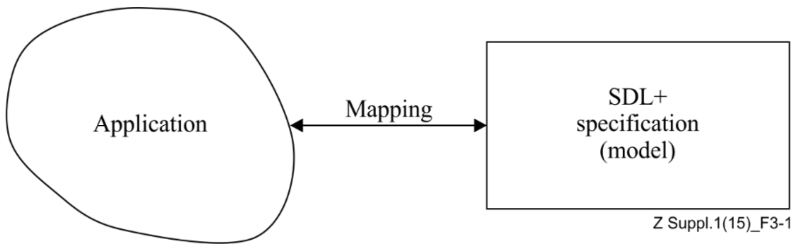


Figure 3-1 – How to understand an SDL+ specification

3.2 The application area of SDL+

Although the Specification and Description Language (SDL-2010, the basis of SDL+) is widely known within the telecommunication field, it has a broader application area and is also being used in other industries. 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-2010 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, that is, the co-operation 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-2010 covers different levels of abstraction, from a broad overview down to detailed design. It was not originally intended to be an implementation language, but automatic translation of SDL-2010 specification to a programming language is possible in many cases. The amount of change required to a specification in SDL-2010 to produce an implementation described in SDL-2010 can be quite small in some cases.

3.3 Relation to implementation

Normally, a distinction is made between *declarative* and *constructive* languages. SDL-2010 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*. SDL-2010 does not make any distinction between specification and description.

The internal structure of the system is normally represented in SDL-2010 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. Standards bodies normally treat conformance of implementations to specifications by *explicit conformance statement*. This is necessary also when a constructive language is used. Typically conformance is required on normative interfaces, each of which in SDL+ is usually a *channel* and often conveys data specified in ASN.1.

Because SDL+ describes structure and behaviour, it is possible for an SDL+ *description* to represent precisely an implementation, and conversely such a precise *description* can be used as the language description from which the operational software is derived automatically. Thus, SDL+ can be used as both an abstract specification language (for example in standards for protocols) and also as an implementation language. The use of one language avoids a possible source of error: the translation of one language into another. A specification and a corresponding implementation in SDL+ may have some significant differences that result from taking implementation issues into account. Although the same language is used at different levels, some parts of an SDL+ specification are likely to require changes to be directly usable in an implementation based on SDL+.

It is an assumption in the methodology in this Supplement that the amount of engineering to be done to change a detailed formal (executable) SDL+ specification into an implementation description is quite small. This assumption is not always valid. If the change needed is significant, the methodology will need to be extended to cover the additional work for implementation description. In any case, the methodology does not cover the generation of real system instances from the description. The methods that need to be applied for realization vary according to the equipment concerned, the organizations involved, the physical locations and many other factors. The methodology produces SDL+ that is used either as a specification or an implementation description.

PART I – THE FRAMEWORK METHODOLOGY

4 Overview of activities and an outline of the methodology

The methodology described by this Supplement is a coherent set of activities used in the engineering of systems to produce a product definition (either a specification or an implementation description). The product definition can be used as part of a standard, part of a procurement specification, as a specification prior to implementation, the source code of an implementation, or as the basis for developing tests for a product.

The methodology is divided into activities. Analysis of requirements and the formalization into SDL+ 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 elaborating the methodology. The selection must be based (among other things) on the characteristics of the application.

An activity is a method or process that is governed by some principles, accepts inputs and produces outputs (see Figure 4-1). The person (or organization) that carries out an activity can be called an "actor" and can be said to carry out different "roles" (or functions). In the context of this Supplement, the term "engineer" is used instead of "actor", because the activities are engineering activities. For engineering using SDL+, the roles that are undertaken by various engineers may be systems analyst, programmer, test engineer, hardware designer, project manager, equipment operator and so on. These roles are usually well defined. For engineering of standards, the roles might be standard scoping authority, standard rapporteur, standard designer, abstract test suite designer, standard approval authority and so on. An activity usually corresponds to one principle role (for example system specification and system specifier), but often interacts with other roles (equipment customer, product manager, test engineer). The decision on roles undertaken by engineers is outside the scope of this Supplement. An engineer often has more than one role.

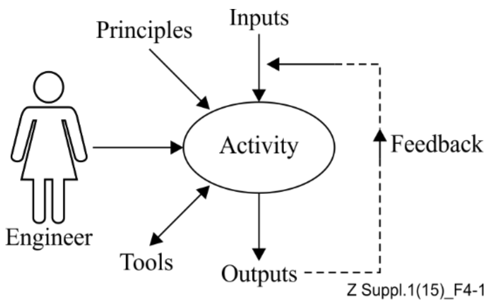


Figure 4-1 – An activity

In equipment engineering it is obvious that the activities for marketing, design and management take place in parallel and also have dependencies. This is usually true even for the creation of standards. Within the activities defined in this Supplement, there are dependencies between the activities but (within the constraints imposed by these dependencies) activities can take place in parallel. The Documentation activity, for example, can start as soon as there is a clear understanding of the scope and objectives of the application, but cannot be completed until a suitable SDL+ description has been produced. Other than the logical constraints imposed by the need for inputs from other activities, this Supplement does not prescribe the way that activities are scheduled, organized and managed. There is no implicit "life cycle" model.

For systems engineering in general it is useful to be able to view the system in different ways. Different views of the system allow different facts about the system to be considered. The open distributed processing (ODP) approach [b-ITU-T X.903] has five different views (viewpoints): enterprise, information, computation, design ("engineering") and technology (see Figure 4-2).

NOTE – "engineering" in the ODP sense is concerned with design issues such as control mechanisms, performance, distribution and so on. In this Supplement, the view is called "design" to avoid confusion with "engineering" as the term used for all activities.

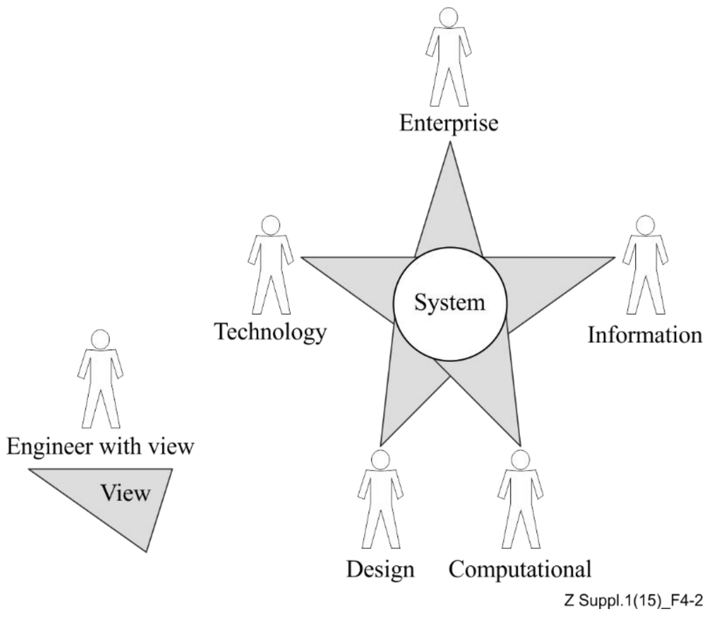


Figure 4-2 – The ODP views of a system

An enterprise view reveals the overall policies, actions and goals for the system. In the cases where this Supplement would be used, the enterprise view concerns the reasons for, the purpose of, and the use of the application. This information should be within the requirements stating the need for the application. The enterprise view is usually expressed in natural language initially. The enterprise view captures requirements. From the initial natural language requirements a description in the goal requirements language (GRL) of the User Requirements Notation (URN) [b-ITU-T Z.151]) can be derived. The enterprise view should be used within the scope description of the application and for validation of the product. Scenarios in the use case map (UCM) notation of URN or in Message Sequence Chart notation can capture use of the application.

An information view reveals the information structures, that is, how the data items in the system are related to one another. In the cases where this Supplement is applied, the information view concerns the places where data items are stored, the number and size of the data items, the links between data items, and the cardinality (that is, multiplicity) of these relationships (one to one, one to many, many to many). Class and object diagrams of the Unified Modelling Language (UML) [b-OMG UML] can be used for the information view. Outputs produced by the main methodology activities in this Supplement contain information views of the system.

A computational view reveals how the data items within the system are processed. The view concerns how the information is transferred, retrieved, transformed and managed. The stimulus for a computation can come from within the system or from outside the system. Together with the information view, the computational view defines the behaviour of the system. In the cases where this Supplement applies, the computational view is described in SDL-2010 processes using data defined in either ASN.1 or SDL-2010. The system is decomposed so that it can be distributed over physical units.

A design view reveals how the behaviour of the system is organized and actually distributed to take account of the supporting environment. Issues such as transmission characteristics, distribution and concurrency are taken into account, and the view may include performance and reliability. Often design is a compromise between different issues. In the cases where this Supplement applies, the SDL+ may include design requirements on distribution of functionality and concurrency.

These requirements are reflected in the structure (blocks and processes) in the SDL+ and annotation. If these issues are mandatory for the application, then the SDL-2010 structure shall match the requirements. In all other cases, the SDL-2010 should be structured to be as clear as possible, while also making it easy to both test and validate.

A technology view is concerned with technology issues within the system. This is only relevant if the implementation description is significantly different from the SDL+ specification. The methodology in this Supplement does not cover this case.

To summarize the above, the engineering of an application involves different views of the system. These views are contained in different models of the system. During the engineering process the system is gradually built up by producing these models. Since the models are all models of the same system, they are all related.

When engineering of a system begins, knowledge about the system generally falls between two extremes:

– The system is poorly understood, and the requirements are only vaguely defined. Knowledge and experience about the system are not available. There are no similar systems from which engineers can draw analogies.

– The system is well understood and well defined. It has already been implemented, and the engineering task is to modify an existing system. Expertise is available, the changes present no difficulty, and specifications for the existing system were created in SDL+.

In the first case, a large part of the work of engineering of a system is to understand what is required of the system and how the system is to perform. In the second case, the work consists of making changes to the SDL+ and documentation. This can be done by creating some new SDL+ types by adding to existing types, or perhaps just assembling existing SDL+ types in a new way to create a new system.

The methodology presented in this Supplement is applicable to the engineering of the system definition of an application, but ignoring implementation details. This is called the "specification" of the application. The engineering required is very similar to that required for any system and can fall anywhere between the two extremes outlined above. Implementation issues are not covered in detail by this methodology. To avoid cumbersome text in the rest of this Supplement, "system definition of the application" is denoted by the word "system".

The methodology concentrates on three main activities:

– Analysis, which is the organization of requirements and identification of concepts (with names and definitions) for the system;

– Draft Design, which is the transformation of classified information into draft designs that cover part of the system or are partially formal;

– Formalization, which is the expression of the specification in terms of formal SDL-2010, supplemented by Message Sequence Chart notation and ASN.1.

The methodology touches briefly on four other activities. One is requirements capture, which covers Requirements Collection at the beginning of a project. The second activity is Validation and is performed on formalized specifications as they are produced. The third activity is Documentation, which deals with the selection of system specifications for archiving and potential reuse. The fourth is Implementation, which deals with the generation of executable code for the target system. Figure 4-3 illustrates the methodology. The activity of requirements capture is only partially covered by this Supplement: the description in this Supplement only covers Requirements Collection. Not all the information flows are shown in Figure 4-3.

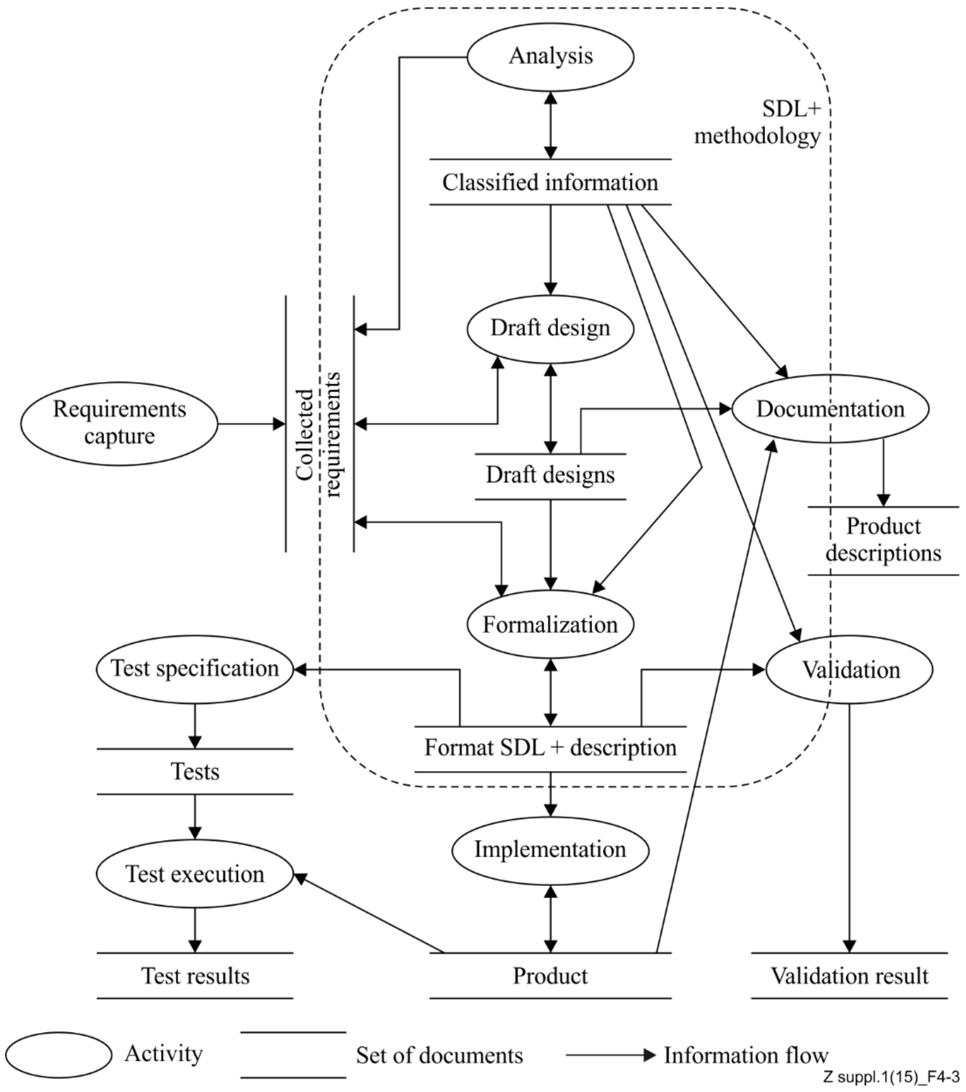


Figure 4-3 – The SDL+ methodology

The entire set of the seven activities (Requirements Collection, Analysis, Draft Design, Formalization, Validation, Documentation and Implementation) is applicable when a system engineering project starts with a poorly understood system. To the extent that the system is well defined however, Analysis and Draft Design may not be necessary. Engineers who use this methodology first assess how well the system is understood and defined, then decide which activity is the appropriate entry point in the methodology. To facilitate this choice, lists of criteria for beginning and completing the three main activities are given in this Supplement.

The lists of criteria also enable engineers to follow the methodology when the work on a systems engineering project is overlapping and iterative. That is, requirements capturing may still be taking place while existing requirements are being classified. This activity may in turn not be complete when Draft Design starts. Formalization may begin as soon as engineers have a clear idea of the system's interfaces, and the first outline of the documentation for a specification may be produced as soon as the overall architecture is clear. Likewise, through formalizing a specification in SDL+, engineers may raise questions that oblige them to reclassify requirements or restructure the description produced during Draft Design.

The following subclauses briefly describe the nine activities into which this methodology is divided. These subclauses also define the flexibility engineers have in switching from one activity to another. Analysis, Draft Design and Formalization are grouped in one clause because they are closely related.

Validation, Test Specification and Test Execution are also grouped because many of the techniques for testing are also useful for Validation.

The activities Analysis, Draft Design, Formalization, Implementation and Validation are also further elaborated in clauses 5, 6, 7, 8 and 9 respectively.

4.1 The Requirements Collection part of requirements capture

The purpose of Requirements Collection is to check that the requirements for the specification are reasonably described. No prescriptions are given in the methodology for creating or collecting requirements; the starting point is that a body of requirements, in some form but preferably URN, has been obtained. The methodology assumes that the collected requirements contain at least an enterprise view of the system.

In this activity, minimum criteria are given for the acceptance of initial requirements. Some guidelines are also given for the use of quality criteria to determine that the Requirements Collection has been completed. The collection of requirements does not necessarily stop as soon as another activity is started. Some requirements may need refinement as a result of better insight into the problem. Changing circumstances can cause changes in requirements, and other activities can generate questions that lead to new requirements or changes in existing requirements. Figure 4-3 shows the feedback to collected requirements from other activities. The information flow is not shown for the resolution of a question through the activity "requirements capture".

The actual generation of requirements through research, knowledge engineering, or other methods, although not defined here, nevertheless occurs before development starts. In addition, whenever engineers have questions about requirements, the assumption of this methodology is that answers or decisions come from this activity. Questions about the completeness, consistency and other quality attributes of requirements may result in changes to existing requirements and in the addition of new ones to the set. When the questions cannot be resolved within the methodology, they need to be resolved by the parts of the requirements capture activity external to this methodology.

Below is a checklist of minimum criteria that initial requirements should meet. If the initial requirements are inadequate, solutions must be found in requirements capture activities outside the domain of this methodology.

1) If there are many initial requirements, or they are complex, is a summary of requirements provided?
2) Do the initial requirements contain a statement of purpose for the system?
3) Do the initial requirements describe the functionality of the system?
4) Is implementation considered feasible given the state of existing technology or the expected results of developing technology?

To determine whether a satisfactory set of collected requirements exists is a subjective judgement as the collected requirements are (usually) not in a form that allows an objective judgement to be made. The checklist for judging completion of Requirements Collection is:

1) There should be a belief that the collected requirements are unambiguous, complete and achievable (these attributes may subsequently be proved wrong).
2) The requirements should not be known to be incorrect, unverifiable, internally inconsistent or externally inconsistent (this does not ensure that they are correct, verifiable or consistent but avoids proceeding if one of these is known to be false).
3) It shall be judged that from the collected requirements it is possible to engineer a formal SDL+ description.

4.2 Analysis, Draft Design and Formalization

In this methodology, the modelling of requirements occurs in three activities. The classified information embodies the enterprise view and other views in the collected requirements. The classified information is a model with the collected requirements structured and defined according to concepts (with names and definitions). This is refined during Draft Design to describe parts of the system from an information view and parts of the system informally from a computational view. These descriptions are transformed during Formalization into a formal specification that gives a precise and complete description of the system covering the information, computational and (if necessary) design views.

4.2.1 Analysis

The collected requirements are structured either based on the Analysis structure of concepts used for previous products or according to a new structure devised expressly for the system being specified. It becomes the classified information. The characteristic of classified information is that the collected requirements (usually natural language and informal diagrams) have been structured as a model of concepts (with names and definitions).

There is no fixed set of concepts, nor is there a fixed structure. When a new application is tackled, the reuse library is searched for records of other systems with similar concepts and structures.

The result is more structured information and a defined terminology. Ideas are clarified and expanded, because the engineer has to put thought into the process and back references to the collected requirements.

The notations used to structure the information can be chosen to provide appropriate ways of recording the inherent concepts and structure in the requirements. Because requirements often contain descriptions of use sequences or such sequences are derived as a result of queries raised, Message Sequence Charts can be an appropriate tool to record some of the behaviour during Analysis.

4.2.2 Draft Design

The informal classified information (and/or collected requirements) is converted into draft designs in notations that have a formal structure and usually with a formal syntax. The main characteristic of the draft designs is that they are incomplete drafts of the system. Thorough examination and determination of the system behaviour are not essential because the purpose is to investigate possible designs; therefore the semantics of the notations used can be informal and the draft designs may allow a number of different behaviours. The result is that the meaning depends on interpretation of the words used and a shared understanding between engineers who use the documents. While this is often adequate for the enterprise view and perhaps the information view, these interpretations are not sufficient for a complete and precise computational view.

The notations used can be chosen to suit the particular system, the formal specification language to be used (SDL+), and the available resources (for example, expertise, effort and tools). SDL+ used informally is a suitable Draft Design notation. Message Sequence Chart diagrams are nearly always used. ASN.1 is useful either because the collected requirements already include ASN.1 or because it is an effective technique for describing the information view of a system. An object class model notation (preferably the same one used in Analysis) can be used to model the entities in the system and the relationships between them.

Improvements and corrections to the system concepts will often result from Draft Design. These are additional requirements and therefore are shown as feedback via the collected requirements in Figure 4-3.

4.2.3 Formalization

The formal SDL+ description is derived from the collected requirements, the classified information and the draft designs. During the creation of the formal SDL+ description, the SDL-2010 diagrams can be considered as a Draft Design as they often contain informal parts or are incomplete according to SDL-2010 language rules. These intermediate descriptions nevertheless provide useful views of the system and aid understanding.

There is sometimes more than one formal SDL+ description. An abstract description as a top-level formal specification is produced first, and then successively refined to the level required for the application. Further refinement may be done to produce a description that can be interpreted for Validation purposes. Such refinement is similar to producing an implementation from a specification in SDL+.

The formal SDL+ description provides a precise computational view of the system and (if needed) a design view. To be a formal description it should not contain any "informal text" (in the SDL-2010 sense), so that the behaviour is determined only by the formal semantics of SDL+ and does not depend on human interpretation.

4.2.4 Result of Analysis, Draft Design and Formalization

The classified information may not need to be produced if there is a clear understanding of the concepts involved and the concepts are well documented. However, when work starts on the concepts, misunderstandings often come to the surface and classified information may then need to be produced. The draft designs may not need to be produced either, if the system behaviour is well understood. In this case the formal SDL+ description can be written directly using the collected requirements, following the steps of Formalization; substituting the use of knowledge and experience for the use of the classified information and draft designs. This approach has the benefit of economy for the particular engineer and system, but does not produce records that might be useful later for a different engineer or system. In most practical situations it is better to start with a skeleton; bad approaches can be discarded before a lot of work is put into Formalization. In this case the option to have informal text in SDL-2010 during Draft Design is a valuable feature of the SDL-2010 language.

The complete result of the approach contains all three levels of description as compared with the formal SDL-2010 description that is only part of it. The complete result contains everything that is known about the system.

4.3 Validation and Testing

The two test activities (Test Specification and Test Execution) are together called *Testing*.

The difference between Validation and Testing is related to what is being compared with what. In the case of Validation the SDL+ description is compared mainly with the classified information model produced by the Analysis, whereas for Testing, the main comparison is between the SDL+ description and an implemented product. Both Testing and Validation concern determining that the application has the intended characteristics, and there is also input to Testing and Validation from the collected requirements (only major information flows are shown on Figure 4-3).

In this methodology, Validation includes all actions that are related to checking the "validity" or "value" of SDL+ definitions. Validation has two aspects: evaluating conformance to standards and other criteria for the application, and evaluating that the purpose (including functionality and performance) expressed in the requirements has been met. In both cases Validation can show that an SDL+ definition is invalid, but if no checks fail then a judgement will need to be taken on whether the SDL+ is valid. The term "verification" (reserved for the proof of the truth of a relationship between two models) is not used.

Within the Validation activity a formal validation model is used. This is either the SDL+ definition that is to be validated, or a model derived from it and including models of parts of the environment.

The validation model is used for formal validation: systematic investigation of the validity such as checking SDL+ language rules and applying test cases. Informal validation techniques, such as walkthroughs and inspection with a checklist can also be applied, and are usually needed to evaluate if the intended purpose has been met.

The formal SDL+ description is used as the basis of a validation model. Because some SDL-2010 constructs make it possible to define systems that may be difficult or impossible to validate, it may be desirable to place some constraints on the SDL+ used in Formalization so that the system can be validated. Depending on the requirements of Validation, it may be necessary to add to the formal SDL+ description. For example, so that the behaviour of the system can be validated for the case of maximum processes, the validation model may need to restrict the number of simultaneous instances of one SDL-2010 process to a lower number than the formal SDL+ description does.

The derivation of test cases for the validation model is similar to Test Specification, except that there are different test purposes. Applying test cases to a validation model is similar to the Test Execution. In fact, many of the same tests and same tools may be used for both testing the validation model and testing a product.

The most common form of testing is *conformance testing:* the assessment by means of testing whether a product conforms to its specification. Conformance testing is an important issue in product development, because it increases the confidence in correct interoperability of open systems, where conformance to a communication protocol or service specification is considered to be a prerequisite. The framework on formal methods in conformance testing [b-ITU-T Z.500] – defines the meaning of conformance if formal methods such as SDL+ are used for the specification of a communication protocol or service and provides a guide to computer-aided test generation.

[b-ITU-T Z.500] is intended for implementers, testers and specifiers involved in conformance testing to guide them in defining conformance and the testing process of an implementation with respect to a specification that is given as a formal description. It is applicable where a formal specification of a communication protocol or service exists, from which a conformance test suite shall be developed. It can guide the SDL+ methodology as well as the development of tools for computer-aided test case generation. [b-ITU-T Z.500] defines a framework and does not prescribe any particular test case generation method, nor does it prescribe any specific conformance relation between a formal specification and an implementation. It supplements the Recommendations on OSI conformance testing methodology and framework for protocol Recommendations for ITU-T applications [b-ITU-T X.290] and [b-ITU-T X.291], which are applicable to a wide range of products and specifications, including specifications in natural language. [b-ITU-T Z.500] interprets conformance-testing concepts in a formal context. The language usually used for tests is TTCN-3 [b-ITU-T Z.161], [b-ITU-T Z.161.1], [b-ITU-T Z.162], [b-ITU-T Z.163], [b-ITU-T Z.164], [b-ITU-T Z.165], [b-ITU-T Z.165.1], [b-ITU-T Z.166], [b-ITU-T Z.167], [b-ITU-T Z.168], [b-ITU-T Z.169] and [b-ITU-T Z.170].

Methods for other forms of tests, such as interoperability tests, can use the conformance testing methods as a basis.

Testing is not further described in this Supplement. For further information, see the references.

4.4 Documentation

The activities of Analysis, Draft Design, and Formalization generate texts and diagrams, some of which are needed for the product specification. The purpose of the Documentation activity is to select the essential descriptions from the documents and to organize these descriptions so that the specification of the application is easy to understand, concise and precise. If the specification is actually part of a standard or procurement document, then this activity is particularly important.

The formal SDL+ description is always included in the product description. If more than one formal SDL-2010 description has been produced, only one should be incorporated or other descriptions should be clearly marked as abstractions of the final model. Other descriptions often needed to understand the formal SDL-2010 description are:

– draft designs such as Message Sequence Chart diagrams
– structured information and concepts in the classified information
– informal text or diagrams in the collected requirements
– logical expressions about the system state
– additional informal text and diagrams not produced as part of this methodology.

In general, Validation is facilitated if behaviour specifications are supported by redundant static information, such as logical expressions bound to system states.

The product description is likely to change over time so version control of documents is essential.

4.5 Parallelism of activities

An activity cannot start until the criteria for starting the activity have been satisfied. An activity is completed when the criteria for completing the activity are satisfied. If there are no constraints (such as the availability of effort, tools or expertise) then the activities can take place in parallel. The implication of this is that logically all activities can be in progress in parallel. In practice this is often what happens, because the requirements or the context for a system often changes as the system is being developed; therefore, the new requirements have to be analysed, and draft designs have to be modified at the same time as Formalization is taking place. It is also quite usual to split the engineering of a system into separate parts and for the different parts to be at different stages of development. This may be due to resource constraints or because some critical parts of the system are engineered first to establish the feasibility, viability or cost of a particular design. Therefore, the methodology does not require one activity to be completed before another activity starts, but only requires the start criteria to be met.

5 Analysis activity

Analysis is a stepwise approach to exploring collected requirements, organizing the information they contain, and recording this information in a format that reflects this organization. Analysis should be used when an application domain or system to be specified is poorly understood, or when the collected requirements contain different descriptions of the same application concept, or to give the application concepts well-defined names.

During Analysis, information from collected requirements is organized in terms of application concepts. At least some of these application concepts, such as "service primitive" and "protocol data unit", are likely to have been previously defined and well-known. It is natural to relate some of the information from collected requirements to these basic application concepts. New application concepts, which can often be defined in terms of existing ones, are also identified and named to stand for system-specific information. The set of accumulated application concepts, and the relationships established between them, enable engineers to reason about the system, communicate ideas about it, and thereby to better understand it. An object-oriented method is well suited for this concept-modelling process. Figure 5-1 describes the input and output of the Analysis activity.

The classified information produced by this activity provides a base for the other methodological activities, in particular the creation of the formal SDL+ specification.

The Analysis activity is described here in a general way, but to carry out Analysis effectively on systems of any reasonable size, defined notations and the corresponding tools should be used. The techniques that are appropriate are usually called "object-oriented analysis" techniques such as (in alphabetical order) [b-Bræk], [b-OOSE] or [b-Rumbaugh]. In clause 13 in Part II of this Supplement, Analysis is elaborated using one of these approaches chosen on an arbitrary basis.

5.1 Starting Analysis

At the start of Analysis the objectives, information available and choices to be made shall be reviewed.

5.1.1 Objectives of Analysis

The objectives of Analysis are:

– to resolve any detected deficiencies in the collected requirements;

– to establish a foundation for Draft Design and Formalization;

– to identify, name and define the application concepts of the system;

– to produce the classified information (still an informal specification) that structures the information in the collected requirements and supports further development, operation and maintenance work on the system.

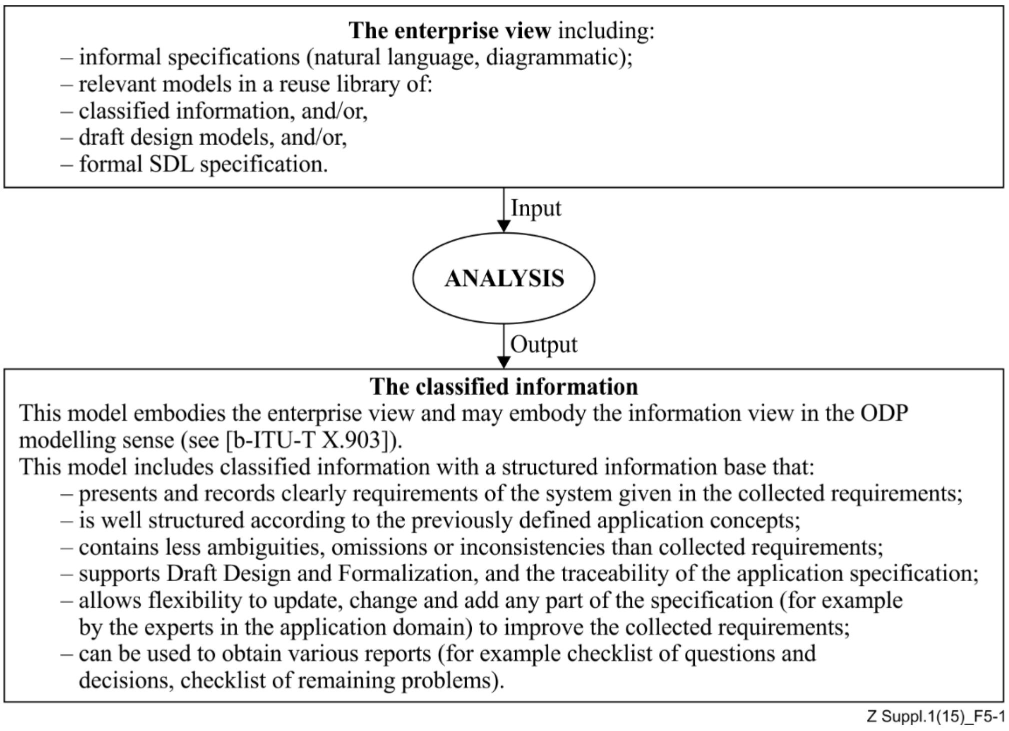

The enterprise view including:
– informal specifications (natural language, diagrammatic);
– relevant models in a reuse library of:
– classified information, and/or,
– draft design models, and/or,
– formal SDL specification.
Input
ANALYSIS
Output
The classified information
This model embodies the enterprise view and may embody the information view in the ODP modelling sense (see [b-ITU-T X.903]).
This model includes classified information with a structured information base that:
– presents and records clearly requirements of the system given in the collected requirements;
– is well structured according to the previously defined application concepts;
– contains less ambiguities, omissions or inconsistencies than collected requirements;
– supports Draft Design and Formalization, and the traceability of the application specification;
– allows flexibility to update, change and add any part of the specification (for example by the experts in the application domain) to improve the collected requirements;
– can be used to obtain various reports (for example checklist of questions and decisions, checklist of remaining problems).
Z Suppl.1(15)_F5-1

Figure 5-1 – Input and output of Analysis

5.1.2 Determining whether to skip Analysis

NOTE 1 – This subclause defines how to evaluate information available to the engineer and the engineer's understanding before Analysis can be started. Analysis may not be necessary for the development of a system. If not, the development of the specification can proceed directly to Draft Design or Formalization.

NOTE 2 – The methodology does not impose a strict sequence on when to perform the Analysis activity. The methodology only gives the engineer advice on how and when to use the Analysis activity.

Before starting Analysis, the engineer makes a judgement about the adequacy of the application concepts in the collected requirements and his understanding of them. He then decides whether the collected requirements are described at the level of detail, which is appropriate for Analysis, Draft Design or Formalization.

If the characteristics in the following checklist are satisfied:

1) There is no identical functionality or data at different locations.

2) It is easy to extract the essential characteristics of each application concept to distinguish it from the others and there is a well-defined and unique term for each application concept.

3) Details are hidden if they are not part of the essential characteristics of application concepts.

4) Application concepts are divided into a set of coherent parts that can be refined separately.

5) Application concepts are well ordered. It is important both to group similar functions and data and to logically decompose data into smaller parts.

The collected requirements are well-structured, and Analysis may be skipped.

5.2 Questions during Analysis

In Figure 4-3, the arrow from Analysis to collected requirements represents questions that arise during the development of an application. Questions about the completeness, consistency and other quality attributes of requirements may result in changes to existing requirements and the addition of new ones to the set. When the questions cannot be resolved within the methodology, they need to be resolved by the parts of the requirements capture activity external to this methodology.

5.3 Modelling approach for Analysis

Before Analysis, the ideas about the system may be:

– limited: a limited number of application concepts are known to the engineer.

– arbitrary: engineers create and classify application concepts on their differing knowledge.

– specific: each engineer has a model in his mind based on his specific background.

– complex: each engineer considers a number of views in his own model.

Analysis uses an object-oriented method to resolve these problems. This method helps the engineer:

– to provide a structured model based on a modular and therefore stable form that better accommodates possible modifications (which only affect some of the objects considered in the model);

– to divide the complexity of the problem into smaller parts where each of these is separately considered;

– to have a common view of the specification with all the engineers involved in the project;

– to control the possibilities of errors, ambiguities and inconsistencies;

– to keep track of the work.

In conclusion, for defining the system, engineers have a model that consists of:

– a logical view: to describe key elements of the system to specify;

– a dynamic view: to describe the individual behaviour of an object and also the behaviour of all the objects.

The logical view is expressed in a suitable object class notation. The dynamic view is usually expressed in Message Sequence Chart notation [b-ITU-T Z.120], or in the UCM notation of URN [b-ITU-T Z.151], or URN and Message Sequence Chart notation.

5.4 Analysis steps

The Analysis activity consists of two steps: inspection and classification.

5.4.1 Inspection

In the inspection step, a systematic survey is made of the content of collected requirements. Performing inspection enables the engineer to gauge whether existing knowledge or expertise about the application domain is sufficient and thus whether specification will be feasible. Since investigation of a little-known application domain can be costly, it is desirable to identify the need for such investigation as early in the specification process as possible.

The objectives for performing inspection are:

– to gain a general idea of the application domain;
– to assess the relevance of the collected requirements to the application domain;
– to determine whether coverage of the application domain by the collected requirements is comprehensive or scanty.

The inspection steps are the following:

1) Collect the source material:
   a) Compile a list of all sources of information. A bibliography represented as a table created in a word processing package may be sufficient.
   b) Categorize each document according to its apparent relevance to the application domain.
2) Inspect each document, beginning with the most relevant.
3) Reassess the relevance of each document to the application domain.
4) Read the most relevant document thoroughly, but without pausing to investigate difficult or poorly understood passages.

5.4.2 Classification

Classification consists of:

– identifying the application concepts, their structure and their various aspects in the collected requirements;
– identifying redundancy and missing information in the collected requirements;
– establishing connections between the input to Analysis and the classified information.

Classification leads to the elaboration of a logical object-oriented model of the system that describes:

– the structure of the identified classes;
– the structure of the identified objects in the classes;
– the aspects associated with each object in a class (information aspects, behaviour aspects, interface aspects and miscellaneous aspects).

The purpose is to identify classes and objects to a given level of detail and to make sure that they are appropriately named and defined.

Classification has 5 steps:

1) Identify and name the part of the system to classify.
2) Identify and name the classes.
3) Identify the object structures and their relationships.
4) Define classes and objects and review naming.
5) Identify the aspects of each class.

The behaviour aspects of the objects can be captured in UCM or Message Sequence Chart notation or both. If UCM is used, the Message Sequence Chart notation may be derived from the UCM [b-Miga et al.]. There should be one (composite) object that represents the system and one or more objects for the environment. There can be several (sequence) cases showing the use of the system.

5.5 Conclusion of Analysis

The results shall be reviewed to determine whether the Analysis can be considered complete; that is, the criteria for skipping Analysis shall apply. Draft Design and/or Formalization may be in progress before it is decided that sufficient Analysis has been done. The criteria for ending Analysis are therefore different from those for starting Draft Design or Formalization.

The classified information is regarded as a structured information base. There should be a checklist to determine whether the classified information is sufficiently structured into objects and whether these objects are adequately named and defined for Draft Design and Formalization to be completed. The checklist also allows the detection and the handling of possible inconsistency and incompleteness in the investigation of the collected requirements.

6 Draft Design

The general purpose of Draft Design of the system is to engineer partial or partly informal specifications from different points of view and at different levels of detail. Draft Designs do not need to be complete or strictly formal, but need to support the investigation of different designs for parts of the system so that engineering design choices can be made.

Use is made of the collected requirements and classified information, knowledge gained during Analysis and where possible models from the reuse library.

Figure 6-1 describes the input and the output of the Draft Design activity.

The engineer uses and elaborates the existing models to produce draft designs that provide a more formal information view and outline computational views of the system. Outline design views may also be included, if needed. The draft designs allow the behaviour of the system to be selectively investigated before systematic formalization is done in the Formalization activity. To a large extent Draft Design and Formalization can take place in parallel, as long as the draft designs to support Formalization are done before they are needed in Formalization steps. For example, draft designs produce typical Message Sequence Chart use sequences that may be used when skeleton processes are produced in Formalization.

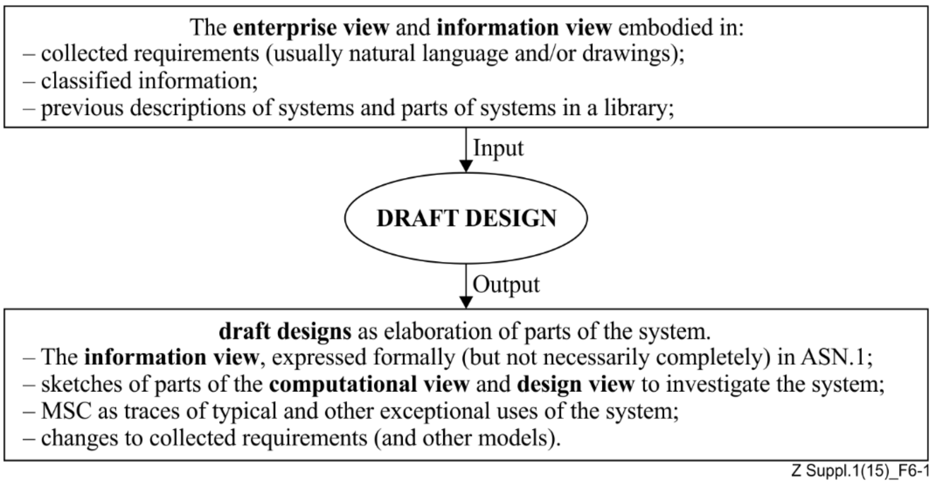

The **enterprise view** and **information view** embodied in:
– collected requirements (usually natural language and/or drawings);
– classified information;
– previous descriptions of systems and parts of systems in a library;
Input
DRAFT DESIGN
Output
draft designs as elaboration of parts of the system.
– The **information view**, expressed formally (but not necessarily completely) in ASN.1;
– sketches of parts of the **computational view** and **design view** to investigate the system;
– MSC as traces of typical and other exceptional uses of the system;
– changes to collected requirements (and other models).
Z Suppl.1(15)_F6-1

Figure 6-1 – Input and output of Draft Design

There is a difference in the way collected requirements are used in Draft Design compared with the way they are used in Analysis:

– Analysis focuses on the nouns (what objects are there?) in the collected requirements to structure and define concepts to produce the enterprise view and an information view in an informal way.

– Draft Design focuses on the verbs (what can the objects do?) in the collected requirements (or the verbs captured in the behaviour aspects of the classified information) to derive a computational view and, if needed, a design view in a formal way.

The classified information used in Draft Design is either the result of Analysis, or, if Analysis was not needed, the equivalent descriptions within the collected requirements. Although other languages or techniques may be used for the Analysis structuring, the content of the structure is often natural language. Draft Design elaborates the information view embodied in the classified information to be formal so that the data can be used to support behaviour. The information view produced in the Draft Design is formal and it is suggested that this be defined in ASN.1, because this allows standardized encodings to be used for the communication of information, in particular for normative interfaces.

Because the final objective of the Formalization activity is to produce a model and description using SDL+, Draft Design uses SDL+ whenever possible. The essential difference between a draft design in SDL+ and a formal model is coverage and completeness. For the purposes of Draft Design it is fine: to consider only typical cases and limited parts of the system, to assume that some data operators are well-defined, and to ignore some of the rules imposed by the formal languages. A Draft Design is a sketch for the specification, compared with a formal model that should be precise, unambiguous and interpretable.

So that all the models produced are traceable, draft designs are linked to the classified information and the collected requirements.

6.1 Starting Draft Design

Because there are varied reasons for producing draft designs, the objectives of Draft Design should be reviewed and recorded. The recorded objectives become the criteria in the checklist for adequate Draft Design.

The usual objectives are one or more of the following:

1) to provide a diagrammatic overview of the system as working documents for the engineers involved;
2) to identify critical parts of the system;
3) to investigate the feasibility of critical parts of the system;
4) to determine a typical set of use sequences;
5) to identify behaviour parts needed to support the service or protocol;
6) to determine needed behaviour parts (functional behaviour parts that will be normative);
7) to sketch a model including normative parts and non-normative parts, where the non-normative parts are usually added so that the model can be interpreted and behaviour can be analysed;
8) to clarify and identify the normative and non-normative interfaces, both at the boundaries of the system described and within the system described;
9) to identify search criteria to retrieve reusable parts (in SDL+) from a reuse library.

An objective that always applies is to identify missing items, inconsistencies and ambiguities in the classified information or collected requirements. Questions, the answers and issues to be further considered related to the classified information, are recorded during Draft Design. This can lead to changes to the classified information in the case that an engineering error has been identified, and to the collected requirements when these should be changed.

6.1.1 Determining whether to skip Draft Design

NOTE – This subclause defines how to evaluate information available to the engineer and the engineer's understanding of the system before Draft Design is started. Draft Design may not be necessary for the development of a system. In such a case, the development can proceed directly to Formalization, although in this case steps to produce ASN.1 and Message Sequence Chart diagrams from Formalization may still be needed.

Before the start of Draft Design, the classified information is evaluated to determine whether it is possible to skip Draft Design and start Formalization. Draft Design may not be necessary if several of the items in the following checklist are satisfied:

1) The category of system is well known and understood by the engineers involved.
2) The engineers involved are experienced in SDL+.
3) The classified information has a clear and direct relationship to system interfaces and an agent (block/process0 structure in SDL-2010.
4) The classified information already contains draft designs in SDL+ for at least typical cases.
5) There is already an information view expressed in ASN.1 in the classified information.
6) It is not expected that there will be any difficult sequencing or structuring issues in the SDL+.
7) It is not expected that multiple levels of abstraction need to be modelled before the formal SDL+ description is produced.
8) The architecture in which the SDL+ should fit does not impose any limitations or constraints that need investigation.
9) There are no feasibility issues that may depend on the SDL+ finite state machine model.

6.1.2 Criteria to start Draft Design

Before Draft Design is started, there shall be classified information that:

1) embodies an enterprise view of the system.
2) describes concepts and names for some parts of the system.
3) gives an information view of the main parts of the system.

6.1.3 Draft Design notations

The selected notations will depend on the object-oriented analysis technique used for Analysis (if this was not skipped) or the form of the input information. Suggested notations for Draft Design are SDL+ (used informally) and the notation used for object and classes to express more detailed entity relationships.

The selected notations should provide a sound basis for Validation and the specification of conformance tests. The notations should be selected to be useful for Formalization. For this reason, SDL+ should be used whenever possible. The introduction of any notation other than SDL+ or a notation used in the input should be avoided.

6.2 Draft Design steps

The general steps for Draft Design are:

1) Make a context model where the system is identified and the system environment is detailed, and describe the communication interfaces and other relations the system shall handle.
2) Make or add to Message Sequence Chart diagrams that describe use sequences; i.e., the typical interaction sequences (protocols), at each layer of the interfaces.
3) Sketch the system structure and identify parts that are subject to the requirements.
4) Extend the Message Sequence Chart diagrams to cover less usual situations that are at the boundaries of the requirements and possible faults.
5) Detail the information in the interfaces and define an information model for the system.
6) Define a prototype system (not necessarily formal or executable) or parts of the system that can handle the "interesting" use sequences (i.e., the ones to investigate).

Because one purpose of Draft Design is to sketch specifications to investigate approaches, some of the models produced will turn out to be unsuitable or unworkable. When draft designs do not lead directly to a formalized result, the cause of the problem is traced to either requirements or the design, and alternatives are tried. This leads to a repetition of one or more steps. Inevitably, some of the draft designs produced are subsequently abandoned. It should not be assumed that every draft design leads directly to the final formal SDL-2010 description. The advantage is that solutions that are not feasible or are unattractive can be investigated and abandoned at a reasonable cost.

The result of each step should contain "links" back to the classified information and collected requirements, and a record of questions arising from the step. Against each question there is either an answer or a mark that the question is still unresolved. There are also likely to be some rejected draft designs. The reason for rejecting each draft design is recorded, and the model in the Goal Requirements Language (GRL) or URN [b-ITU-T Z.151] is useful for this purpose.

6.3 Conclusion of Draft Design

When Draft Design has been done, there is a set of results as defined below. These shall be reviewed to determine whether the Draft Design activity can be considered complete.

The data in the system, and structure and interfaces of the system should be sufficiently well described and understood so that:

1) the objectives of the draft design as recorded at the start of Draft Design are met;

2) formalization can be completed.

7 Formalization

The general purpose of Formalization is the production of a formal specification for the system to be used as the application and for Validation. Use is made of all the classified information or draft design models as shown in Figure 7-1, of knowledge gained in Analysis and Draft Design, and, where possible, of models from the reuse library. Some mappings between the models from Analysis and Draft Design to the formal SDL-2010 description are suggested in guidelines in the detailed Formalization steps in clause 15.

The engineer formalizes and generates refined designs to identify, understand and analyse the application at a detailed level. The work is guided by the use of SDL+. Figure 7-1 describes the input and output of the Formalization activity.

The classified information referenced in Formalization is either the result of Analysis, or (if Analysis was not needed) the equivalent descriptions within the collected requirements. Similarly, references to draft designs are either the results of the Draft Design activity or (if the draft design was not needed) equivalent descriptions in the classified information (or the collected requirements).

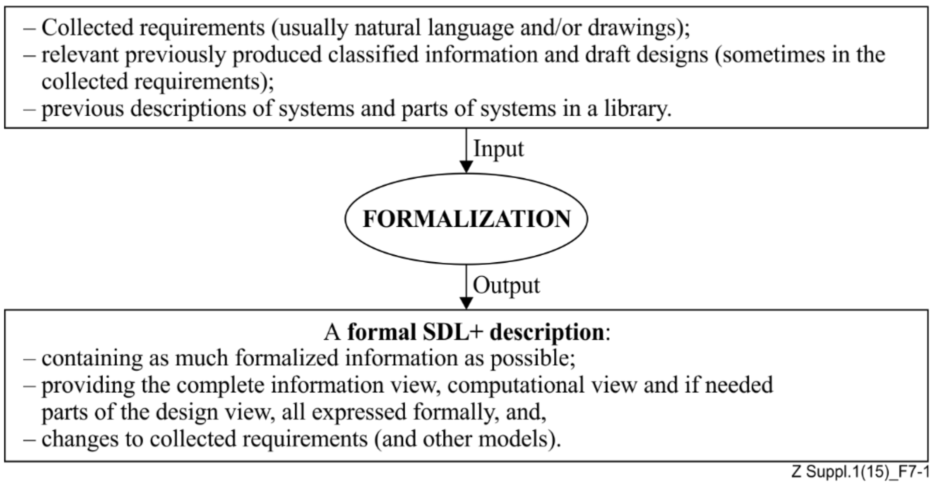

– Collected requirements (usually natural language and/or drawings);
– relevant previously produced classified information and draft designs (sometimes in the collected requirements);
– previous descriptions of systems and parts of systems in a library.
Input
FORMALIZATION
Output
A **formal SDL+ description**:
– containing as much formalized information as possible;
– providing the complete information view, computational view and if needed parts of the design view, all expressed formally, and,
– changes to collected requirements (and other models).
Z Suppl.1(15)_F7-1

Figure 7-1 – Input and output of Formalization

7.1 Starting Formalization

At the start of Formalization, the objectives, available inputs and use of formalisms shall be reviewed.

The main objectives of Formalization are:

– actual generation of the formal SDL+ description;

– investigation of the system guided by the use of SDL+:

  Understanding increases and investigation takes place as the SDL+ is produced. If Analysis or Draft Design is omitted, then investigation during Formalization is more important. The guidelines associated with the Formalization steps cover the investigation that is required.

– consistency and inconsistency handling:

  The creation of the SDL+ description can find inconsistencies and ambiguities in the draft designs, classified information and collected requirements. Questions, answers, and items to be considered related to each of these inputs are recorded during Formalization. This can lead to changes in any of the inputs and (except where an engineering error has been made in Draft Design or Analysis) the collected requirements should be changed.

Before Formalization is started, the following criteria should be met:

1) There shall be classified information that embodies an enterprise view of the system.
2) There shall classified information that describes concepts and names for the main elements of the system.
3) There shall be classified information that embodies interface descriptions for normative interfaces.
4) There should be draft designs that embody the main structural elements of the system, such as the functional model used in the ITU-T Q.1200-series of standards [b-ITU-T Q.1200].
5) There should be draft designs that embody the main elements of an information view of the system.

7.2 Formalization steps

The steps to follow during Formalization will depend to some extent on techniques used for Analysis or Draft Design and the context in which SDL+ is being used. The formal specification will be in SDL+; therefore, the steps do not depend on the notation used. Nevertheless, the capabilities of tools may make the use of some SDL+ constructs impractical and there may be other constraints (such as customer requirements) that cause variation in the use of SDL+. However, in all cases it is necessary to define the structure, behaviour and data of the system and the Formalization steps can focus on these issues. Steps can also be added to exploit the type facilities of SDL-92. Furthermore, if Message Sequence Chart diagrams have not already been produced during Draft Design, then they will be needed to help the SDL-2010 process formalization.

Since there is little dependence on context, the Formalization steps given in clause 15 can be used as the basis for a set of steps for any context. These are subdivided into: Structure, Behaviour, Data, Type and Localization steps. The latter two groups are applied when parts of the system can be reused. When these steps are used as a basis for another context, then each step will need to be reviewed, some steps may be deleted and steps may be added.

Classified information and draft designs provide the understanding and information needed for Formalization. Even if they are not explicitly mentioned in a particular step, they are always used as an input. Draft designs should be in SDL+, in which case some of the Formalization may be trivial. These steps are trivial because some of the descriptions may need only to be checked as correct according to the language rules and any rules given for each step, and changed if needed. Usually, however, the draft designs are incomplete and informal so that they only cover part of the system.

Reference to the use of draft designs and classified information can usually be found in the guidelines. Some of the draft design (such as those producing detailed Message Sequence Chart diagrams) will probably be done in parallel with Formalization.

The result of each step should include "links" to the questions, decisions, collected requirements, classified information and draft designs related to the step. This allows the SDL+ description to be traced back to other descriptions.

7.3 Conclusion of Formalization

A complete model of the system in SDL-2010 is produced by Formalization. Together with the other documentation from the Analysis and Draft Design, this model provides a complete specification of the system. The model embodies the computational view of the system and those aspects of the design view of the system that need to be within the application.

The Formalization steps are intended to produce an executable model, except possibly for some sorts of data that can usually be made executable by using a support tool interactively or with the data in a programming language. The benefit of an executable model is that by running the model, the feasibility and functionality of the system can be demonstrated. The disadvantage of this model is that some aspects of the system have to be explicitly defined so that the system is executable. For example, data passed through the system without modification has to be modelled in some explicit way (for example, a character string) so that the model can be executed, but for the application this may be the abstract concept of a data unit.

The SDL+ model is executable so that execution of the model can be used in Validation and the model can run against conformance tests to ensure that these are compatible with the model.

The result of Formalization should be checked against the following criteria:

1) The quality attributes for the system should be satisfied by the formal SDL+.
2) The formal SDL+ shall conform to SDL-2010 ([b-ITU-T Z.100], [b-ITU-T Z.101], [b-ITU-T Z.102], [b-ITU-T Z.103], [b-ITU-T Z.104], [b-ITU-T Z.105] and [b-ITU-T Z.107]) for the Specification and Description Language part that optionally can be presented in the Common interchange format for SDL-2010 (CIF) defined in [b-ITU-T Z.106], and [b-ITU-T Z.120] for the Message Sequence Chart part.
3) The formal SDL+ description shall be consistent with the draft designs.
4) The SDL+ shall conform to any rules in the Formalization steps.
5) It shall have been shown (by a thorough design review or audit, and by execution with defined input sequences) that the formal SDL-2010 description is a satisfactory model of the functionality of the system described by the collected requirements.

8 Implementation

For some applications, the amount of engineering to be done to transform an SDL+ formal model into an implementation that can be used may be small or even trivial (such as compiling the SDL+ for the target hardware), but conceptually the initial SDL+ model and the implementation are quite different levels of abstraction. As derived from the methodology in this Supplement, the *formal SDL+ description* in Figure 4-3 will ignore non-functional implementation requirements and constraints that have to be included in the *product* in the same figure. If the formal SDL+ description is very abstract, it may be possible to refine the description to produce another formal SDL+ description that also allows for more product requirements and constraints. When there are large number of constraints or significant constraints (such as performance or distribution issues), then the SDL+ can be refined iteratively to produce functionally equivalent, but operationally different, systems.

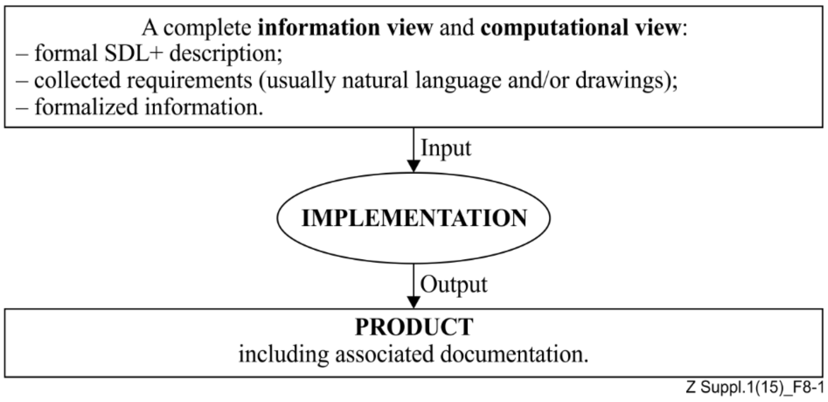

A complete information view and computational view:
– formal SDL+ description;
– collected requirements (usually natural language and/or drawings);
– formalized information.
Input
IMPLEMENTATION
Output
PRODUCT
including associated documentation.
Z Suppl.1(15)_F8-1

Figure 8-1 – Input and output of Implementation

There will be some constraining issues (such as interfacing with the underlying system) that are either inexpressible in SDL+ or can be more efficiently expressed in another language (for example C++, or operating system calls, or assembly language). It may also be the case that some of the SDL-2010 is not implemented in software (in the conventional sense of the word). These situations do not mean that the SDL+ design has to be completely rewritten into another language, because there are tools available that translate SDL-2010 into other languages (typically C++ or Java, but also hardware languages) whilst maintaining the SDL-2010 as the system description. These tools allow linking to parts written in other ways.

Implementation is more application and organization dependent than other activities and varies widely. Only a broad outline is given here. The main steps to consider are:

1) Perform the trade-off between hardware and software (the overall architecture).
2) Determine the hardware architecture to support the software (the hardware architecture).
3) Finalize the software architecture.
4) Restructure and refine the SDL+ description adding descriptions as necessary to define the product.

The SDL+ description can be taken as a starting point together with miscellaneous aspects in the classified information.

Key factors in the overall architecture are: requirements for physical distribution and the implied communication bandwidths between distributed items; performance requirements including overload situations, time critical responses and average throughput; reliability and redundancy requirements. There may be some choice of what to put where and what is implemented in hardware. Sometimes the system is required to be all in software. The economics of hardware (usually more expensive but faster) versus software will have to be considered. Utilizing existing hardware or software designs or equipment may be a factor.

The hardware architecture needs to be defined in terms of the number of processors and other hardware units, the number of connections and the hardware to support transmission, the performance characteristics of the hardware items such as the processing power and delays on physical connections. If the hardware system is defined in the requirement, it will be important to determine if this system is actually capable of supporting the software and meeting the performance constraints.

The software architecture is an elaboration of an SDL+ description. The elaboration may require some restructuring of an SDL-2010 description to match the distribution of the SDL-2010 across physical units. This may require turning some agents (blocks/processes) in the SDL-2010 system into cooperating SDL-2010 systems. A run-time system to support the SDL-2010 will be needed, and there needs to be a mechanism (possibly part of the run-time system) that converts external events into SDL-2010 signals.

More detailed guidelines can be found in [b-Bræk], [b-Mitschele-Thiel] and [b-TIMe].

9 Validation

Validation has two aspects:

– checking that the syntax and semantics of a specification are correct

– checking that the known requirements are expressed by the specification.

Although validation is frequently accomplished through expert review, one of the principal reasons for expressing a specification in SDL+ is to enable computer-assisted validation. The first aspect of validation is well suited to automation. Conformance to the rules of SDL+ ensures that a specification is self-consistent and contains no unintended ambiguity, and SDL+ tools automate the checking of syntax and static semantics. To check the dynamic semantics usually requires the SDL+ model to be executed, and some test cases will be needed to exercise the model. The dynamic semantics may also be validated by such techniques as state space exploration.

The second aspect of Validation has been more difficult to automate. Formal proof that a specification meets requirements has so far been automated only for simple protocols. It is nevertheless possible to improve confidence that the specification meets requirements with automated techniques, including simulation and the application of tests.

Defining a system with SDL+ resembles to a large degree writing software in a programming language. Like software programs, the formal SDL+ description needs to be *tested*, because there is a high probability it will initially contain engineering errors and does not provide the known requirements. Such errors arise despite the application of checks during the creation of the SDL+, because the engineering itself cannot be fully automated and humans make mistakes. Applying the tests as part of Validation resembles testing of a concurrent program. Unlike a concurrent program, which is supposed to run on a real machine in a real environment, the SDL+ system runs on an abstract machine with basically unlimited resources, otherwise testing SDL+ is similar to testing software. Such testing contributes to both the aspects of Validation described above.

To make the SDL+ executable on a practical machine so that the tests can be applied may require the SDL+ model (and the tests) to be modified. The modified model is a validation model. Similarly, a non-execution technique (such as state space exploration) to validate SDL+ by automated examination may also require a modified validation model to be derived from the SDL+. Validation by testing or by the application of other tools to validation models is called *formal validation*.

The various activities can produce many definitions, one refining the other or describing the system from a different viewpoint. The definitions may be using different specification techniques such as natural language, Message Sequence Chart notation, ASN.1 and TTCN. The validation process makes sure that the specifications are in line with each other. It is important to note here that the SDL+ usually does not exist in isolation but in a context, and therefore has to be validated within this context.

Not only can a number of different notations be used, a system definition may also make reference to other specifications, such as standards, without explicitly incorporating them. In these cases, one or more formal validation models have to be developed before correctness can be checked by formal means. If a specification contains options, a particular set of them has to be chosen in order to make the model executable. Figure 9-1 shows the general validation scheme; circles represent activities and rectangles represent documents.

Formal validation uncovers errors that developers trace to either the validation model or the specification. After this diagnosis is made, the validation model is revised, the specification is also revised if necessary, and the validation model is re-executed.

The methodology in this clause covers only the derivation and execution of the validation model, which is only part of the Validation shown in Figure 4-3. Some of the checks needed for Validation are covered by Analysis, Draft Design and Formalization. Other checks that are not covered by this Supplement can be derived from the application of quality assurance techniques for software (see [b-ISO 90003]).

9.1 Characteristics of a validation model

A validation model has two essential characteristics:

– The SDL+ is sufficiently detailed that the model is executable.

– It is practical to use the model to exercise boundary cases (for example, the number of circuits, calls, and other types of traffic is limited; the size of data items is limited).

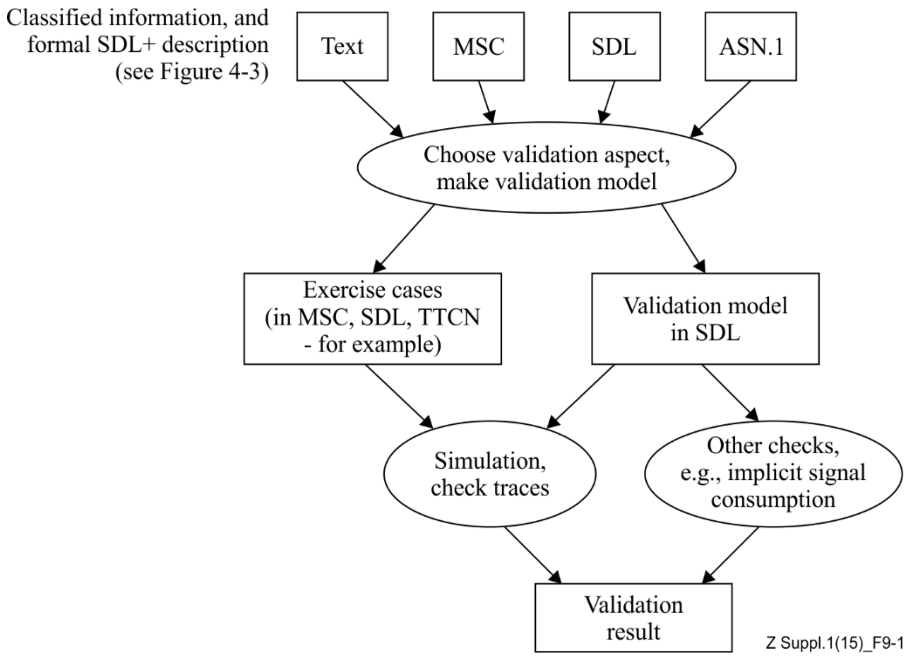

Classified information, and formal SDL+ description (see Figure 4-3)
Text
MSC
SDL
ASN.1
Choose validation aspect, make validation model
Exercise cases (in MSC, SDL, TTCN - for example)
Validation model in SDL
Simulation, check traces
Other checks, e.g., implicit signal consumption
Validation result
Z Suppl.1(15)_F9-1

Figure 9-1 – General Validation scheme

9.2 Comparison of the validation model with the formalized model

When the methodology is being used to produce an SDL+ description for a standard, then it is the SDL+ description in the standard that is of interest: the formalized standard.

Both the validation model and a formalized standard are derived from the formal SDL+ model produced during the Formalization activity (the formalized model). The assumption is that the formalized standard is generalized from the formalized model. For example, the formalized standard can omit pieces that are required from a technical standpoint but are not part of the standard; or the formalized standard may allow options, at least one of which needs to be implemented to make the validation model executable. It is often desirable that a validation model includes parts of the environment.

The relationship between the validation model and the formalized standard is that the validation model consists of the SDL+ in the formalized standard and a minimum of modification is necessary to make the model executable. These modifications can be derived from the formalized model, on the assumption that the formalized model is executable, and in some cases the validation model can be identical to the formalized model.

When the methodology is being used to produce an implementation, the validation model may still need to differ from the formalized model, because either the formalized model needs some implementation support to be executable, or the implementation boundaries are too large for validation. In the latter case, validation can usually be done using smaller limits.

If the formalized model is not executable, or if it is executable but not practical for validation, a separate validation model is derived.

Instructions

1) Define new boundaries for the validation model, including parts of the environment needed to make the validation model executable.
2) Define limits for circuits, calls and other types of traffic.
3) Identify exercise cases or observers.

Guidelines

The building of the validation model can be divided into:

1) identifying the scope of validation;
2) choosing a particular set of implementation options;
3) making the system complete;
4) optimizing the model to make verification practical.

More detailed guidelines can be found in [b-Hogrefe].

9.3 Issues in defining the validation of a specification

During Validation, more than one validation model may be produced, depending on the characteristics of the application and the automated techniques selected for executing the model. For example, when a protocol specification in an application is quite complex, one option is to partition it and validate each part. This and other tactics for simplifying specifications are necessary if a complex specification is to be validated through reachability investigation, which requires that all possible states be evaluated during execution of the model. On the other hand, one of the advantages of selecting random state space validation when a protocol specification is complex is that the validation model can represent the entire protocol.

10 Relationship with other methodologies and models

The methodology in this Supplement can be used in combination with other methodologies. As usually applied, the methodologies described below result in one or more semi-formal models. Most of the models use the draft designs produced by the Draft Design activity, but with additional characteristics resulting from the other methodology. For example, the methodology of [b-ITU-T I.130] generates three models: a model from a user's viewpoint, another model showing the organization of network functions, and a third model showing switching and signalling capabilities to support the user model.

10.1 Relationship with Recommendations I.130/Q.65 (3-stage method) 1988

Because Recommendation [b-ITU-T I.130] is commonly used as a basis for local methodologies, this is taken as a context for the elaboration of the steps of the methodology in Part II.

Use of [b-ITU-T I.130] and [b-ITU-T Q.65 1988] has not been limited to the "characterization of telecommunication services supported by an ISDN and network capabilities of an ISDN" scope indicated in the title of [b-ITU-T I.130]. In [b-ITU-T Q.65 2000] it is noted that the method has been generalized beyond ISDN to include services provided in and among networks of various types. Stage 1 and stage 2 are used to produce standards for service functionality, which are then used in stage 3 to produce standards for protocols to support the service. In the three stages of the [b-ITU-T I.130] method, the ITU-T Specification and Description Language is used:

– in stage 1, step 1.3, for the dynamic description of a service where the "information is presented in the form of an overall Specification and Description Language (SDL) diagram" (see clause 3.1 of [b-ITU-T I.130]);
– in stage 2, step 2.3, where "functions performed within a functional entity" are "represented in the form of a Specification and Description Language (SDL) diagram" in (clause 3.2 of [b-ITU-T I.130]);
– in stage 3, where "SDL diagrams from the stage 2 form the basis" (clause 3.3 of [b-ITU-T I.130]) and "Recommendations on SDL (Z.100-Series)" are shown as the "tools of the method, description techniques (models) and the associated library of generic material".

Sometimes all three Specification and Description Language descriptions appear in standards and sometimes only one. Sometimes the different descriptions appear in different standards in a set of standards. The decisions of which descriptions should appear in which standards and what needs to be standardized (the user's viewpoint, the functional implementation of the service, and/or the access and inter-node protocols and procedures) are the concern of the group responsible for the standard and beyond the scope of this Supplement. This Supplement assumes that each Specification and Description Language description appears in a separate standard.

When [b-ITU-T I.130] was first drafted, the Recommendation for Message Sequence Chart [b-ITU-T Z.120] diagrams did not exist; therefore, "information flows" in [b-I.130] should now be replaced with the use of Message Sequence Chart diagrams.

The [b-ITU-T I.130] approach is consistent with the SDL+ methodology. What descriptions should be provided is defined by [b-ITU-T I.130] and the SDL+ methodology gives detail of how to produce these descriptions. The stage 1 description of [b-ITU-T I.130] is produced by making a context diagram and then elaborating this with a simple SDL-2010 description that can be considered as a draft design for the functional entity SDL-2010 in stage 2. The stage 2 SDL-2010 can be produced by applying all activities in this methodology. The stage 3 SDL-2010 can be produced by a reapplication of this methodology, using the stage 1 and stage 2 as part of the collected requirements.

Recommendation [b-ITU-T Q.65 1988] gave more detail for the method of stage 2. In step 2.1, a functional model is drawn that shows the relationship between the functional entities. This should be redrawn as an SDL-2010 system diagram with the functional entities as block types, functional entity instances as blocks and the relationships as channels. Because the SDL-2010 in a standard should be complete, the SDL-2010 version of this diagram should appear in the standard (if this includes an SDL-2010 model). It can be useful to include the functional entity model in the standard as well, because it does not include all the information in the SDL-2010 diagrams and therefore gives a more abstract view of the system. The information flow diagrams ([b-ITU-T Q.65 1988] Figure 4 and [b-ITU-T Q.65 2000] Figures 8 and 25) are Message Sequence Chart diagrams with a functional entity instance (corresponding to an SDL-2010 block) for each instance axis.

Although [b-ITU-T Q.65 2000] and [b-ITU-T Q.65 1988] use the same Recommendation number, [b-ITU-T Q.65 2000] is not really a replacement for [b-ITU-T Q.65 1988]: the [b-ITU-T Q.65 1988] document is stage 2 of a method ("The overall *method* for deriving switching and signalling Recommendations for ISDN services consists of three stages and is described in general in Recommendation I.130."), whereas [b-ITU-T Q.65 2000] is an architecture ("… a common functional *architecture* for providing services and addressing signalling requirements for service implementation."). In both cases the objective is to produce a description in the Specification and Description Language: in step 4 (clause 2.4 of [b-ITU-T Q.65 1988] and in clause 2.5 of [b-ITU-T Q.65 2000]); so that [b-ITU-T Q.65 2000] includes many elements of [b-ITU-T Q.65 1988].

When a formal SDL-2010 description is produced for stage 2, the functional entity actions are typically redundant, because the functions are completely described by the SDL-2010.

These functional entity action descriptions are sometimes still useful to help explain the SDL-2010 and some existing standards have used them extensively. If functional entity action descriptions are used in addition to the SDL-2010 diagrams, they should be clearly marked as informative.

10.2 Relationship with OSI layered modelling

This clause updates SDL-92 material that was first published in [b-Belina et al.1], [b-Belina et al.2] and in Appendix I clause 1.2.1 of [b-ITU-T Z.100a].

The OSI concepts used in this clause are defined in [b-ITU-T X.200] and [b-ITU-T X.210]. In order to make the clause 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 10-1). A service primitive can be used for connection management (connection, disconnection, resetting, etc.), or can 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).

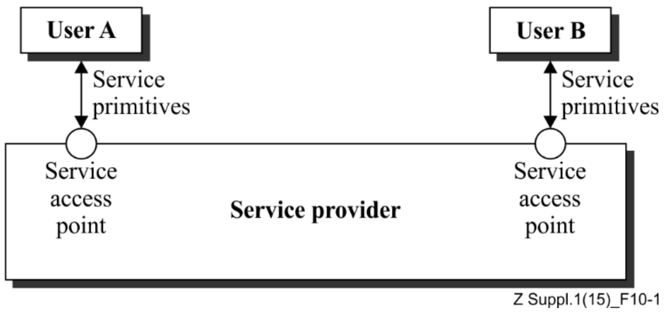


Figure 10-1

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 together provide a refinement of the (N)-service provider (see Figure 10-2).

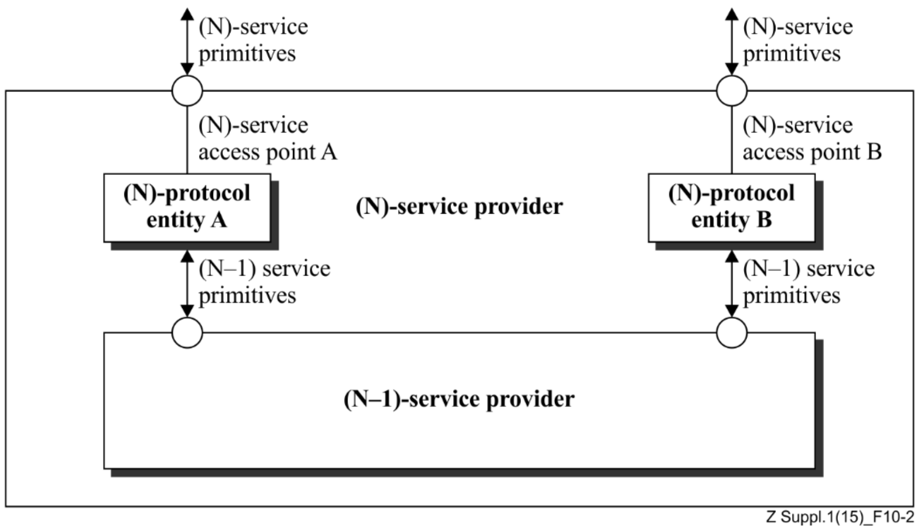


Figure 10-2

The refinement of the (N)-service provider shown in Figure 10-2 may 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.

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 reorder 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-2010 in a number of alternative ways, mainly depending on what aspect should be emphasized.

In the examples, the graphical syntax of SDL-2010 is used as far as possible. However, for practical reasons the example given is very simple and does not correspond to any real OSI layers.

10.2.1 Basic approach

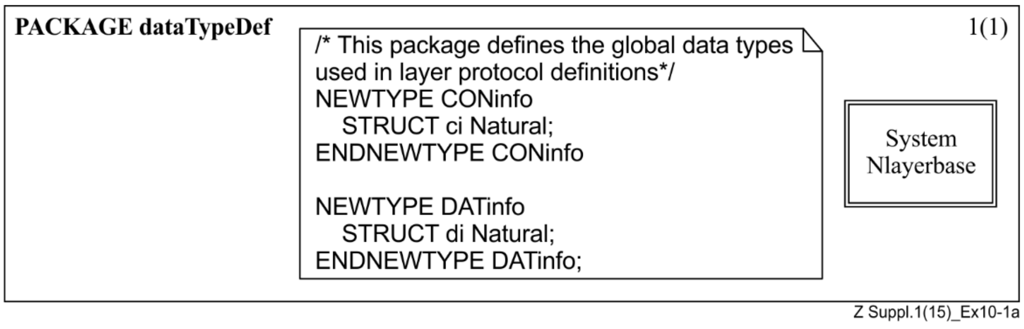
The objective is to describe the system in a layered way, so that in this example at the highest abstraction level only the interfaces with the system are shown, the next abstraction level shows service behaviour, the next level shows a coding level and so on. At the highest abstraction level the interfaces with the system are defined. At lower levels of abstraction the way the system behaves is defined, and in each case a Specification and Description Language **SYSTEM** is defined with the same interfaces. So that the definitions at higher abstraction levels can be reused at lower levels of abstraction, in each case the **SYSTEM** definition is given as an instance of a **SYSTEM TYPE** that inherits properties from a **SYSTEM TYPE** defined for a higher level.

NOTE – In SDL-92 and earlier versions of the Specification and Description Language, a "substructure" was allowed for blocks or channels. When the SDL-92 **SYSTEM** definition was instantiated, for each substructure a choice had to be made whether to use the block/channel or the block/channel substructure. Such an SDL-92 definition therefore defined many different systems, depending on the choices made. How to make the choices was not defined by the SDL-92 and has to be provided in some way by the user. The substructure mechanism was discontinued in SDL-2000. Instead for each **SYSTEM** a **SYSTEM TYPE** is used that inherits from a higher-level **SYSTEM TYPE**, and a **VIRTUAL BLOCK TYPE** is **REDEFINED** for the substructure (see below).

Service interface specification

In the example (Examples 10-1a, 10-1b and 10-1c), the users of this service (User A and User B) are in the environment of the system and can be considered as processes, capable of communicating with the system by means of signals to and from the system.

A **SYSTEM TYPE** has to be defined within a package. For convenience, the base **SYSTEM TYPE** for the service specification *Nlayerbase* is defined within the same package *dataTypeDef* as the data sort specifications (other than for predefined sorts) needed for all the descriptions of the system at different levels of abstraction (see Example 10-1a).



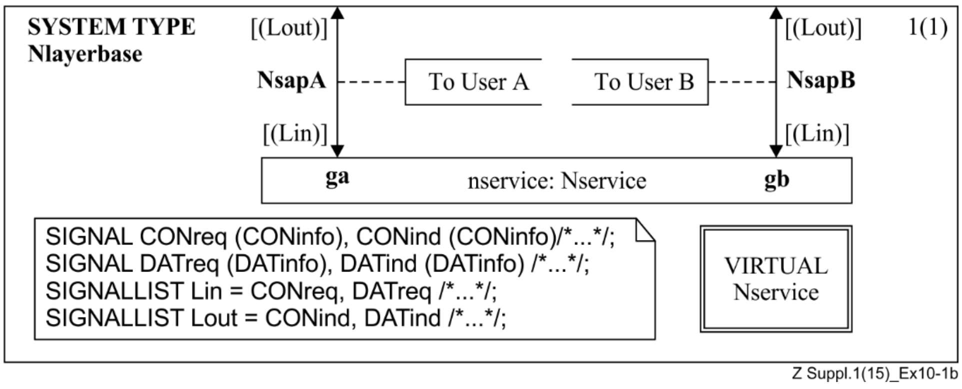
```
/* This package defines the global data types
used in layer protocol definitions*/
NEWTYPE CONinfo
   STRUCT ci Natural;
ENDNEWTYPE CONinfo

NEWTYPE DATinfo
   STRUCT di Natural;
ENDNEWTYPE DATinfo;
```

Example 10-1a – PACKAGE *dataTypeDef*

The *Nlayerbase* is modelled in a straightforward manner as connecting to block *Nservice* (see Example 10-1b).

Each service access point is represented by a channel (*NsapA* and *NsapB*) conveying signals, which represent service primitives, to and from users A and B in the environment. A signal is capable of carrying a value of the sort of data given in the signal specification: for example, the *CONreq* signal has a *CONinfo* value. For a real service there would probably be many more kinds of signal (as implied by the /*…*/ comments to the **SIGNAL** definitions). The **SIGNALLIST** *Lout* and **SIGNALLIST** *Lin* define the signals conveyed on the interfaces and /*…*/ again implies that for a real service there would probably be many more kinds of signal. As suggested by the signal names, the example is a (very simple) connection-oriented service. In general a connection-less service is simpler.

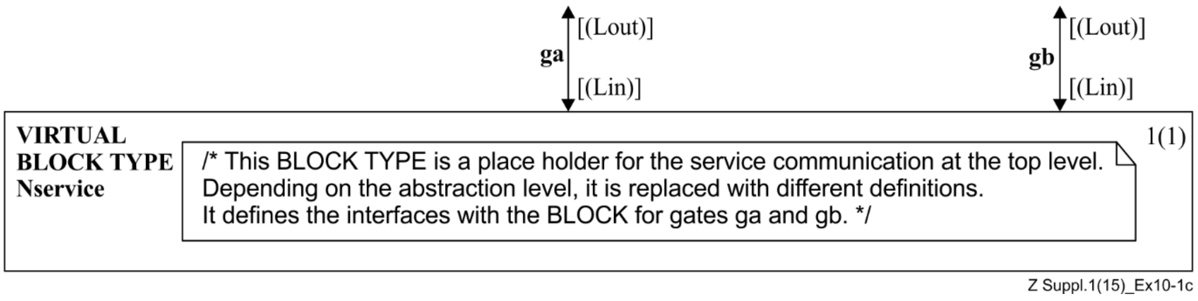


```
SIGNAL CONreq (CONinfo), CONind (CONinfo)/*...*/;
SIGNAL DATreq (DATinfo), DATind (DATinfo) /*...*/;
SIGNALLIST Lin = CONreq, DATreq /*...*/;
SIGNALLIST Lout = CONind, DATind /*...*/;
```

Example 10-1b – SYSTEM TYPE *Nlayerbase*

The **BLOCK TYPE** *Nservice* (see Example 10-1c) is empty at the service interface level, because this level does not define how the system behaves. However, it defines gates *ga* and *gb* so that when *Nservice* is redefined at lower levels of abstraction, these handle the same signals. The **VIRTUAL** attribute of a **TYPE** like *Nservice* allows it to be **REDEFINED** when the enclosing type is inherited.

Instantiation of *Nlayerbase* is not meaningful because it does not describe how the system behaves, therefore no **SYSTEM** definition is given.

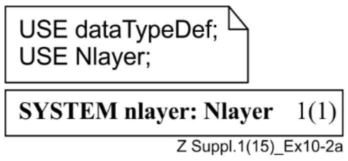


[(Lout)]
ga
[(Lin)]
[(Lout)]
gb
[(Lin)]
VIRTUAL BLOCK TYPE Nservice
1(1)
/* This BLOCK TYPE is a place holder for the service communication at the top level. Depending on the abstraction level, it is replaced with different definitions. It defines the interfaces with the BLOCK for gates ga and gb. */
Z Suppl.1(15)_Ex10-1c

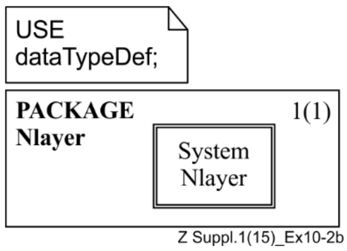
Example 10-1c – BLOCK TYPE *Nservice*

Service specification

The **SYSTEM** *nlayer* (see Example 10-2a) is an instantiation of **SYSTEM TYPE** *Nlayer* defined in **PACKAGE** *Nlayer* (see Example 10-2b) that gives the highest abstraction view of how the system behaves. The data type definitions from *dataTypeDef* are used.

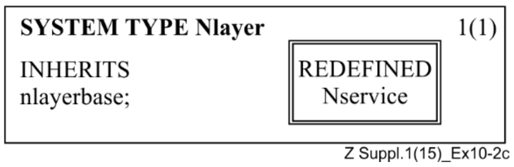


Example 10-2a – SYSTEM *nlayer*



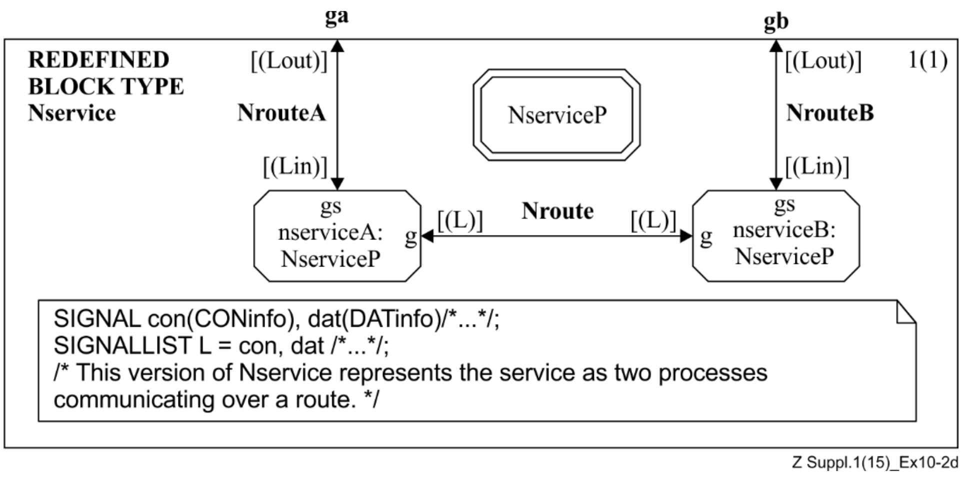
Example 10-2b – PACKAGE *Nlayer*

The **SYSTEM TYPE** *Nlayer* inherits **SYSTEM TYPE** *Nlayerbase* and redefines *Nservice* (see Example 10-2c), so the channels for the service access point are unchanged.



Example 10-2c – SYSTEM TYPE *Nlayer*

Only two processes are considered at this level of abstraction of the example, one for each service access point. The redefined **BLOCK TYPE** *Nservice* (see Example 10-2d) adds the service access point processes *nserviceA* and *nserviceB*, which communicate via *Nroute* to define how the service behaves. The service is symmetrical and the two processes behave in the same way, so they are both based on the same **PROCESS TYPE** *NserviceP*. These processes communicate with each other by signals (*con*, *dat* /*...*/), 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*. Two processes are used because in general an originating process does not communicate with itself (in this specification that is not allowed), and also allows the scenario of a collision (dual-seizure) to be described.

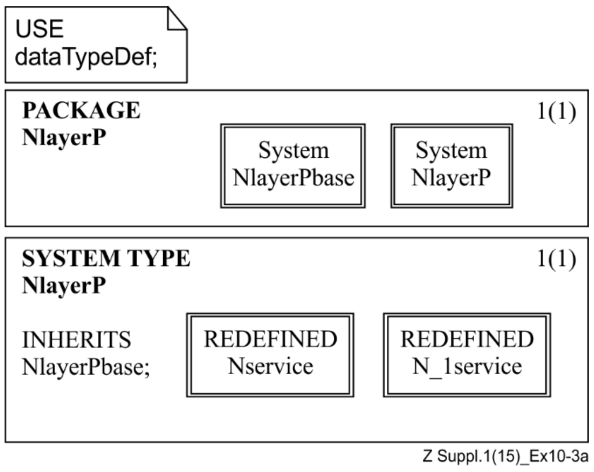

ga
gb
REDEFINED BLOCK TYPE Nservice
[(Lout)]
NrouteA
[(Lin)]
NserviceP
[(Lout)]
NrouteB
[(Lin)]
1(1)
gs nserviceA: NserviceP
g
[(L)]
Nroute
[(L)]
g
gs nserviceB: NserviceP
SIGNAL con(CONinfo), dat(DATinfo)/*...*/;
SIGNALLIST L = con, dat /*...*/;
/* This version of Nservice represents the service as two processes communicating over a route. */
Z Suppl.1(15)_Ex10-2d

Example 10-2d – BLOCK TYPE *Nservice*

The process graph for *NserviceP* is deliberately omitted here, because it would distract from the main issue: how to describe layered systems.

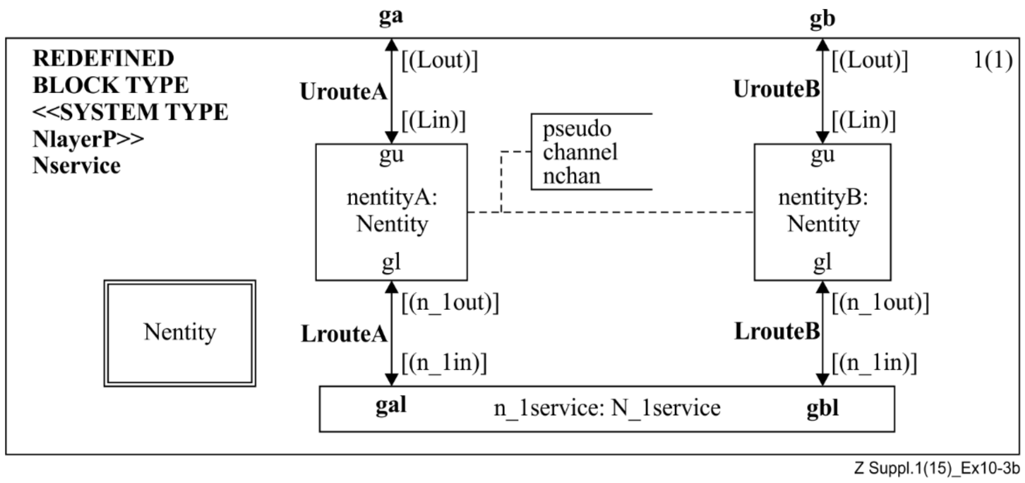
"Protocol" specification – layer N

The "protocol" specification is modelled by a redefinition of the block *Nservice* in the **SYSTEM TYPE** *NlayerP* that otherwise inherits the next higher-level **SYSTEM TYPE** *Nlayerbase*. Similar to *Nservice* at the higher level, at the *NlayerP* level the next layer service is introduced in **SYSTEM TYPE** *NlayerPbase* as an empty VIRTUAL **BLOCK TYPE** *N_1service* to define the interfaces for *N_1service*, and **SYSTEM TYPE** *NlayerPbase* also defines any data types, signals and signallists to interface with *N_1service*. It is *NlayerPbase* that directly inherits *Nlayerbase*. *NlayerP* inherits *Nlayerbase* indirectly by inheriting *NlayerPbase*. Rather than have different packages for *NlayerPbase* and *NlayerP*, these are both included **PACKAGE** *NlayerP*. The **SYSTEM** *nlayerP* is an instance of *NlayerP* (see Example 10-3a).



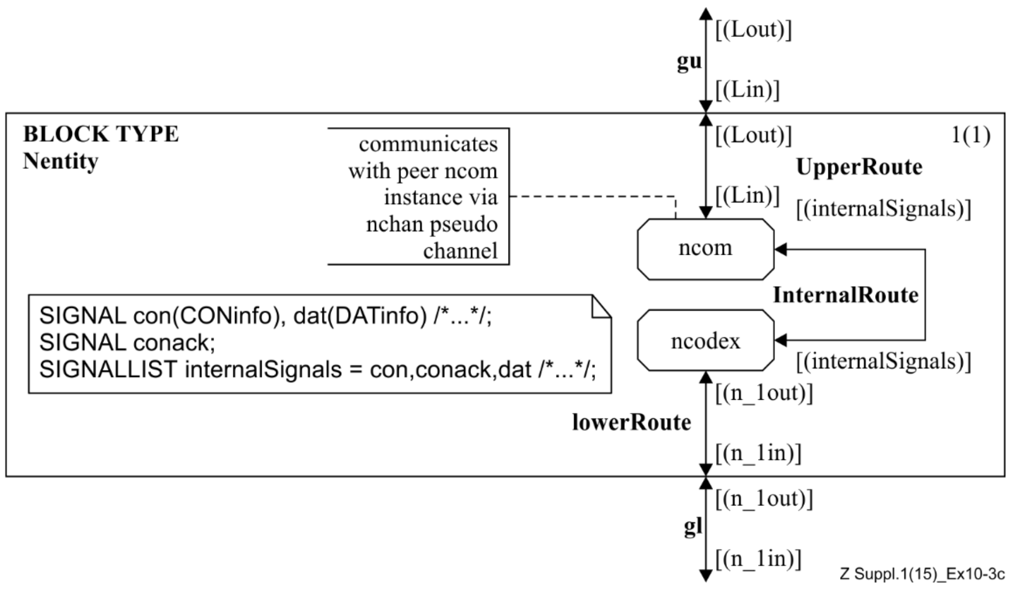
Example 10-3a – *nlayerP* based on *NlayerP*

The specification of the redefined *Nservice* (see Example 10-3b) has 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.



Example 10-3b – BLOCK TYPE *NService*

Each (N)-protocol entity block is an instance of *Nentity* (see Example 10-3c) and contains one or more processes, depending on the characteristics of the protocol.

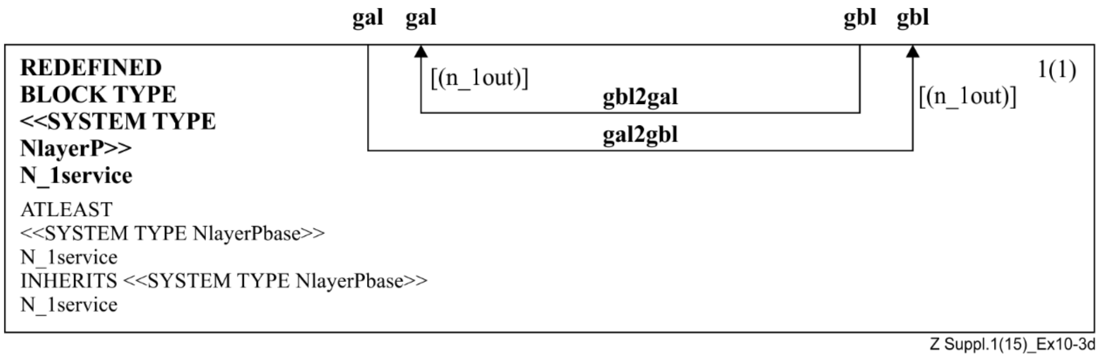


Example 10-3c – BLOCK TYPE *NEntity*

In this case, the choice is that *Nentity* contains two processes: *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 *N_1service*. Conceptually, the processes *ncom* communicate directly via a pseudo channel nchan (conveying protocol data units – signals *con*, *conack*, *dat* … defined in *Nservice*), but in reality they communicate indirectly via the *ncodex* process instances of *Ncodex* through the underlying service. *Ncodex* transforms the protocol data units at this level to the N-1 service units (*byte* signals here) for the underlying *N_1service*. Normally the N-1 service would also require some information to establish a connection to deliver the service units to the correct *Ncodex* instance, but for this simple case the *N_1service* is a simple A to B connection. The *Nservice* would also normally be more complex than shown here, for example having an additional interface for management control.

For **SYSTEM TYPE** *NlayerP* the *n_1Service* passes signals received on each gate to the other gate, so that in Example 10-3d for *N_1service* the two gates are simply connected by communication paths *gbl2gal* and *gal2gbl*. No signals are lost because **SIGNALLIST** *n_1out* is the same as **SIGNALLIST** *n_1in*. The ATLEAST clause in Example 10-3d constrains *n_1Service* only to *n_1Service* of *NlayerPbase*, so that the communication paths *gbl2gal* and *gal2gbl* are not inherited when *n_1Service* is redefined in less abstract models based on *NlayerP*. Without the ATLEAST clause *n_1Service* redefinitions have to be based on the *n_1Service* of *NlayerP* and *gbl2gal* and *gal2gbl* are inherited.

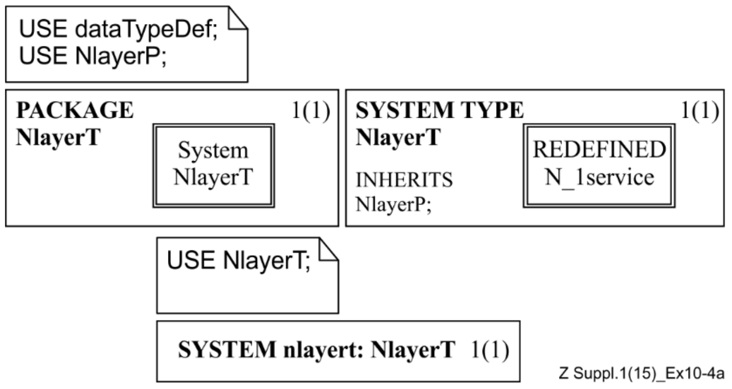
NOTE – Some tools do not allow the gates of a **BLOCK TYPE** to be connected by an internal channel. One alternative (for a model rather than an implementation) is to have two processes in *n_1Service*, each of which takes each signal from one gate on *n_1Service* and echoes it to the other gate on *n_1Service*. Another alternative is to connect *nentityA* to *nentityB* by a channel in **BLOCK TYPE** *Nservice*, but this would has the disadvantage that **BLOCK TYPE** *Nservice* then needs to be redefined again for a **SYSTEM TYPE** that describes a detailed behaviour for the (N-1) service.


gal
gal
gbl
gbl
REDEFINED BLOCK TYPE <<SYSTEM TYPE NlayerP>> N_1service
ATLEAST <<SYSTEM TYPE NlayerPbase>> N_1service
INHERITS <<SYSTEM TYPE NlayerPbase>> N_1service
[(n_1out)]
gbl2gal
gal2gbl
[(n_1out)]
1(1)
Z Suppl.1(15)_Ex10-3d

Example 10-3d – BLOCK TYPE *N_1service*

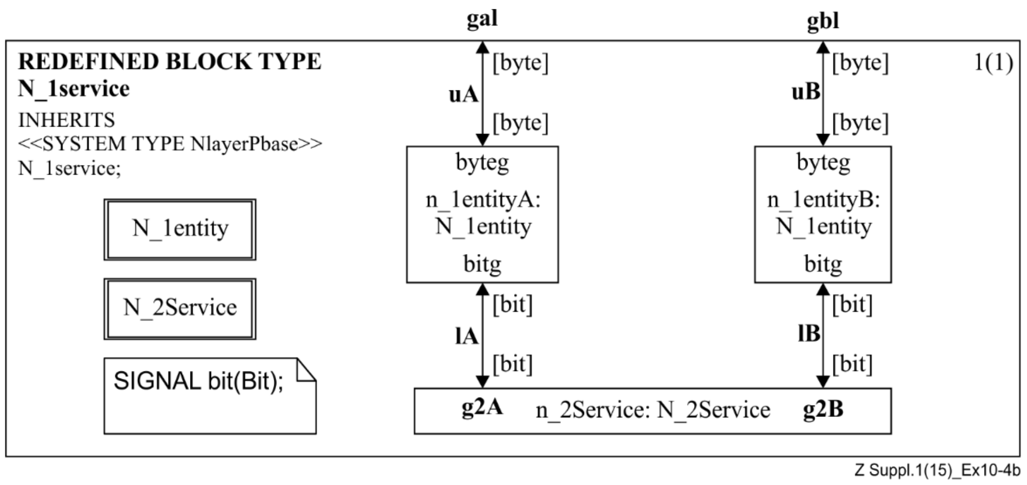
"Transport" specification

The "transport" specification for layer (N-1) is modelled by a redefinition of *N_1service* in the **SYSTEM TYPE** *NlayerT* that otherwise inherits the next higher-level **SYSTEM TYPE** *NlayerP*, is defined in **PACKAGE** *NlayerT* and is used to define **SYSTEM** *nlayerT* (see Example 10-4a).



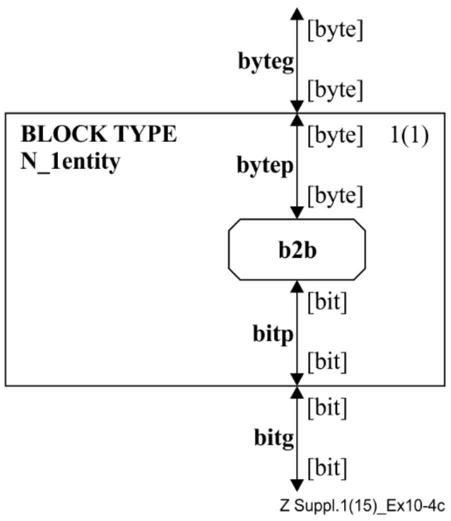
Example 10-4a – T layer

The (N-1) service is redefined and therefore replaces the *N_1service* of Example10-3d. The general structure of this **BLOCK TYPE** (shown in Example 10-4b) is similar to the *Nservice* in Example 10-3b, except in this case the (N-2) service **BLOCK TYPE** and signals are introduced here (rather than in the **SYSTEM TYPE**).

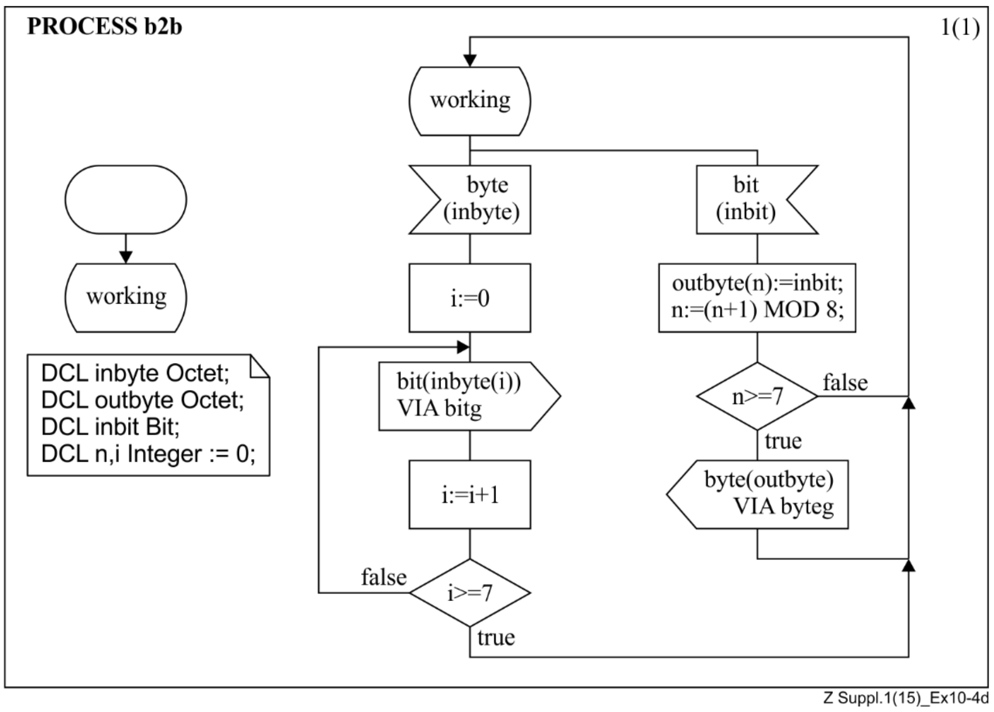


Example 10-4b – T layer BLOCK TYPE *N_1service*

For the example, the (N-1) entity is a simple conversion of byte signals to bit signals, as shown in Example 10-4c and Example 10-4d.



Example 10-4c – BLOCK TYPE *N_1entity*



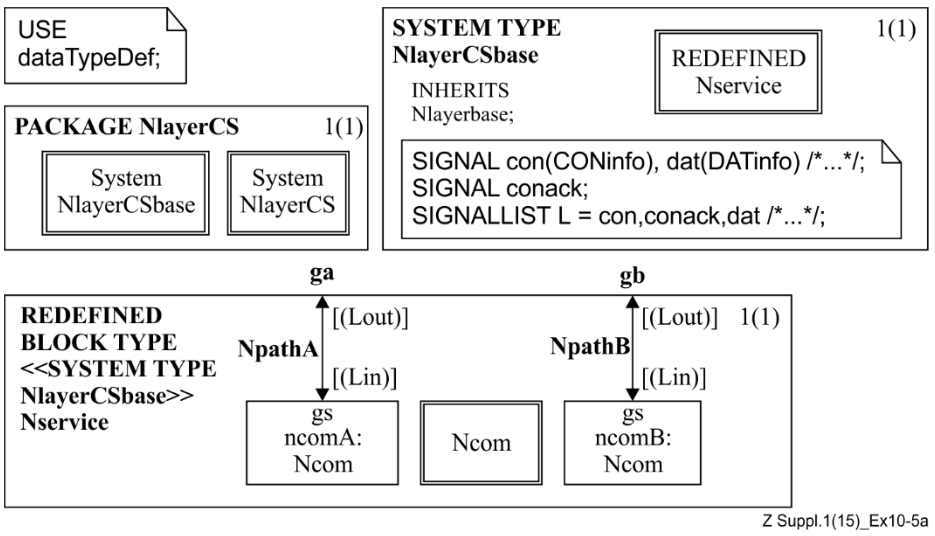
Example 10-4d – PROCESS *b2b*

The (N-2) service in **SYSTEM** *nlayert* is modelled in the same way as the (N-1) service in **SYSTEM** *nlayerp* and therefore is not shown here.

10.2.2 Channel substructuring

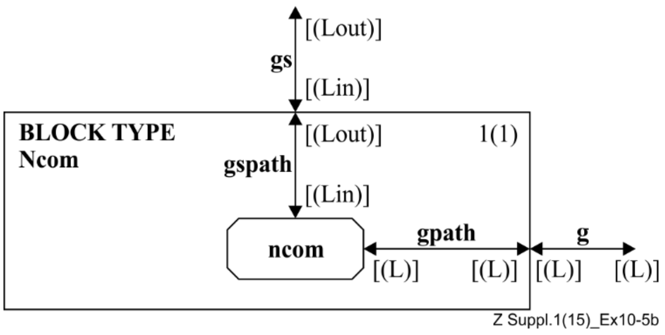
In Example 10-3b and Example 10-3c, a "virtual" channel *nchan* is indicated that conveys signals *con, conack, dat* /*…*/. If the block structure is changed, it is possible to show a real channel that conveys these signals, which is obtained from the approach in Example 10-3 by grouping the processes differently, introducing the real channel *Nchan*.

The **PACKAGE** *NlayerCS* (see Example 10-5a) contains **SYSTEM TYPE** *NlayerCSbase* and **SYSTEM TYPE** *NlayerCS* based on the **SYSTEM TYPE** *Nlayerbase* above.



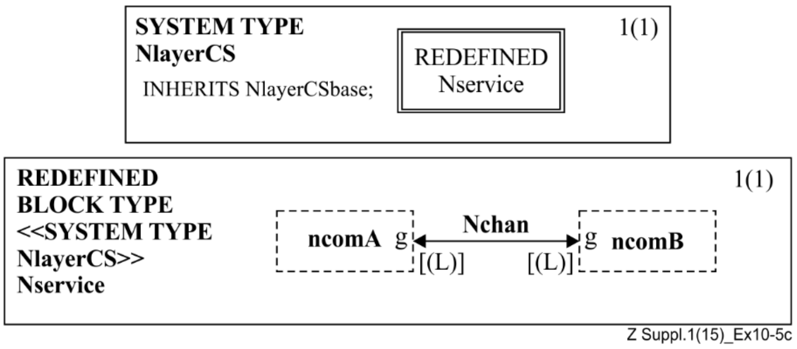
Example 10-5a – *NlayerCSbase*

The base **SYSTEM TYPE** *NlayerCSbase* defines the block type *Nservice* to contain the definition of a **BLOCK TYPE** *Ncom* and two instances of *Ncom* that will communicate. *NlayerCSbase* is incomplete because the communication between the instances of *Ncom* is not shown, but this **SYSTEM TYPE** can be inherited and *Nservice* further redefined to add the communication. **SYSTEM TYPE** *NlayerCSbase* is defined in **PACKAGE** *NlayerCS* together with **SYSTEM TYPE** *NlayerCS* that inherits *NlayerCSbase*. **BLOCK TYPE** *Ncom* (see Example 10-5b) defines the gate *gs* that is connected in *Nservice* of *NlayerCSbase* and gate *g (*connected in *Nservice* of *NlayerCS* and *N_1layerCS,* see below Example 10-5c and Example 10-5e).



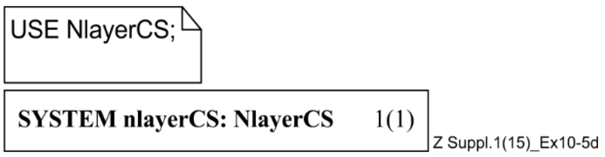
Example 10-5b – ***Ncom***

The **SYSTEM TYPE** *NlayerCS* (see Example 10-5c) redefines *Nservice* to include the real channel *Nchan*. The dashed versions of *ncomA* and *ncomB* refer to instances as defined in *Nservice* of *NlayerCSbase*, so these instances are connected to the users A and B by the communication paths *NpathA* and *NpathB*. The channel *Nchan* conveys protocol data units, indicated by the signal list *L*. This approach emphasizes the protocol view and the horizontal orientation of OSI.



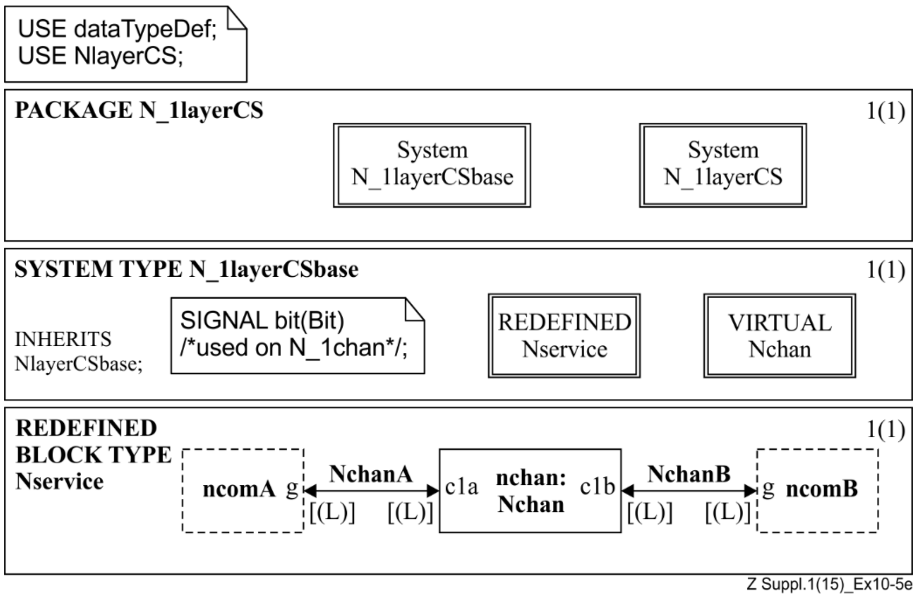
Example 10-5c – ***NlayerCS***

The **SYSTEM TYPE** *NlayerCS* can be instantiated as **SYSTEM** *nlayercs* (see Example 10-5d).



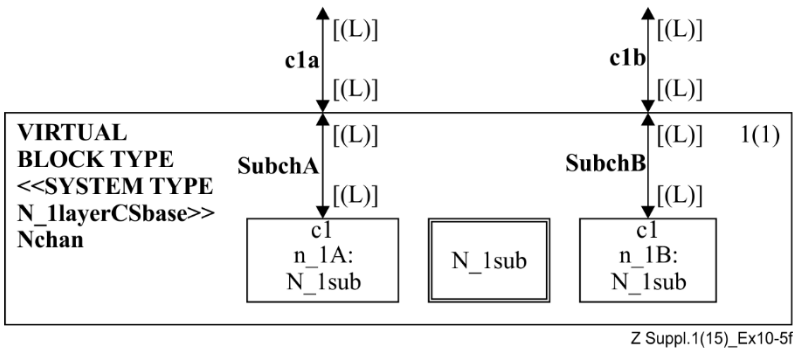
Example 10-5d – SYSTEM ***nlayerCS***

SYSTEM TYPE *N_1layerCS* in **PACKAGE** *N_1layerCSbase* redefines *Nservice* and introduces **VIRTUAL BLOCK TYPE** *Nchan* to "substructure" the channel *Nchan* (see Example 10-5e). The redefinition of *Nservice* inserts the *nchan* instance of *Nchan* between *ncomA* and *ncomB*. The definition of **BLOCK TYPE** *Ncom* is not changed because the way it behaves and its interfaces are unchanged.



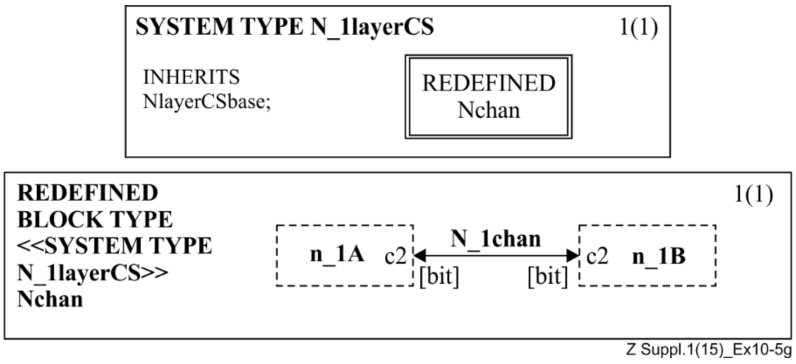
Example 10-5e – SYSTEM *n_1layerCS*

Similar to the **BLOCK TYPE** *Nservice* in **SYSTEM TYPE** *NlayerCSbase* (see Example 10-5a), **BLOCK TYPE** *Nchan* in **SYSTEM TYPE** *N_1layerCSbase* (see Example 10-5f) introduces blocks for the instances (*n_1A* and *n_1B* instances of **BLOCK TYPE** *N_1sub*) that in redefinitions of the **BLOCK TYPE** will be at the ends of the channel (or substructure for the channel) at the layer. Like **BLOCK TYPE** *Ncom* at the top layer, **BLOCK TYPE** *N_1sub* is not redefined for system instances with further channel substructuring.



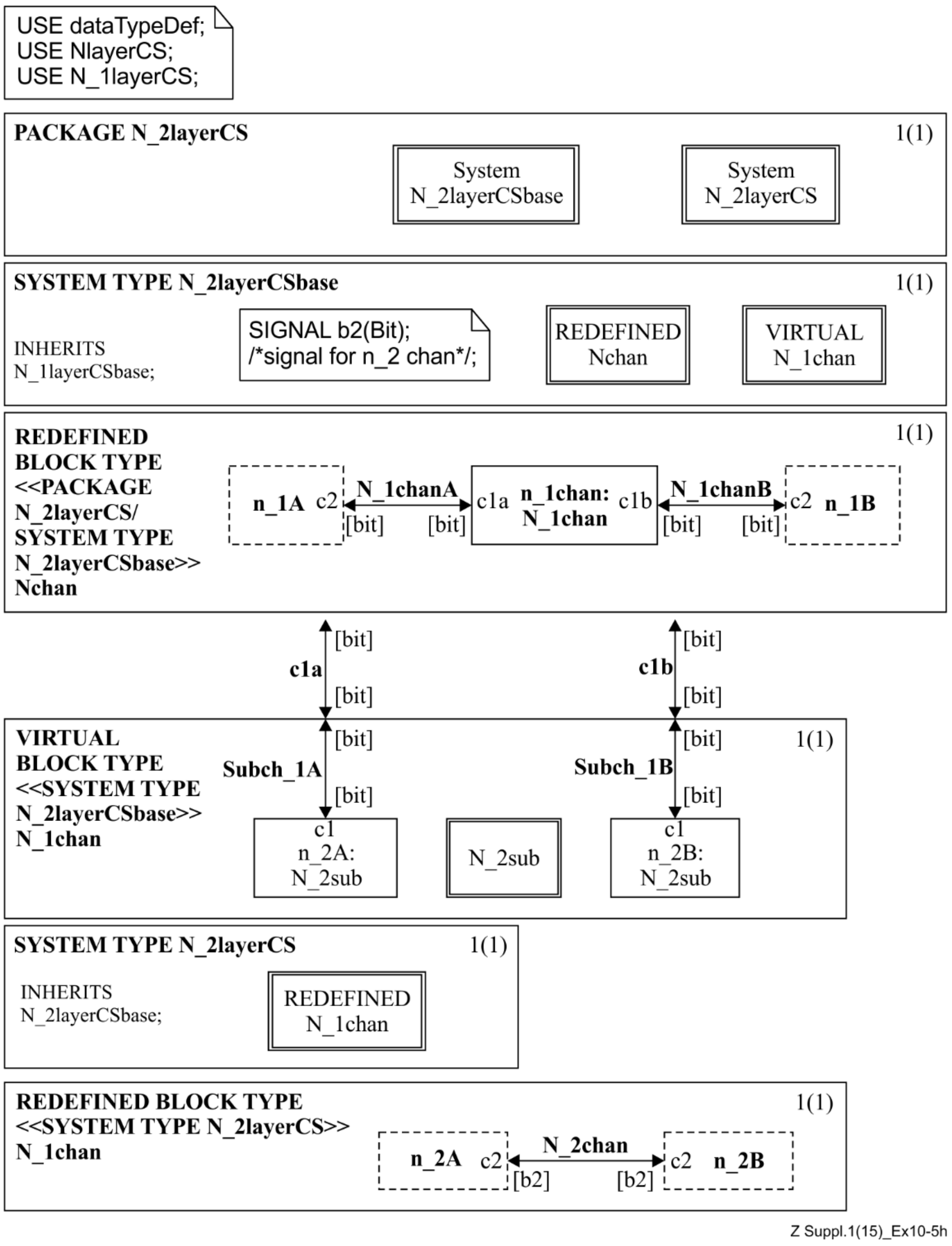
Example 10-5f – VIRTUAL BLOCK TYPE *Nchan*

In the **SYSTEM TYPE** *N_1layerCS*, *Nchan* is redefined to introduce the direct channel *N_1chan* between *n_1A* and *n_1B* (see Example 10-5g).



Example 10-5g – channel *N_1chan*

The channel *N_1chan* is substructured in a similar way. **SYSTEM TYPE** *N_2layerCSbase* inherits from **SYSTEM TYPE** *N_1layerCSbase*, **BLOCK TYPE** *Nchan* is redefined (similar to the redefinition of *Nservice* in *N_1layerCSbase*) and **BLOCK TYPE** *N_1chan* is introduced for the substructure of channel *N_1chan*. Any additional signals (signallists or interfaces) needed for (N-2) layer are defined in **SYSTEM TYPE** *N_2layerCSbase*; in this simple model *b2*. **SYSTEM TYPE** *N_2layerCS* inherits **SYSTEM TYPE** *N_2layerCSbase* and redefines **BLOCK TYPE** *N_1chan* to add a direct channel.

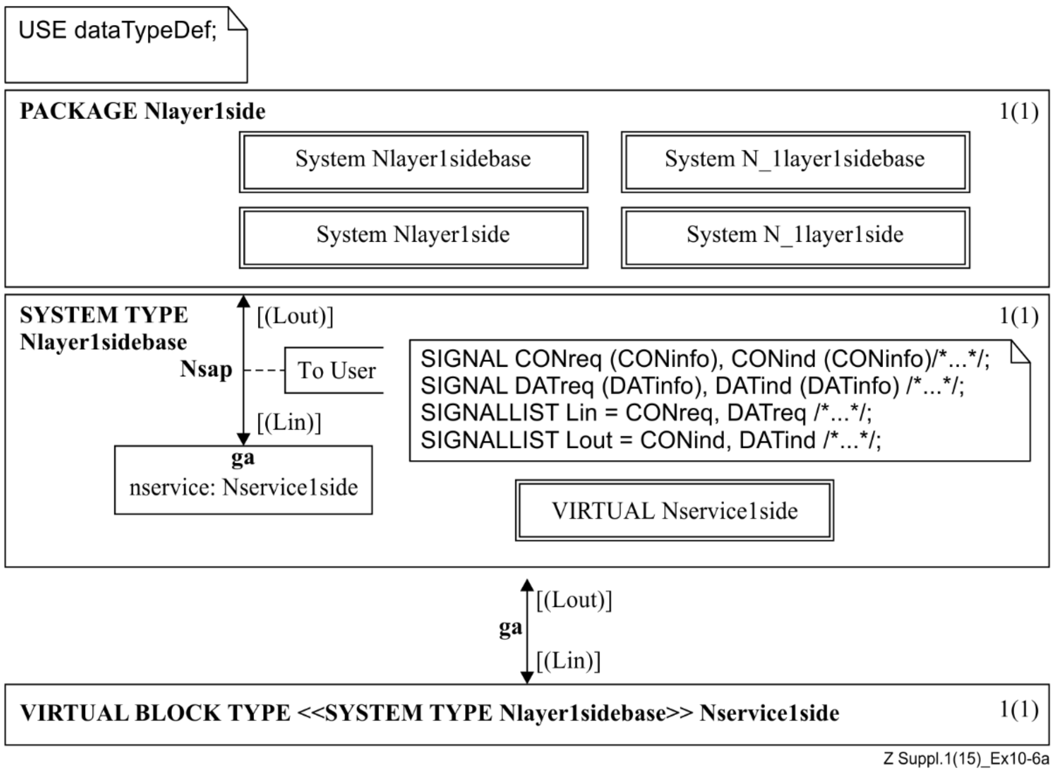


Example 10-5h – channel *N_2chan*

10.2.3 Application to 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 by instantiation of a type, like the block type *Nentity* in the block type *Nservice* in Example 10-3b, where block type *Nentity* is instantiated into *nentityA* and *nentityB*. Note that service specifications are always symmetrical; only protocol specifications can be asymmetrical.

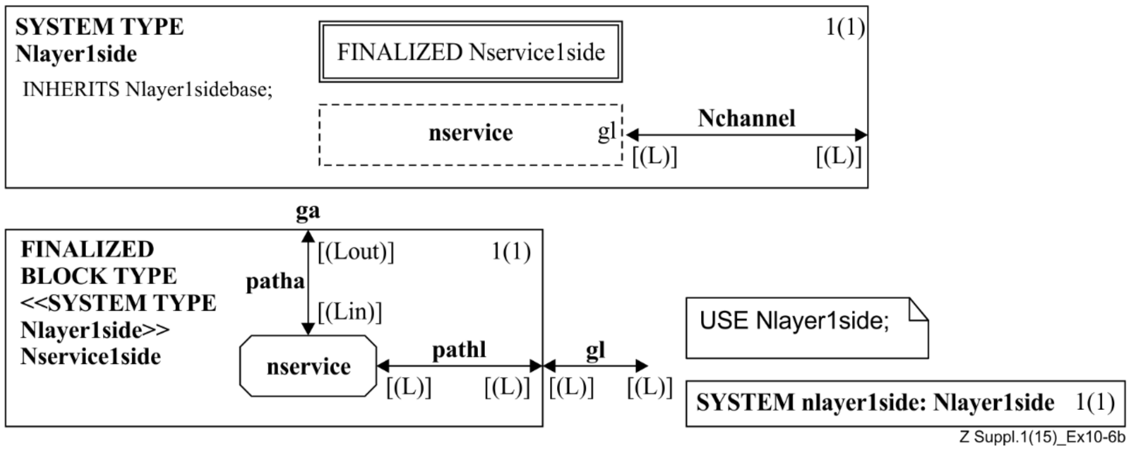
An alternative approach is to represent only one side (see Example 10-6), which is a modification of the system diagram in Example 10-1. The channel *NsapA* has been renamed *Nsap*, in the **SYSTEM TYPE** *Nlayer1sidebase*, which also defines the signals and signallists to communicate on channel *Nsap*, the signals and signallists to communicate at the top level with the user in the environment, and a virtual **BLOCK TYPE** *Nservice1side* with a defined interface with the next level up, which for this level is the environment (see Example 10-6a).



USE dataTypeDef;
PACKAGE Nlayer1side
1(1)
System Nlayer1sidebase
System N_1layer1sidebase
System Nlayer1side
System N_1layer1side
SYSTEM TYPE Nlayer1sidebase
[(Lout)]
1(1)
Nsap
To User
[(Lin)]
SIGNAL CONreq (CONinfo), CONind (CONinfo)/*...*/;
SIGNAL DATreq (DATinfo), DATind (DATinfo) /*...*/;
SIGNALLIST Lin = CONreq, DATreq /*...*/;
SIGNALLIST Lout = CONind, DATind /*...*/;
ga
nservice: Nservice1side
VIRTUAL Nservice1side
[(Lout)]
ga
[(Lin)]
VIRTUAL BLOCK TYPE <<SYSTEM TYPE Nlayer1sidebase>> Nservice1side
1(1)
Z Suppl.1(15)_Ex10-6a

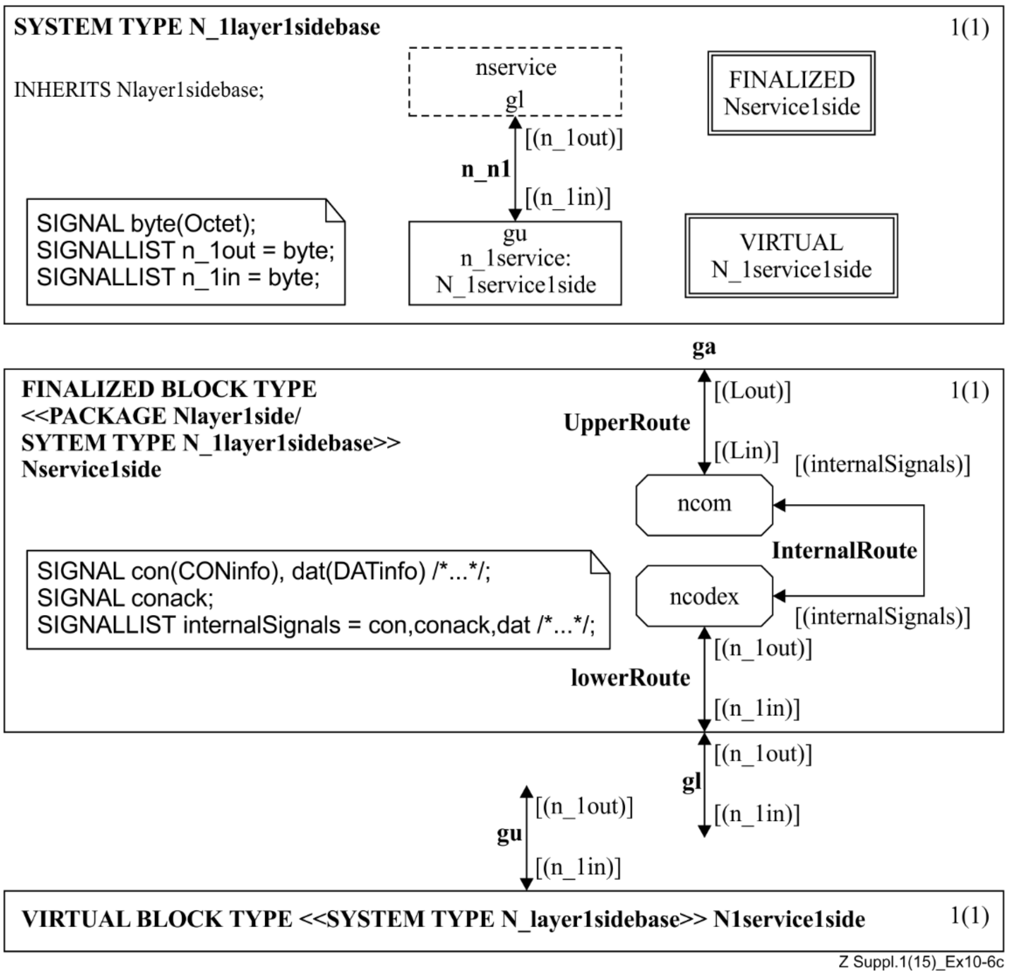
Example 10-6a – Single side model base top level

The **SYSTEM TYPE** *Nlayer1side* inherits *Nlayer1sidebase*, defines the signals for the signallist *L* to communicate with the other side in the environment at this level via with channel *Nchannel*, and finalizes **BLOCK TYPE** *Nservice1side*. The finalized *Nservice1side* determines how the instance *nlayer1side* behaves (see Example 10-6b).



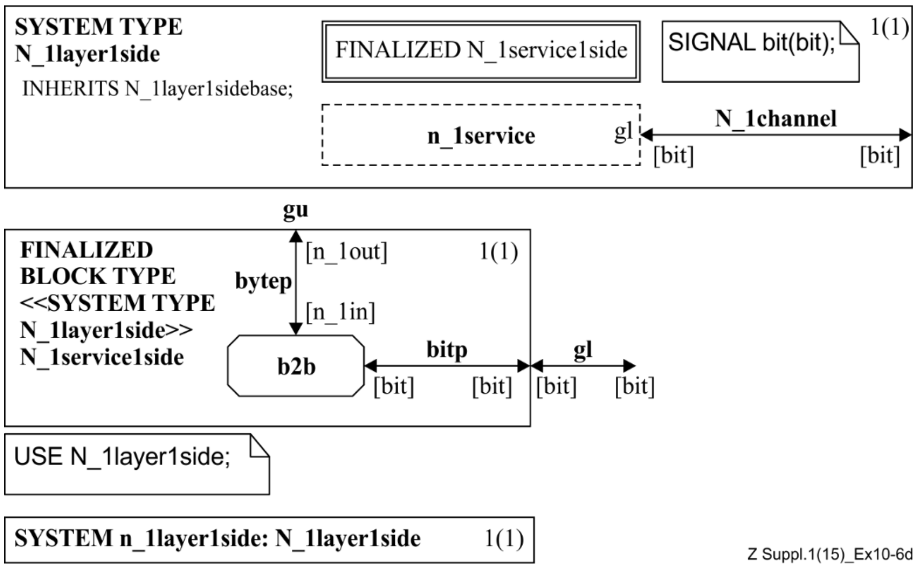
Example 10-6b – Single side model top level

A similar pattern is repeated at each level: a base **SYSTEM TYPE** that defines the interface from the next higher level and a **SYSTEM TYPE** that inherits the base one and adds the peer-to-peer communication. Level 1 and lower, the base **SYSTEM TYPE** inherits from the base **SYSTEM TYPE** of the higher level, and defines a block that connects from the higher level to the current level. This is again inherited at lower levels and does not need to be redefined again as the communication between levels and the way higher levels behave does not change. For this reason the connection block is **FINALIZED**. In **SYSTEM TYPE** *N_1layer1sidebase* the upper gate of *Nservice1side* is inherited, the contents and lower gate are defined (see Example 10-6c).



Example 10-6c – Single side model base level 1

The **SYSTEM TYPE** *N_1layer1side* inherits *N_1layer1sidebase*, defines the communication with the other side in the environment at this level via with channel *N_1channel*, and finalizes **BLOCK TYPE** *N_1service1side*. The finalized *N_1service1side* determines how the instance *n_1layer1side* behaves (see Example 10-6d).



Example 10-6d – Single side model level 1

A system *n_2layer1side* where the peer-to-peer communication is at the next level down is defined in the same way: a base **SYSTEM TYPE** *N_2layer1sidebase* inherits *N_1layer1sidebase* with a finalized **BLOCK TYPE** *N_1service1side* for *n_1service*, adding a **BLOCK** *n_2service* based on a virtual **BLOCK TYPE** *N_2service1side*, and the communication from *n_2service* to *n_1service*; then a **SYSTEM TYPE** *N_2layer1side* that inherits *N_2layer1sidebase* is defined for redefining **BLOCK TYPE** *N_2service1side* to complete *n_2service*.

10.3 Relationship with ITU-T Q.1200-series (IN) architecture and SIBs

The ITU-T Q.1200-series [b-ITU-T Q.1200] "Intelligent Network" (IN) architecture published in 1997 includes a full SDL-92 model that supports service independent building blocks (SIBs). The linking of basic call processing to the service logic is a significant modelling issue that needed to be considered for the IN architecture. The major difference between this architecture and earlier ones is the introduction of a global functional plane above the (distributed) functional plane and the use of SIBs.

SDL-92 provided mechanisms to support the IN approach, and an object-oriented approach was taken in defining the SDL-92 model, which is published in machine-readable form in [b-ITU-T Q.1228]. The SDL-92 in [b-ITU-T Q.1228] describes the operation of Capability Set 1 (CS1) and Capability Set 2 (CS2). Although this model was written using SDL-92 it is still valid in SDL-2010.

There is a system model for CS1 based on a **system type** (INAP_CS1) that has a number of its component types defined as **virtual**, for example the **block type** SSF_CCF for call control, the **block type** TCAP_Simulator, **virtual process** CallSegment and **virtual process** SSF_FSM that manages IN CS1 operations. In these processes, some of the transitions are defined as **virtual**. The reason for these **virtual** definitions is so that the items can be **redefined** in the **system type** for CS2.

ASN.1 is used to define the data communicated in various messages.

The system model for CS2 is based on a **system type** (INAP_CS2) that **inherits** INAP_CS1, so that the general structure, the data definitions and most of the operation of CS1 is reused. The CS2 model adds additional messages and operation for CS2. For example, CallSegment is **redefined** to handle the extended connection view (in terms of call legs) and the additional IN CS2 primitives. In CallSegment for CS1 the Connnect signal in every state (except Stable_1_Party) is simply forwarded to SSF. This transition is defined as **virtual** and in CallSegment for CS2 the Connnect signal is **redefined** for the Terminating_State to add a new leg to the call.

The system model for CS2 is the basis for the unified functional methodology architecture described in [b-ITU-T Q.65 2000].

10.4 Relationship with remote operations (RO and ROSE) in [b-ITU-T X.219]

In [b-ITU-T X.219] remote operations provide a way of associating a request with the possible responses. If an operation is synchronous (class 1), it can be mapped onto an SDL-2010 remote procedure. If the operation never has a response (class 5), it can be mapped onto an SDL-2010 signal. If the operation is asynchronous and provides a response of some kind (success or failure), it maps onto signals in both directions on a channel, and the operation parameter and the response signals are linked informally by comments (and also implicitly by the SDL-2010 process behaviour). Similarly, association is not mapped onto any explicit SDL-2010 feature.

Because ROSE uses ASN.1 to define the data, this can be used directly for the signal parameters.

Example

```
HoldMPTY::= OPERATION
RESULT
ERRORS{
     IllegalSS_operation,SS_errorstatus, SS_incompatibility,
     FacilityNotSupported, SystemFailure }
```

can become in SDL-2010

```
signal   HoldMPTY,
     IllegalSS_operation, SS_errorstatus, SS_incompatibility,
     FacilityNotSupported, SystemFailure;
```

with these signals used on the channel to and from the process containing the operation.

11 Justification of approach

The methodology presented in this Supplement is the view of the experts contributing to ITU-T on the use of SDL+ in the period 1988 to 1995. It was based on several years of research and experience. Some of the material on which the Supplement has been based can be found in [b-Olsen], [b-Reed] and [b-TIMe]. There have been a number of changes in the kinds of systems that are produced since the methodology was first published in 1997, but the fundamentals of defining a model in an extended finite state machine language are unchanged. The main advances in the ITU-T languages have been the evolution of TTCN to TTCN-3, and the introduction of the User Requirements Notation. Neither of these has replaced the ITU-T Specification and Description Language.

The methodology is specifically focused on the generation of a precise specification of a system. This limitation is chosen for four reasons:

1) There is general agreement on the approach for formalizing requirements into SDL-2010.
2) There are many different ways of implementing a product from an SDL-2010 specification.
3) Time to market is often more important than execution efficiency and SDL-2010 tools are capable of producing reasonable code from implementation descriptions that are close to the product specifications.
4) The methodology can be used both for producing specifications for products and for specifications for use within standards (or for procurement).

The methodology is defined in terms of SDL-GR. Most users prefer to use the SDL-GR form and find it easier to understand. The methodology can be adapted for SDL/PR use.

The methodology is presented as a number of activities and steps to be followed, because a number of users asked for this approach and the steps presented in Appendix I of [b-ITU-T Z.100a] were received favourably by users. These steps have been elaborated and extended in Part II of this Supplement. The activities are defined in Part I and elaborated for a particular case in Part II. Part I therefore defines a general framework and Part II defines one set of steps that can be used.

The methodology does not use every feature of SDL-2010 and in some cases suggests that features should not be used. It should not be implied that there is never a good case for using these features. The methodology is general and deviation for a particular application or organization is expected.

PART II – AN ELABORATION OF THE FRAMEWORK METHODOLOGY

12 Elaboration of the methodology for service specification

The activities are described using a number of *steps* that are sometimes further divided into *instructions* and *guidelines*. Throughout the steps a number of *rules* are stated. The rules are either expressed with the verb "shall" or the verb "should". The word "shall" is used for a rule that needs to be followed to ensure a valid well-formed system that is understandable, implementable and testable. The word "should" is used for rules that may be broken under some circumstances. The distinction of a *guideline* from a *rule* is that a guideline gives advice that may be ignored without serious consequences, or indicates choices that can be taken.

The terms *step*, *instruction*, *guideline* and *rule* are defined in clause 2.

12.1 Three-stage methodology: Stage 2 (Recommendation ITU-T Q.65)

The relationship between the three-stage methodology of [b-ITU-T I.130] plus [b-ITU-T Q.65 1988] and/or [b-ITU-T Q.65 2000] and the methodology in this Supplement is described in clause 10.1. Here the mapping of ITU-T Q.65 onto SDL-2010 constructs is explained.

Ideally, it should be possible to use structural concepts of SDL-2010 in ITU-T Q.65 step 1, and behavioural concepts in ITU-T Q.65 steps 2-4 with stepwise introduction of data. The general argument for using SDL-2010 in steps 1-4 of Recommendation ITU-T 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 ITU-T Q.65 can be mapped onto SDL-2010.

Structure

Step 1 in [b-ITU-T Q.65 1988] is: Functional model, identification of functional entities and their relationships matches the use of structural concepts in SDL-2010. Figure 12-1 shows an example of a functional model according to [b-ITU-T Q.65. 1988].

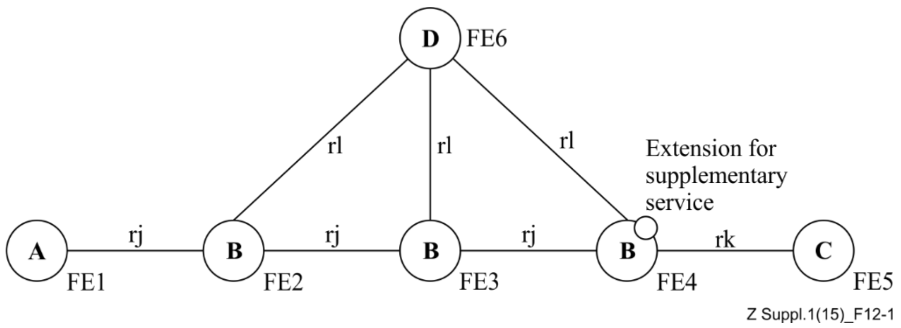


Figure 12-1 – Example of a functional model [b-ITU-T Q.65 1988]

The elements of the figure are:

– names within circles (for example *A*): types of functional entities;

– names in upper case adjacent to circles (for example *FE1*): names of functional entities (instances);

– names in lower case between circles (for example *rj*): relationships between types of functional entities;

– added shadowing circle: extension for supplementary service.

Note that this model clearly distinguishes between types of functional entities (for example *A*) and their instantiation (in this case *FE1*) when describing the structure of the system. [b-ITU-T Q.65 1988] 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 **type** features of SDL-2010.

Investigation has shown that the "Functional entities" can be mapped onto SDL-2010 structural concepts as follows:

– model → system

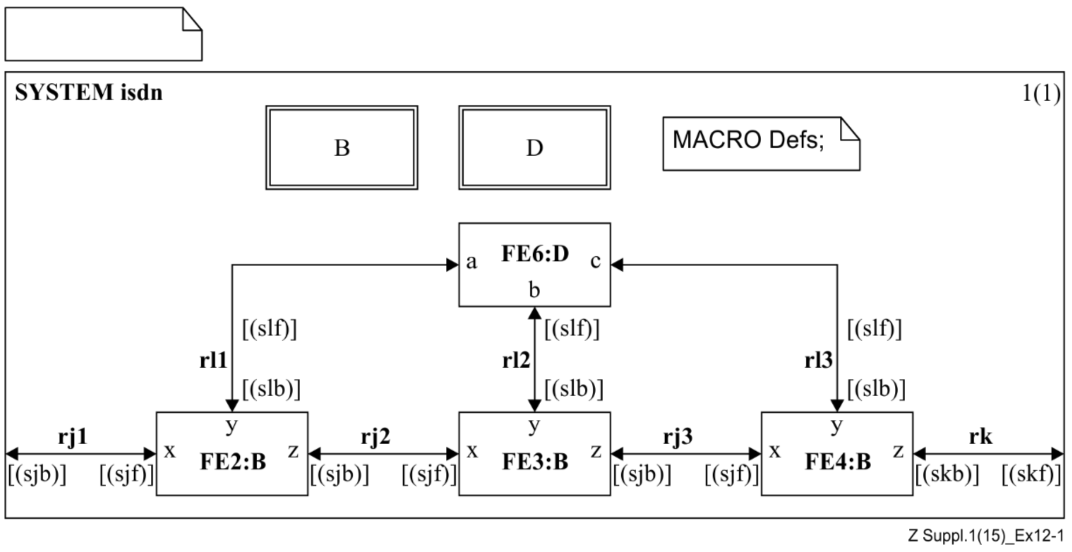
– functional entity → agent with a state machine (usually a process)

– relationship between functional entities → channel.

In addition, by using **type** constructs:

– type of functional entity → agent (process or block) type.

A difference between the functional model according to [b-ITU-T Q.65 1988] and SDL-2010 structure diagrams is that SDL-2010 structure diagrams normally model open systems.



Example 12-1

Example 12-1 shows the functional model of Figure 12-1 in SDL-2010. 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-2010 structure diagrams. *B* and *D* are block types.

The advantages of using SDL-2010 diagrams for structure are mainly that SDL-2010 diagrams contain definitions of signals, data types etc. which are important for subsequent steps in [b-ITU-T Q.65. 1988]. In Example 12-1, these definitions are indicated by the **macro** *Defs*.

The unified functional model architecture in [b-ITU-T Q.65 2000], Figure 3 is a composite model based on basic call (Q.71), IN CS-2, IMT-2000, and B-ISDN and each functional entity has (such as Call Control Agent – CCA) has a specific description of the service it provides.

Behaviour

The behaviour is described in steps 2-4 of [b-ITU-T Q.65. 1988]. The interaction between functional entities is described in information flow diagrams. Based on these, an SDL-2010 agent diagram with a state machine (i.e., states and transitions - usually for process) is produced for each functional entity type.

The information flow diagrams correspond to Message Sequence Chart diagrams [b-ITU-T Z.120]. The information flow diagrams are produced for "normal" cases, and SDL-2010 is then used to describe how the functional entities behave. This occurs in the Draft Design and Formalization activities in the methodology in this Supplement.

Step 4 of [b-ITU-T Q.65 1988] 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 elements of an information flow diagram (for example, Figure 12-2) are:

– names on top of columns (for example, *FE1*): functional entities;
– names in lower case between circles (for example, *rj*): relationships between types of functional entities;
– names on arrows between columns (*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 (for example, *Action B*, *100*): names of functional entity actions;
– numbers within parenthesis [for example, *(5)*]: labels to the SDL-2010 diagrams.

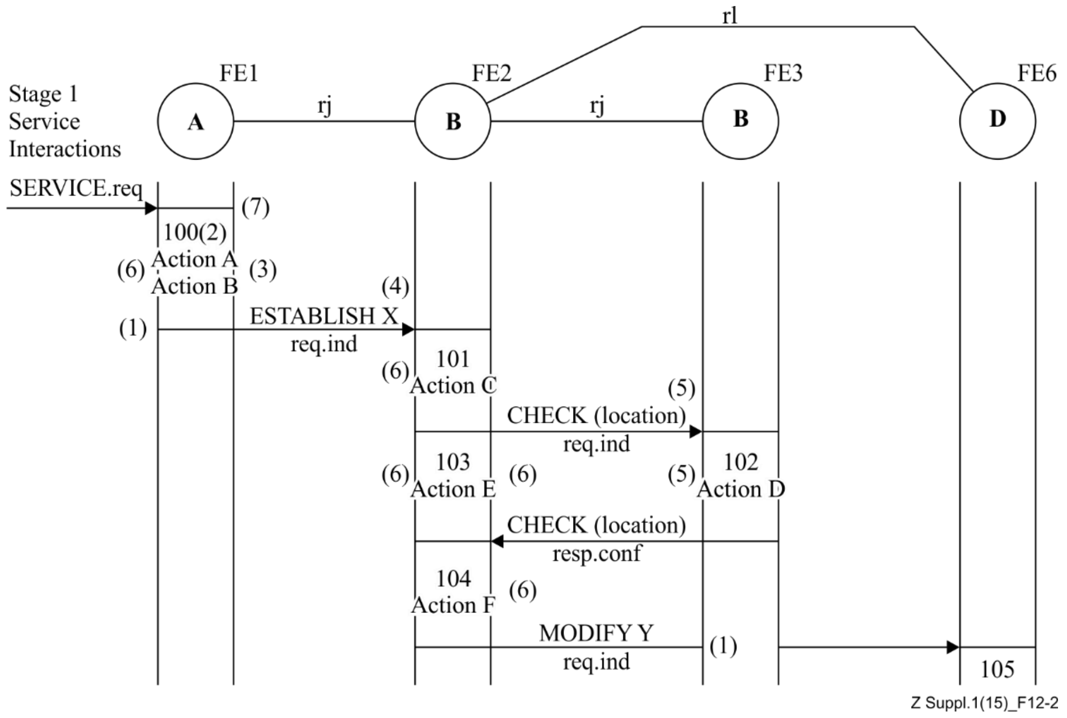


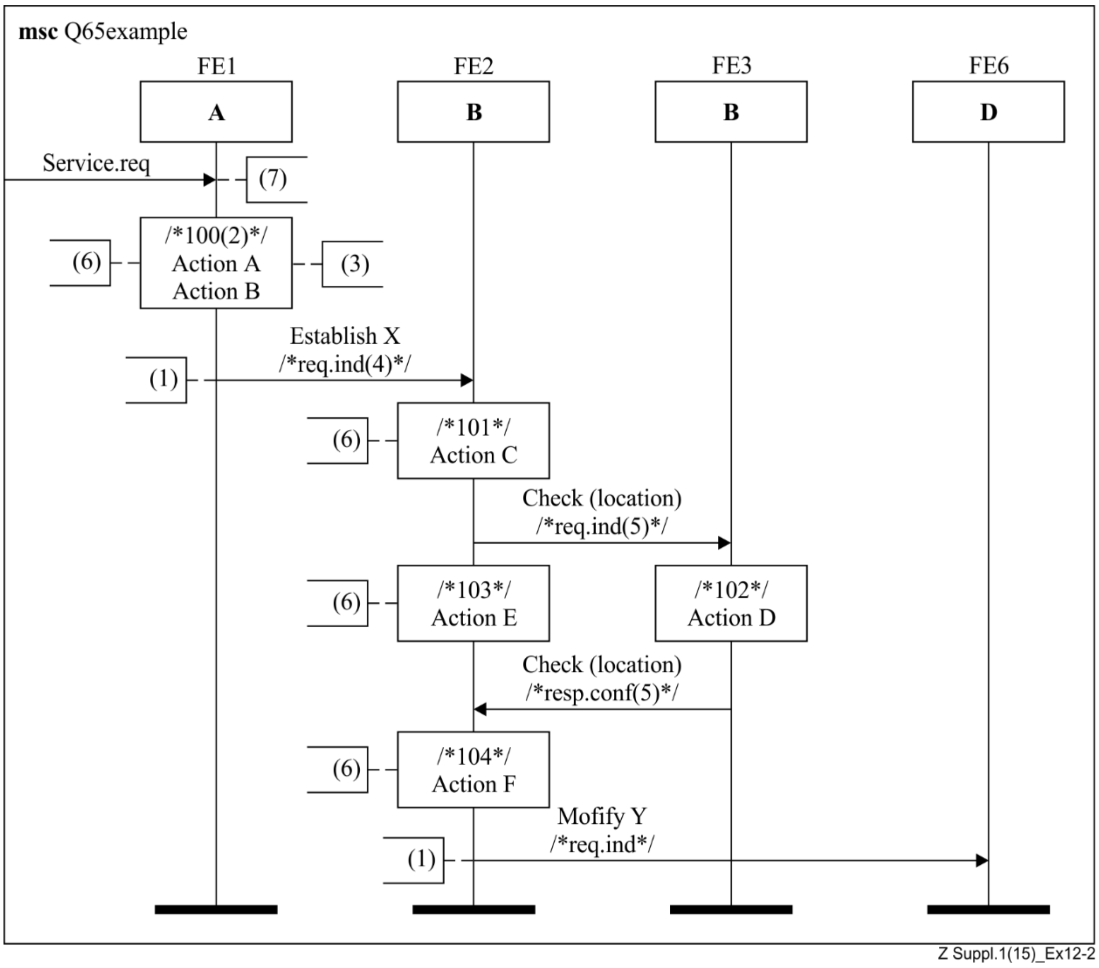
Figure 12-2 – Example of information flow diagram from [b-ITU-T Q.65 1988]

Example 12-2 shows the Message Sequence Chart diagram that corresponds to the information flow diagram in Figure 12-2. The mapping between the two is:

– functional entity name → instance (one process instance per block)
– information flow → message
– additional information → message parameters
– basic functional entity action → action

– labels to the SDL-2010 diagrams → comments.

The information flow diagrams recommended in [b-ITU-T Q.65 1988] thus basically contain the same information as the Message Sequence Chart diagrams. The SDL-2010 specifications can in general be checked against the Message Sequence Chart diagrams 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. Message Sequence Chart diagrams can be derived from SDL-2010 specifications, but normally the Message Sequence Chart diagrams are only intended to show a subset of the behaviour ("the normal behaviour"), so some human interaction is needed to derive the appropriate Message Sequence Chart diagrams from the SDL-2010 specification. Another possibility is to derive automatically skeletons of SDL-2010 agent diagrams with state and transitions from the information flow diagrams. See clause 15.2.2, **Step B:2 – Skeleton processes**.



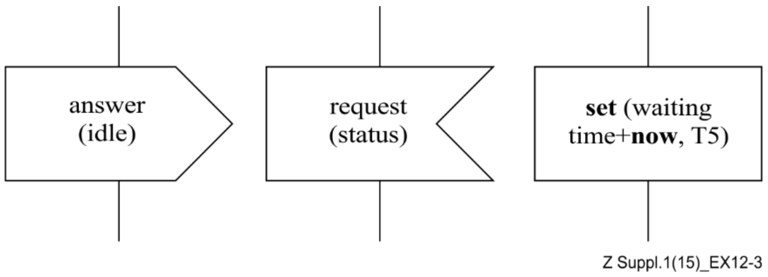
Example 12-2 – Message Sequence Chart version of the example in Figure 12-2

In clause 2.5 of [b-ITU-T Q.65 2000], step 5 is concerned with the production of Specification and Description Language diagrams for the functional entities. Some examples are given.

Data

Tasks and **decisions** were mainly informal in the existing Specification and Description Language diagrams created for ISDN Recommendations, and there was little need for defining operators for data types. Data items mainly appeared in **input**, **output** and **timer** constructs (see Example 12-3).

In most cases, *Boolean* and *Integer* are sufficient, for example, 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.



Example 12-3 – Use of data

However, more recent Recommendations have used data more formally as outlined in [b-ITU-T Q.65 2000] and used in [b-ITU-T Q.1228].

The definition of data as enumerated types is useful. A data type with only literals matches an enumerated data type, for example (in SDL/PR):

```
value type Indication { literals idle, busy, congestion; }
...
signal Check_location (Indication) /*resp.conf */;
...
output Check_location (busy) /*resp.conf*/;
...
```

NOTE – The legacy **newtype** notation for the data type is: `newtype Indication literals idle, busy, congestion; endnewtype Indication; …`

Data items are introduced in steps 1-4 when formalizing the specifications; for example, signal specifications are formalized by adding 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-2010 specification.

13 Analysis steps

The Analysis activity consists of two steps: Inspection and Classification. The application domain need not be totally inspected before starting the classification step. Application concepts or requirements that have been inspected can be classified, while other concepts or requirements are still in inspection.

The Inspection and Classification activities produce two views of the application:

– a logical view to describe key elements of the system to specify

– a dynamic view to describe the behaviour of all the objects.

The logical view can be carried out by component-relationship modelling (called *object modelling*). This corresponds to an information view of the system. In order to ease reuse – to include external domain components, as well as to create new reusable domain components, object-oriented analysis (OOA) is applied. Reuse is greatly eased by the concept of encapsulation of data and services and by the concept of inheritance. These two concepts allow the reuse of more general components or components with a behaviour close to the expected one without modifying them. The specialization of general components or the adaptation of closely related components is performed by introducing new appropriate components. As a consequence, OOA techniques are very useful to create and reuse domain components (i.e., components appropriate to a domain). The Analysis steps are explained in detail by using a subset of UML, but another approach supporting OOA could be used.

The Message Sequence Chart notation [b-ITU-T Z.120] is perfectly suited to model a dynamic (or behaviour) view that corresponds to an initial computational view of the system. This task consists of defining use sequences expected by the system user. These use sequences will be generally composed of: the typical scenarios that form the usual behaviour of the system, and the exceptional scenarios that specify the response of the system to faulty input, failures, etc. Other use sequences can be added to specify particular aspects of the behaviour, for example when the system starts or when the system stops. This modelling is similar to the use case modelling in [b-OOSE].

This clause also takes care of detailing some consistency rules to be fulfilled when the object modelling and the use sequence modelling are performed in parallel.

The Analysis activity shall be performed without making assumptions about the application architecture or implementation details. In particular, data shall be modelled according to the requirements from the user point of view only and not from the point of view of a systems engineer who seeks to optimize the system implementation. In this respect, an enterprise model that clearly identifies who the users are and what roles they play is useful.

13.1 Inspection step

The different tasks constituting the inspection step are presented in clause 5.4.1. These tasks are not specifically supported by system engineering techniques. Engineers can preferably use a word processor to build the bibliography and to write reader's notes.

13.2 Classification step for object modelling

The object modelling is concerned with two categories of object:

– Physical objects: models of physical entities that compose the environment of the system or are used by the system to supply the expected services.

– Logical objects: models of the information consumed, produced or stored by the system, or models of application concepts.

Objects that model entities of the environment (physical as well as logical) are called *external agents*. Many of the objects in telecommunications (such as calls, routes, subscriber account) are logical rather than physical. This is particularly true for a specification system in a standard.

Associations are dependent relationships between objects such as:

– physical connections between physical objects

– producer/consumer relations between objects (physical or logical)

– use relations between objects (physical or logical)

– *aggregation* links between objects (physical or logical)

– inheritance links that allow reuse of more general or similar objects.

An *aggregation* is an association between a composite object and the component objects: sometimes called a refinement.

The modelling is neither concerned with how/when information is generated/ passed from place to place, nor concerned with how/when physical entities interact. It is a static (or structural) model.

So that the model is general and can apply to different instances of the application, class models are produced. A class of an X (for example, a telephone) defines the characteristics that hold for all X objects (that is, all telephones in the example). The class model shows the associations (relationships) between classes. In an actual application, the objects (instances of the classes) shall conform to the class diagram. For example, if the "exchange" class is "wired to three or more" (an association) of the "telephone" class, there will always be at least three telephones for a real exchange. Although it is possible to draw object instance models, this is not usually done.

The notation used is the class diagram notation of [b-OMG UML], the basic elements of which are shown in Figure 13-1. Example 13-1 shows a partial diagram where the application domain relates to banking operations with cash cards. The class diagram is an "object model": it defines how object instances of the classes are related.

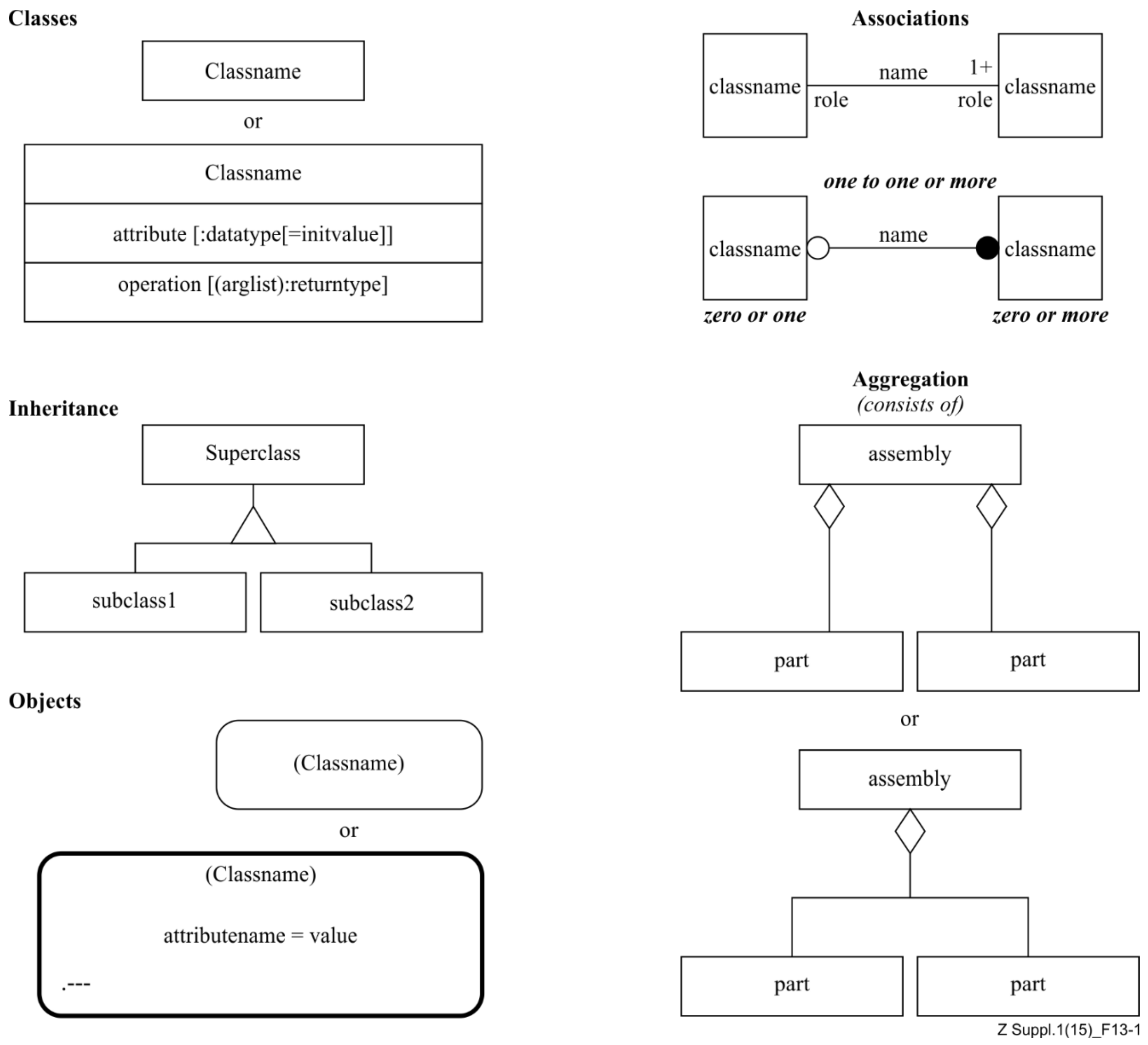
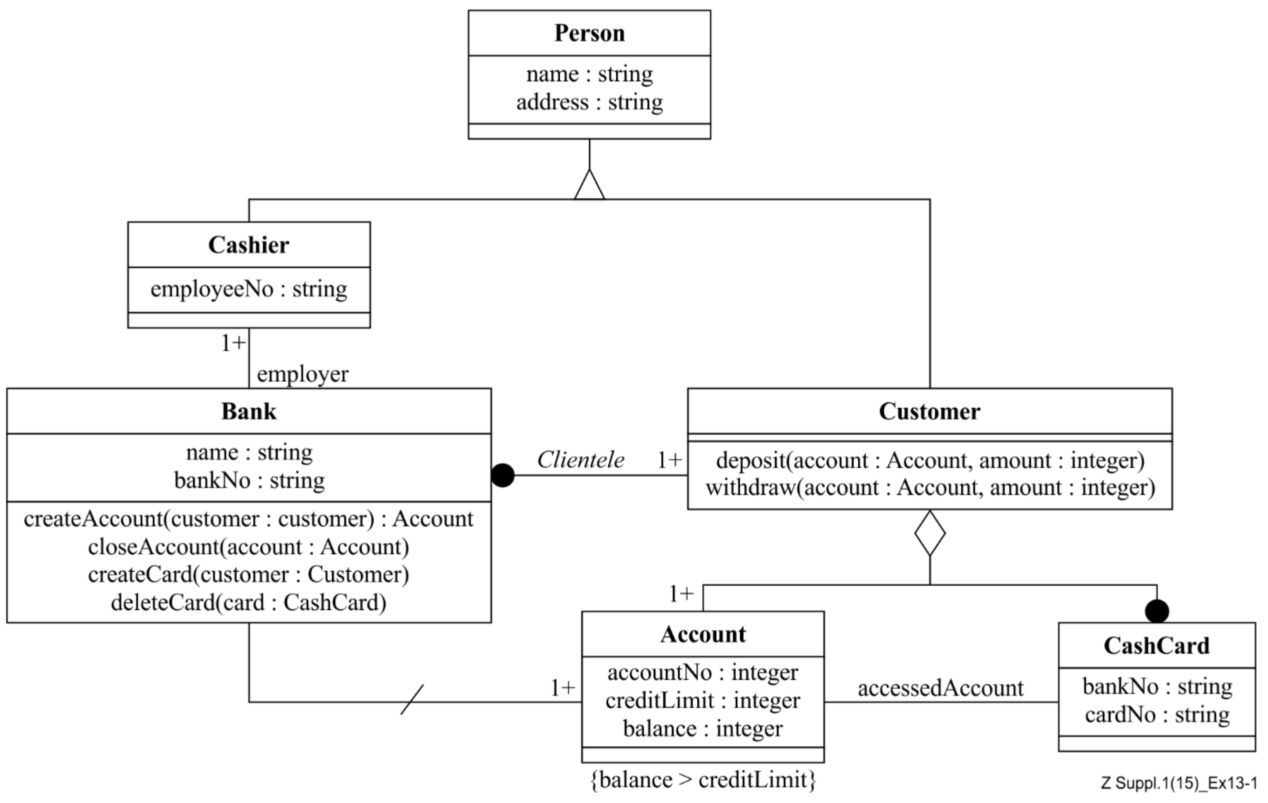


Figure 13-1 – Basic elements of UML class diagrams



Example 13-1 – Example of an UML class diagram

- Classes (e.g., Bank): They contain attributes (e.g., name and bank number of Bank) and operations (e.g., createAccount and closeAccount of Bank); attributes are optionally typed and the signatures of the operations can be declared.
- Associations (e.g., Clientele between Bank and Customer): An association has no privileged direction; an association can be named either with one name or by naming its two roles (roles of the two classes linked by this association); multiplicities are indicated for each role of the association.
- Aggregations (e.g., between Customer and Account and between Customer and CashCard): An aggregation is a special association modelling refinement link; it is the means to describe hierarchical structure.
- Inheritance links (e.g., between Person and Cashier and between Person and Customer): It is the support for class properties reuse.

The information aspects of a class are contained in the attributes of the class, the multiplicity of the associations it has with other classes and the aggregation associations involving the class.

The interface aspects of a class are the associations it has with other classes that are not aggregation associations, together with the signatures of the operations of the class that can be invoked by other classes.

In general, behaviour aspects are not detailed by the class model produced during Analysis and they remain as text in the collected requirements possibly linked (in some way) to the class descriptions, where appropriate. Miscellaneous aspects of a class (General: flexibility, compatibility, usability, reliability, safety, security; Design constraints; Failure modes; Performance; and so on) are also left as text. However, as far as possible, the texts for behaviour and miscellaneous aspects should be linked to the classes produced, so that it is possible to determine what aspect is handled by what object.

In order to manage the complexity of a class diagram and to preserve its readability, a diagram is structured into modules. Modules usually represent different views on the system, such as the user point of view, the physical environment point of view, and the point of view of a service to be provided by the system. The collection of all these modules constitutes the complete class diagram. Classes can be, and often are, represented in several modules. For example, in the cash card operations domain, significant modules are: the description of a customer from the bank point of view (account, cash card, etc.), the network of automated teller machines (bank consortium point of view), the composition of a teller machine in terms of equipment, and a module describing what physical entities and concepts are involved when a banking transaction is performed.

Instructions

1) Identify and name the classes, and create them in the model.
2) Identify and name associations and aggregations between classes.
3) Identify and name the class attributes.
4) Identify and name the class operations.
5) Organize and simplify classes using inheritance.
6) Group classes into modules.
7) Verify that access paths exist for usual queries and that all the required data exists.
8) Iterate and refine the model.

Guidelines

Identifying classes

The analysis of the application requirements leads to identifying physical and logical objects. Generally, classes are initially identified by examining the nouns used in the textual documents collecting the requirements. It can be useful to create a data dictionary that lists all the classes together with a description copied or derived from the captured requirements. Alternatively hypertext links to the collected requirements might be used. It will be important to determine the boundary between classes inside the system and classes outside the system. This distinction can be reflected in the names given to agents outside the system. A name should be given to the system.

Naming classes (associations, attributes and operations)

The choice of names is an important decision: often there are several terms in the collected requirements for a class of objects (or an association, attributes or operation) and a bad choice of name can lead to confusion later. However, the names that are chosen should be reviewed later and possibly amended, because there will be better understanding after some engineering has been done.

Keeping the right classes

Classes should only be derived from the collected requirements and should not introduce architectural details that are not part of the collected requirements. That is the only architectural constraint to be considered concerning the conditions under which the system must run. As a consequence, classes representing physical entities should be limited to the entities that concern the application domain and not the implementation of the application. Logical entities should represent commonly accepted application concepts and the use of a library of concepts as well as general experience will help decide if a class is appropriate. For example, a database constituted by a list of customers with their name, their address and the list of the products they have purchased can be modelled by the class "Customer" with the two attributes "name" and "address" and the class "product", without creating a third class representing the database. Redundant classes (which represent the same concept, the same physical entity or which contain the same attributes) need to be removed. Empty classes (that is, classes without attributes and loosely coupled with the other classes of the model) need to be likewise removed.

Identifying associations

Any dependency between classes is modelled as an association. An association can represent a physical link, a producer/consumer relation or a use relation (aggregations are detailed later). Associations are initially identified by examining the verbs (they "link" nouns together, such as "employs" or "accesses"). Most associations are binary, that is, they only involve two classes (a notation for multiple association exists – if needed). Do not be too concerned about naming associations, because there can be many associations that correspond to ownership ("has", "owns", ...) and connections ("joined to", "links to", "addresses" ...) that do not need names – the association is obvious from the class model.

Identifying aggregations

Associations that represent an ownership link or a composition link are usually better modelled as aggregations. An aggregation adds a new constraint compared to a simple association: the behaviour of the aggregating object has a strong influence on the behaviour of its aggregated objects; in particular, the creation and the deletion of the aggregating object generally leads to the creation and to the deletion of its aggregated objects. Aggregations are created by transforming associations that are named "part of", "composed by", "element of", etc.

Keeping the right associations and aggregations

Redundant associations must be removed or explicitly marked as redundant (the "derived" association notation – see the association between "Account" and "Bank" in Figure 13-2). Interactions between classes are modelled as associations only if they represent structural properties of the application domain and not a transient event. The multiplicity (number involved in each side) of associations should be properly defined.

Identifying the right attributes

Attributes are properties of classes. They are used to capture the data defined in the collected requirements, such as name, address, card number, etc. Design or implementation details such as process Pid must be eliminated. Complex attributes (structured data, data with unlimited size, etc.) can be transformed into (data) classes. Attributes can be typed, but sophisticated type description should be avoided if it corresponds to implementation details. Attributes cannot be pointers to other classes: associations must be used instead.

Identifying the right operations

Classes are enriched by adding operations. Operations are not easily identified when performing the object modelling only. Modelling the dynamic view of the requirements greatly helps to identify the operations the classes must provide. As a consequence, engineers must model the dynamic view in parallel with the logical view. Operations correspond to services (including access or generation of data) that the class provides to the environment or to classes with which it is connected through associations. Signatures of operations can also be defined (formal parameters and return value).

Using inheritance

Classes that have partially common structures (attributes or associations) could be reorganized by introducing a new class that encapsulates this common substructure. The initial classes then become specialized sub-classes of the super-class. Introducing super-classes can also come from the reuse of already defined classes available in the application domain. However, having many levels of inheritance should be avoided.

Grouping classes into modules

Modules are created to represent significant points of view in the application domain. Classes are grouped into modules. They can be present in more than one module and it is generally the case because the points of view are not a partition of the system. To improve the readability of the diagrams, the engineer should not create more than twelve classes in one module. Depending on the complexity of each class, number of attributes, number of operations, number of associations, the ideal number of classes for a module is between 4 and 8.

Verifying access paths and data coverage

The object model can be checked against the requirements. Two items must be checked: the ability of navigating through the model to access the information, and the data coverage. If a required navigation query is not supported even indirectly through a set of associations, new associations must be introduced. The dynamic model is very helpful to check the completeness of access paths because it shows which entities interact with each other and which information is accessed by which entity (information transmitted by messages). If a required piece of data is not present in the model either as an attribute, a set of attributes or as a result of operation, new attributes or operations must be introduced.

Refining the model

Object modelling is an iterative process. The initial class diagram is very close to the collected requirements (nouns as classes, verbs as associations, etc.). During successive iterations, the model is reorganized in order to be non-ambiguous, non-redundant, easily understandable and completely compared to the collected requirements. In particular, the identification of attributes and operations for each class will be refined in later iterations. An engineering judgement will have to be made on how complete the model should be. The dynamic modelling leads to the enrichment of the object model, especially with respect to the class operations and associations. Therefore, the iterations on the object model should be performed in parallel with the iterations on the dynamic model. Such a refinement may be considered more appropriate to the Draft Design activity, but since Draft Design is optional it can be considered as part of Classification.

Rule 1 – Plural nouns shall not be used to name classes (there is one class and several instances).

Rule 2 – The object model shall contain all the domain-relevant agents of the system environment, by means of classes.

Rule 3 – The object model shall contain all the data expressed in the collected requirements, by means of attributes or operations.

Rule 4 – The object model shall fulfil all the requirements in terms of navigation through objects, by means of associations and aggregations.

Result: A coherent and complete object model capturing in a diagrammatic way the logical view of the collected requirements.

13.3 Classification step for use sequence modelling

The objectives of the use sequence modelling is to capture and classify the expected behaviour of the system in response to stimuli coming from its environment, without assumptions on how the stimuli are considered inside the system and how the responses are elaborated. The behaviour is classified by means of a set of Message Sequence Chart diagrams usually constituted by the typical use sequences (in fact the mission of the system) and by exceptional use sequences that describe the expected robustness. Other sequences can be added to clarify particular states of the system (start, stop, etc.).

Because use sequence modelling aims in Analysis to classify the expected behaviour of the system, there is no need to produce state transition diagrams. These diagrams are created by Draft Design or Formalization steps.

Use sequences described at this step ignore the internal system architecture. The only involved elements are: the system represented as one instance and all the external agents involved in the communication with the system. These external agents correspond to dynamic entities such as a user or device interacting with the system. On the other hand, when the system architecture is sketched in Draft Design, the use sequences produced in Analysis can be refined according to it to produce use sequences that reflect internal components of the system.

One of the major encountered difficulties is to successfully manage the large number of Message Sequence Chart diagrams to be created. The set of produced Message Sequence Chart diagrams cover all scenarios in the collected requirements, but redundancy should be avoided in order to have a minimal set. For this purpose, engineers should pay special attention to the structure of the Message Sequence Chart document: the document that collects together all the Message Sequence Chart diagrams.

Instructions

1) Identify the nominal and exceptional use sequences to be created.
2) Decompose complex use sequences into smaller ones and indicate how the lower-level use sequences are grouped together by High-level Message Sequence Chart diagrams (HMSCs).
3) For each low-level use sequence, draw the Message Sequence Chart diagram with the system and all the relevant agents in this sequence as instances (vertical bars).
4) Describe the expected chronology (from the requirements) of control and data flows as messages in the Message Sequence Chart diagrams.
5) Verify that all the typical and exceptional use sequences collected in the requirements are covered by the Message Sequence Chart document.
6) Verify the consistency of the dynamic model, as described by the Message Sequence Chart document, with the object model.

Guidelines

Identifying the required use sequences

The use sequences to be produced are identified from the requirements that concern the behaviour aspects of the system. Often, it is the case that particular sequences are of interest ("What if...?"), but were not considered in the collected requirements. In this case, the engineers can decide on an appropriate sequence, and confirm this by raising a question about the collected requirements.

Organizing the use sequences

Usually, the expected behaviour of a system is a set of variants from some nominal sequences. These variants represent all the exceptions that can occur and should be considered when nominal sequences are running.

In order to avoid redundancy between described sequences, and to ease the reuse of parts of sequences, the expected behaviour should be divided into small and reusable Message Sequence Chart diagrams. There are several ways to combine Message Sequence Chart diagrams.

– Message Sequence Chart references are used to refer to other Message Sequence Chart diagrams from within a Message Sequence Chart diagram.
– Complex use sequences should be described by a High-level Message Sequence Chart diagram (HMSC) – a directed graph of Message Sequence Chart references.
– Combination of Message Sequence Chart diagrams can also be described as Message Sequence Chart expressions within Message Sequence Chart references, for example "MSC1 **seq** (MSC2 **alt** MSC3)".
– Small variations where, for example, there is a choice between alternative messages, are best shown by inline expressions.

Simple Message Sequence Chart diagrams should be carefully named in order to give an idea how they could be combined. For example: "no_answer_from_third_party" or "caller_hangs_up".

Identifying the instances in a Message Sequence Chart diagram

The Message Sequence Chart instances appearing in a use sequence should be instances of agents defined as classes in the object model, and the application system. Usually, the Message Sequence Chart instances correspond to physical entities. The logical objects in the environment generally correspond to passive elements that transmitted as parameters to Message Sequence Chart messages.

The system appears as a vertical bar in a Message Sequence Chart diagram. There may be as many instances of the system in the Message Sequence Chart diagram as there are concurrent instances in the real world communicating together, but it is more usual to consider only one system. For example, in the case of an interconnected network of systems, the Message Sequence Chart diagram contains several instances of the system. In the same way, an agent can be multi-instantiated in a Message Sequence Chart diagram if this situation occurs in the real world. Different instances of the same agent should be carefully named in order to have an understandable Message Sequence Chart, for example "Caller" and "Called" in a Message Sequence Chart diagram involving two subscribers in dialogue. It is usual to limit the number of instances in the Message Sequence Chart to the minimum number required to show the behaviour in the real world.

There is no need in Analysis to formally describe the means (communication channels and instance identification) used by the agents and the system to interact. Engineers must consider that the instances can be accessed independently without any problem of identification and connection.

Identifying and drawing the messages

Messages usually correspond to class operations in the object model and their parameters usually correspond to class attributes or results of class operations.

To fulfil the criteria of accessibility of the information, paths (direct or indirect) should exist in the object model between Message Sequence Chart instances that are in dialogue. However, associations of the object model need not necessarily result in messages of the Message Sequence Chart diagrams.

Verifying the completeness of the dynamic model

Each possible scenario should be covered by a Message Sequence Chart diagram. So, the dynamic model is normally completed when all the possible scenarios are covered. Nevertheless, completeness is usually impossible to obtain because the number of possible use sequences is very large. The iterations on the dynamic model will be stopped when the engineers consider that a reasonable set of use sequences has been produced. This choice is subjective and system dependent.

Verifying the consistency of the dynamic model with the object model

The dynamic model must be checked compared to the object model and the missing information must be added to the object model in order to respect the following rules:

1) Message Sequence Chart instances shall correspond to the system or to agent objects (instances of classes).

2) Messages shall correspond to class operations. Usually a message sent from an instance of "a" to an instance of "b" is declared in class "b" showing that "b" provides this service to "a". But sometimes a related operation can also be declared in "a" showing that "a" provides this service to other classes that do not want to directly access to "b" (for example because "b" provides a too low-level service).

3) Message parameters should correspond to class attributes or results of class operations. Attributes should be declared in the class corresponding to the message or declared in other classes accessed through associations.

4) For two Message Sequence Chart instances in dialogue, their corresponding classes shall be connected in the object model, either directly or indirectly through associations.

The consistency between the two models is performed during the last iterations on the models. It is quite common to enrich the object model by operations derived from the messages of corresponding Message Sequence Chart diagram.

Rule 5 – For showing the sequences of messages exchanged between the system and its environment against time in diagrams, the Message Sequence Chart notation shall be used.

Rule 6 – Use sequences shall show the system as a whole interacting with the environment agents, without considering the internal system architecture.

Result: A use sequence model (Message Sequence Chart document), consistent with the object model, which captures in a diagrammatic way the dynamic view of the collected requirements.

14 Draft Design steps

The steps detailed in 14.1 to 14.6 are presented in an order that is reasonable for starting the steps, but they can be carried out in any order except where information from one model is used in another model. The steps can largely be carried out in parallel, because each step focuses on a different way of modelling. Each model may cover all or part of the system.

Because the number of draft designs needed varies significantly, it is possible to indicate only which draft designs are likely to be more useful to Formalization than others. This is summarized in Table 1.

Table 1 – Usefulness of draft designs in Formalization

Draft Design model	Use
Component relationship modelling	Sometimes useful to establish how many blocks, processes or data items there are.
Data and control flow diagrams	Context diagram often useful. Hierarchy of diagrams useful for ASN.1.
Information structure	Needs to be done, either in Draft Design or as part of Formalization.
Use sequence modelling	Use sequences are normally used for process design. Useful for informative parts of the application.
Process behaviour modelling	Only produced occasionally. Used as basis of formal diagrams.
State overview modelling	Useful for checking processes, but need not be in the application.

The use of bit tables to model the information view for interfaces may initially seem attractive, but has significant disadvantages. Bit tables confuse the issues of defining information structure with the encoding of information. Bit tables tend to de-emphasize the purpose of each data element. Sometimes, important data elements are not given a meaningful name in bit tables, because they are just represented by a single bit, or a few bits. Bit tables are not flexible, partly because they mix meaning and structure with encoding: a change of transport medium may require a change of encoding but with no other semantic change. Bit tables can sometimes be misunderstood when it is not clear whether the highest numbered (or leftmost) bit is most or least significant, left or right aligned, or transmitted first or last. The presentation of bit tables is not standardized. Bit tables do not support composition, analytical investigation and systematic checking, which are all well supported by tools for ASN.1 and SDL-2010.

Rule 7 – Bit tables shall not be used to model the information view.

NOTE – Clause 10.7 of [b-ITU-T Z.104] defines how encoding can be expressed in SDL-2010.

Encoding for interfaces is preferably described by using ASN.1 for the data conveyed and one of the recommended ASN.1 encoding rules (identified by one of the encoding rule names BER, CER, DER, PER, APER, UPER, CAPER, CUPER, BXER, CXER or EXER). Alternatively, if ASN.1 is not being used, encoding is preferably determined from the implicit ASN.1 CHOICE type equivalent to the set of signals conveyed and an ASN.1 encoding rule (identified by name). If neither of the previous alternatives is applicable, encoding is preferably described using the text encoding rules or an implementation/application defined encoding as in [b-ITU-T Z.104]. The use of bit tables is acceptable only when ASN.1 is absent, as a way of illustrating encoding.

14.1 Component relationship modelling

The objective of this step is to complete the object model produced in Analysis to describe the internal architecture of the system. This model is used for data and control flow modelling, the information modelling, use sequence modelling and Formalization. The detail added to the object model corresponds to the internal architecture of the system: the information aspects and the design of the associations. However, this modelling is considered part of Analysis, if Draft Design is not included as an activity. There are two ways in which this modelling can be done: as a refinement of the class model from Analysis, or replacing classes with alternative more detailed classes.

Instructions

1) Define the sorts of data of the class attributes and the class operations declared in the Analysis object model.
2) Design the class associations.

These activities lead to an enrichment of the object model by introducing new attributes, new associations, new inheritance links or new classes.

Guidelines

Typing attributes and operations

In Analysis, sorts of data used for attributes and operations should either be simple or names that are not otherwise refined. Lists, tables or complex structures should not have been used. If they are relevant for Analysis (that is, they capture application concepts), they should be modelled as classes and associations, otherwise complex data information items are ignored. For Draft Design, the sorts of data need to be more precisely defined, but describing all the sorts of data in the object model could be redundant effort compared to other steps such as data flow modelling or information structure modelling. Therefore, the modelling sorts of data should be limited to the most important attributes and operations that will aid the data flow modelling without making superfluous work.

In simple UML class diagrams, sorts of data are declared at the model level only as a class and then are accessible by all the classes; sorts of data cannot be declared locally to a class without using other kinds of UML diagrams. Attributes are typed either by creating new global sorts of data or by adding aggregations or associations if the corresponding complex data structures contain classes or accesses to classes. New classes can also be added to the model if the complex data structures contain more information than the one available in the Analysis object model (through the attributes).

Adding data sorts for parameters and results of operations could also lead to a reorganization of the inheritance hierarchies to ease redefinition of operations. Operation signatures should be compliant with the inheritance hierarchies.

Designing associations

The associations defined in the Analysis object model are bidirectional and enable the access to class instances without knowing how it is possible. The Draft Design should explain how to access the instances. Generally, for each association, a privileged direction should be chosen and for the addressee class of this association, the "key attribute" should be chosen among the existing ones (or should be created if it does not yet exist), this key attribute identifying an instance of this class among the other instances. New classes could also be added to solve the problem of associations with multiple cardinalities on both sides.

Rule 8 – The object model is completed by using the same notation than this one used for Analysis.

14.2 Data and control flow modelling

Instructions

1) Produce a context diagram.
2) Decompose the SDL-2010 agent (usually a block) in the context diagram into subagents (usually blocks) using the top-level aggregation and decomposition structure from the classified information (that is, the object model).
3) Repeat decomposition for the subagents until the behaviour of each agent cannot sensibly be further divided or it is obvious that the agent corresponds to a single SDL-2010 process.

Guidelines

The context diagram is an SDL-2010 system diagram with no external channels containing a single agent that represents how the system behaves internally and one of more agents representing the external agents in the environment. The channels leading to and from the internal agent represent the information flow between the system and the external agents in the environment. In the classified information, these can be identified from the associations between the system and agents in the environment and the Message Sequence Chart events between system instances and agent instance. These are the **interface** aspects of the system. There is one channel for each separately identified information flow (usually one per agent instance in the Message Sequence Chart). If it is required that two or more information flows are merged in the environment, they are shown connected to the same point on the boundary of the internal agent. Signals are not attached to the channels, but the information flow is inferred from natural language descriptions.

If there is no class in the classified information for the system, the system is made from an aggregation of internal classes (that is, not classes for agents in the environment) at the top level that are not parts of other classes (that is, not aggregated).

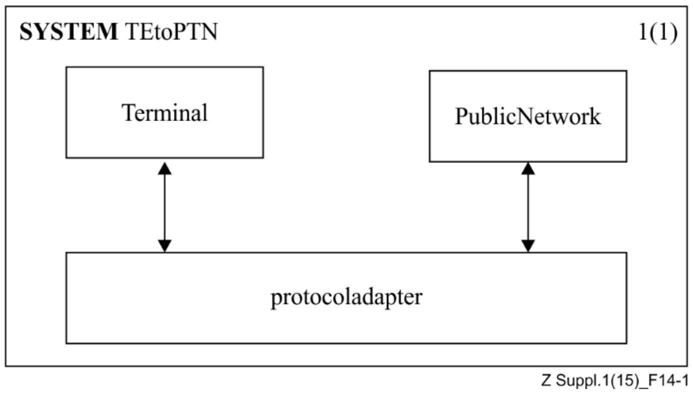
The subagents that have strong **information** aspects and little or no specific behaviour (other than storing information) are annotated as "data". There should never be a direct connection between two such "data blocks". Channels are identified from interface information. Channels to and from "data blocks" indicate data flow. Channels between other (behaviour) blocks are control flow. There should be a description of the behaviour of these blocks. If such a description does not exist, write one in natural language.

These descriptions use SDL-2010 to show data and control flow but do not show signal detail or process behaviour. The approach allows various decompositions to be investigated and to focus on the data flow and data repositories. Whenever a "data block" has connections to more than one block, the decomposition should be reconsidered, as the data should be contained in one process in the formalized model. If the "data block" cannot be contained in one process, in SDL-2010 it can be represented as data definitions in an enclosing agent, in which case the enclosed agents access the data by implicit remote procedure calls. In earlier versions of the Specification and Description Language (or in tools not supporting data in enclosing agents as in SDL-2010) a "data block" is represented either by the import of one or more remote variables exported by an agent representing the "data block" or by remote procedure interfaces to the agent representing the "data block".

The accepted architecture used in the area of application should be considered. For example, an architecture that separates information flow may be a requirement, so that there are different protocol planes: "user plane", "control plane" and "management plane".

Often, the full context cannot be specified using a top-down approach. In this case, the initial context diagram may need some bottom-up compilation from process specifications.

Rule 9 – Any context diagrams shall be in SDL-2010.



Example 14-1 – Context diagram for protocol adapter

NOTE – For the context diagram, the SDL-2010 need not be complete; for example, the channels have no names and signals are not defined in Example 14-1.

14.3 Information structure modelling

The information structures in the system are modelled using ASN.1 to elaborate the information view from the class model within the classified information. Even if the information coding and structure are mixed in the collected requirements, the encoding should be considered separately from the information structure. A logical model of the system can be produced without encoding, and it is only when the constraint of matching the information structure to bits is considered that encoding is needed. If default encoding rules are satisfactory, the actual encoding can be ignored.

Instructions

1) Identify the information flows in the least abstract (most detailed) data and control flow diagrams, or if Formalization is already in progress, the channels from the blocks containing processes and signal routes within these blocks.
2) Identify the structure from information aspects associated with the information flows and define a meaningful ASN.1 type name for each sort of data that is not part of an aggregation (the top-level data messages).
3) Identify and name the next level of the structure in the top-level data messages, choosing the same meaningful ASN.1 type name if the same information is used in several places.
4) Define the higher level ASN.1 type in terms of the next level type names and repeat the last two steps until every part is a simple ASN.1 type.
5) Identify the structure from the information aspects of the innermost blocks (both behaviour and data) in the least abstract (most detailed) data and control flow diagrams.
6) Identify and name the sorts of data within these blocks in a similar way to the data for interfaces, but reusing the ASN.1 types defined for the interfaces.
7) Identify any values of the ASN.1 types that are referred to by name, define ASN.1 names for these values starting with a simple ASN.1 type and reusing the value names where appropriate in compound ASN.1 types.
8) Identify from the behaviour aspects of the innermost blocks, identify the operators needed for the sorts of data, and record these as a comment to the ASN.1.

Guidelines

Usually there will be aggregate sorts of data for different messages, often called a Protocol Data Unit (PDU) for protocols. The structure of these may be outlined or defined in the collected requirements and the class model in the classified information.

The intermediate ASN.1 types are chosen so that they can be used in different messages. In particular, when information is transformed from one protocol to another, there are often some common data elements that are transferred from messages at one interface to messages at another interface.

Aggregation corresponds to compound types in ASN.1, so that SEQUENCE, SEQUENCE OF, SET, SET OF and CHOICE are used when decomposing types.

In addition to searching through the types defined as part of this step, the library of ASN.1 "useful types", and ASN.1 definitions used for the interfaces of standards related to the application should be consulted.

During development it may be convenient to use the ASN.1 "ANY". When the ASN.1 type definitions are complete, "ANY" should have been replaced by a specific name, possibly externally defined.

If bit tables have been provided in the collected requirements, then these should be rewritten using SDL-2010 encoding. Encoding probably only needs to be defined for protocols over normative interfaces, such as the external interfaces of the system or between parts of the system possibly produced by different providers. ASN.1 basic encoding is not always suitable for a protocol; for example, it may be too inefficient. In this case, another standardized ASN.1 encoding should preferably be used or an explicit encoding for the particular application. A simplified form of the bit tables may be needed in the documentation to describe encoding. For those SDL-2010 data (including the language syntax bindings defined in [b-ITU-T Z.104C]) may be used instead of ASN.1 if:

– ASN.1 is not required for interface definitions;
– ASN.1 basic encoding is not to be used;
– SDL-2010 provides good support for the sorts of data needed.

If there is no reason to prefer SDL-2010 data, then ASN.1 data should be used.

Explicit encoding is left until as late as possible. Sometimes it can be done after Formalization. The encoding of messages does not change the behaviour of the formal SDL-2010 description, but encodes the interfaces so that the bits conveyed match use in the environment.

Rule 10 – ASN.1 "useful types" and other already standardized types should be used whenever possible.

Rule 11 – Encoding shall be done separately from the structuring and naming of types.

Example

See Table 2.

Table 2 – ASN.1 for Digital trunk radio call control set-up

```
CC-SETUP::= SEQUENCE
   {
{  pd    ProtocolDiscriminator;
   ti    TransactionIdentifier;
   mt    MessageType;
   pi    PortableIdentity OPTIONAL;       --only present if "incoming call" implemented
   fdi   FixedIdentity OPTIONAL;          --only present if "incoming call" implemented
   bs    BasicService OPTIONAL;           --only present if "incoming call" implemented
   ia    IWU_Attributes OPTIONAL;         --mandatory if basic service indicates "other"
                                          --not allowed if basic service indicates
                                          -- "default attributes"
   ri    RepeatIndicator OPTIONAL;        --if ri appears before call attributes,
                                          -- not after, this indicates a prioritized
                                          -- list of call attributes for negotiation
   cla   SEQUENCE SIZE(0..3) OF
         CallAttribute OPTIONAL;

   ri    RepeatIndicator OPTIONAL;        -- if ri appears after call attributes
                                          -- and not before
   cna   SEQUENCE OF
         ConnectionAttribute OPTIONAL;

   cri   CipherInfo OPTIONAL;
   cni   ConnectionIdentity OPTIONAL;
   fy    Facility OPTIONAL;
   pi    ProgressIndicator OPTIONAL;      -- not allowed if signal is sent from P to F
   dy    Display OPTIONAL;                -- not allowed if signal is sent from P to F
   kp    KeyPad OPTIONAL;                 -- not allowed if signal is sent from F to P
   sl    Signal OPTIONAL;                 -- not allowed if signal is sent from P to F
   fea   FeatureActivate OPTIONAL;        -- not allowed if signal is sent from F to P
   fei   FeatureIndicate OPTIONAL;        -- not allowed if signal is sent from P to F
   np    NetworkParameter OPTIONAL;       -- not allowed if signal is sent from F to P
   tc    TerminalCapability OPTIONAL;     -- not allowed if signal is sent from F to P
   eec   EndToEndCompatibility OPTIONAL;  -- mandatory if system uses
                                          -- LU6(V.110/I.30 rate)
   rp    RateParameters OPTIONAL;         -- mandatory for call set-up of a
                                          -- rate adaptation service
   td    TransitDelay OPTIONAL;
   ws    WindowSize OPTIONAL;
   cgpn  CallingPartyNumber OPTIONAL;
   cdpn  CalledPartyNumber OPTIONAL;
   cds   CalledPartySubaddress OPTIONAL;
   sc    SendingComplete OPTIONAL;        -- mandatory if all necessary
                                          -- information for call set-up
   iti   IWU_to_IWU OPTIONAL;
   ip    IWU_Packet OPTIONAL
}   }
```

14.4 Use sequence modelling

The use sequence modelling has been started in the Analysis phase capturing the point of view of the system user only. At this stage, an SDL-2010 data and control flow model is available and the use sequence model will be detailed in accordance to this SDL-2010 description.

The additional tasks mainly concern: the use of SDL-2010 entities instead of classes of the object model, the use of SDL-2010 signals instead of events related to the object model, the replacement by the real protocols of the abstract exchanges modelled in Analysis. For example, in the Analysis model, information generated by an external entity is sometimes modelled as a unique abstract message, whereas in Draft Design, this information is modelled by the set of interactions that occurs in the real world possibly with several messages. These additional interactions lead also to introduction of new Message Sequence Chart diagrams detailing the previously abstract exchanges.

Because Message Sequence Chart diagrams are often used to support the behaviour steps in Formalization, Message Sequence Chart diagrams may need to be produced as part of Formalization. In that case this step, in combination with the step in clause 13.3, is used to generate the Message Sequence Chart diagrams.

Instructions

1) Follow instructions 1 to 5 of clause 13.3 to produce Message Sequence Chart diagrams, but model Message Sequence Chart instances within the system that correspond to SDL-2010 agents (blocks or processes) internal to the system.
2) Check the consistency between the Message Sequence Chart diagrams at this level and the Message Sequence Chart diagrams (if any) in the classified information.
3) Produce (optionally) Message Sequence Chart diagrams for lower level interactions within SDL-2010 and check the consistency with Message Sequence Chart diagrams at higher levels.

Guidelines

Interfaces to the environment (or for a Message Sequence Chart internal to an agent, to higher levels) should use the sides of the chart. However, if there are three or more interfaces with the environment, it is more convenient to show these as instance axes.

There may be more than one instance of the same class on a message sequence chart. For example, there may be two instance axes that both represent different instances of a call control or interfaces with the environment.

If there is only one instance of each class or SDL-2010 item, the names of the instances on the chart can be the same as the names of the corresponding classes or SDL-2010 items. If there is more than one instance, they must have unique names and the class (or SDL-2010 item) is instead indicated by their kind. For example, in Figure 14-2 PTNX is an instance kind and PTNX_A is an instance name.

The messages have usually already been identified before Message Sequence Chart diagrams are drawn, either as part of the behaviour and interface aspects of the classes corresponding to the instances in the classified information, or as signals used in SDL-2010. The sequence of messages can be identified from the behaviour and interface aspects of the classes in the classified information.

It is not essential to divide Message Sequence Chart diagrams into simple diagrams that are referenced from one main diagram, but this is useful when many sequences have the same "set up" or "clear down". When Message Sequence Chart references to simple diagrams (and Message Sequence Chart conditions) are used, they should be given meaningful names such as "call_in_progress".

The choice of a reasonable set of use sequences is subjective and system dependent. It is not usually possible to draw all the possible message sequences for a particular system, as the number is very large.

Rule 12 – For showing the sequence of messages exchanged between parts of the system (with the environment) against time in diagrams, the Message Sequence Chart notation shall be used.

Example

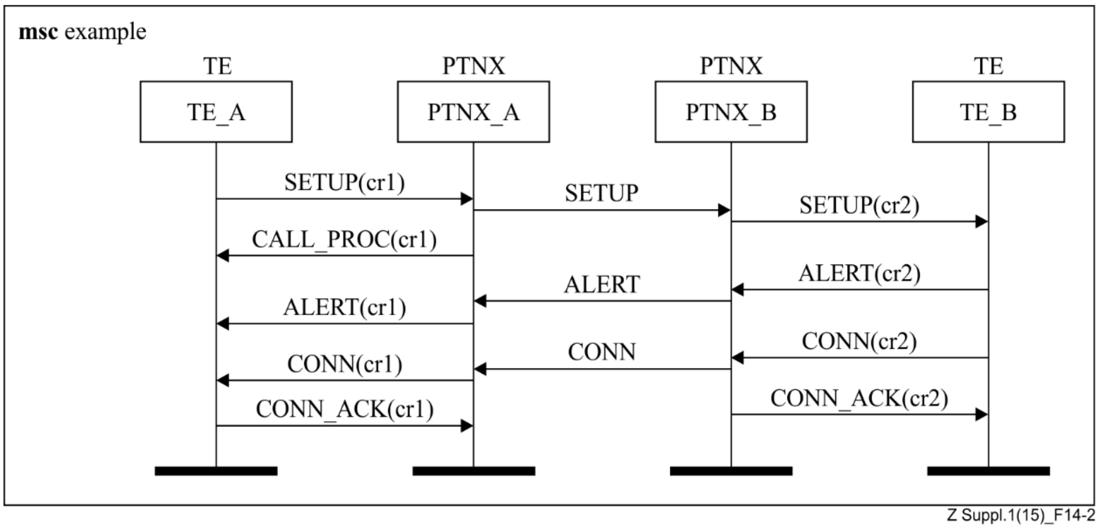


Figure 14-2 – En-bloc sending, successful call

14.5 Process behaviour modelling

The modelling is achieved by the informal use of SDL-2010 agent diagrams with states and transitions to sketch the behaviour of agents. During Draft Design, this step is usually only used for critical agents. The behaviour of other agents is inferred from use sequences and other descriptions. The set of signals may not be formally defined, and SDL-2010 "informal text" is used rather than formal expressions in tasks, decisions and parameters.

NOTE – An agent diagram containing states and transitions is shorthand for an agent based on an agent type that has a state machine defined by the states and transitions – see clause 9 of [b-ITU-T Z.103].

Because this modelling can be done in a way similar to Formalization, it is not described in detail in this step. The essential difference between Formalization and the process modelling in Draft Design is the level of formality. Draft Design SDL-2010 diagrams may be informal, incomplete, inconsistent and invalid SDL-2010. The rules of Formalization need not apply. This does not prevent the diagrams from providing useful views of the application, and allows economies to be made.

14.6 Overview modelling

This modelling uses the SDL-2010 auxiliary diagrams to provide overviews of the system behaviour. Such diagrams are tree diagrams, state-overview diagrams, and state-signal matrix diagrams (see below clause 14.6.3).

Overview modelling also can be useful to annotate the states in the overview (or in the SDL-2010 sketches in clause 14.5) with properties of the state expressed as logical expressions. These expressions can be useful when defining test purposes.

Auxiliary diagrams contain information that it is possible to extract (manually or by means of a tool) from an SDL-2010 system specification. The purpose is to provide overview. However, these diagrams can also be created before the SDL-2010 system specification, and can then serve as a basis for this.

14.6.1 Tree diagram

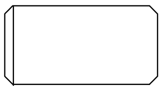
A tree diagram shows the components of an SDL-2010 system. The components (also called nodes in the sequel) form a hierarchical structure, with the system as the root, according to the containment relationships expressed by the syntax rules. The nodes of a tree diagram can be (in addition to the system-node):

– blocks only, called *block tree diagram*; for an example see Figure 14-3;

– blocks and processes, called *basic tree diagram*; for an example see Figure 14-4;

– in addition to the nodes of *basic tree diagram* also procedures and macros etc., called *general tree diagram*; for an example see Figure 14-5 which includes a **block type**.

Note that the different branches of a tree diagram need not contain the same number of nodes. All nodes represent definitions of entities. For the macro node the following symbol is used.



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 a 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.

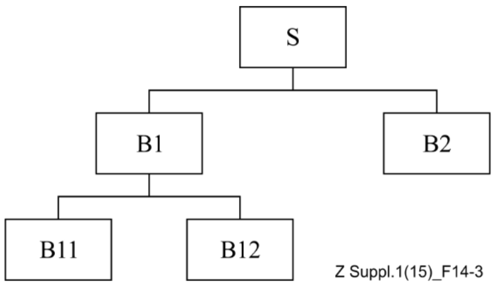


Figure 14-3 – Block tree diagram

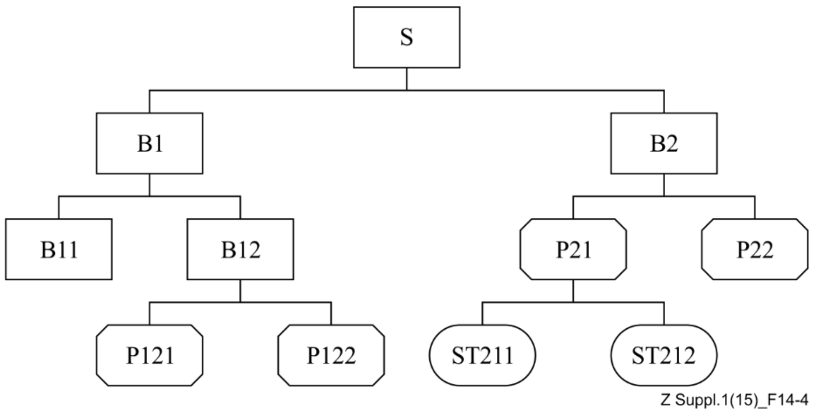


Figure 14-4 – Basic tree diagram

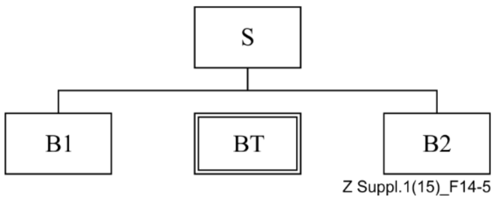


Figure 14-5 – General tree diagram

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 14-6, which splits the diagram in Figure 14-4 into two diagrams. 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.

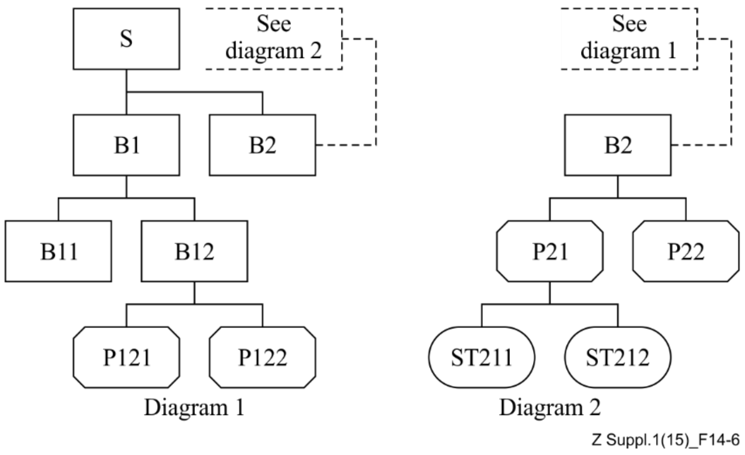


Figure 14-6 – Partitioning a tree diagram

14.6.2 State-overview diagram

The intention with the state-overview diagram is to give an overview of the states of an agent and show the possible transitions between them. The diagram soon gets very complicated when the number of states and transitions increase. 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 preceded by the character /. Timer signals and setting of timers can be treated as ordinary signals. An example of a state-overview diagram is given in Figure 14-7.

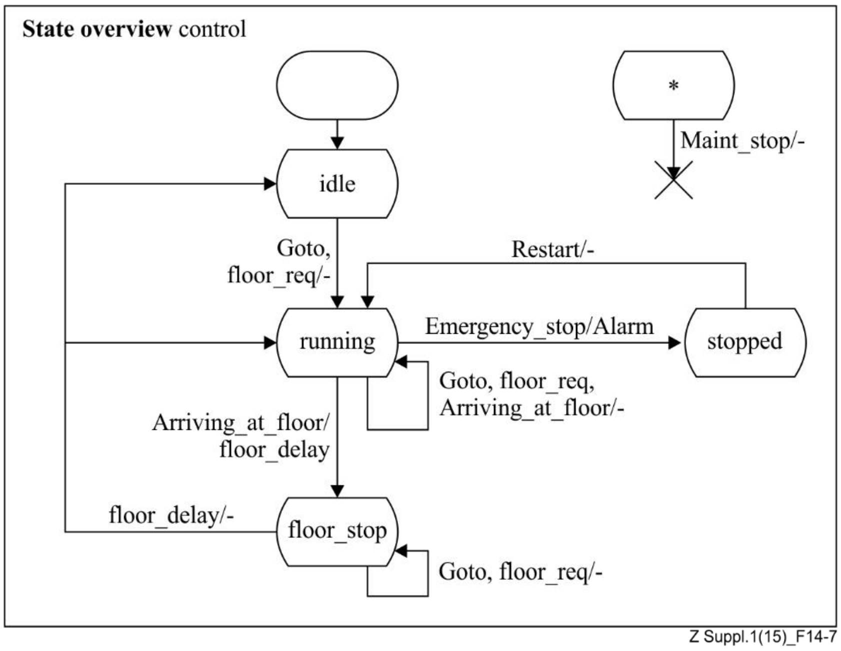


Figure 14-7 – A state-overview diagram

14.6.3 State-signal matrix

The state-signal matrix is an alternative to the state-overview diagram containing exactly the same information. It can also be used however when the number of states and transitions is large. In addition, it offers a possibility to check that all combinations of states and input signals are considered. See Figure 14.8, which corresponds to the state-overview diagram in Figure 14.7.

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

The element corresponding to the dummy state "start" and the empty signal is used to show which is the initial state of the agent.

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 agent behaves.

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 14-8 – A state-signal matrix

15 Formalization steps

The steps for Formalization are divided into groups for Structure (S-steps), Behaviour (B-steps), Data (D-steps), Types (T-steps) and Localization (L-steps). One group of steps does not have to be completed before another group is started, and it is usual for work to be in progress in more than one group at once. For example, the D-steps can be used to define the parameters of signals in parallel with S-steps or B-steps. Similarly, as the system is refined, it is possible to be applying S-steps to one block, while another block is being detailed with B-, D-, or T-steps. To take full advantage of types in SDL-2010, the T- and L-steps should be applied at the same time as the S- and B-steps.

NOTE – In all steps where an agent (system, block, or process) or composite state or procedure is defined, always consider searching for an existing type that might be used or modified. This guideline is placed here rather than be repeated it in every step where it applies.

If UML object models have been used in Analysis, Table 3 gives a general guide for the derivation of SDL-2010.

Table 3 –Transformations of UML object models into SDL-2010

UML object model notation	Possible SDL-2010 transformation
object	agent, data declaration
class	block type, process type, state type, (data) type (or syntype)
instantiation relationship	process create line
attribute (of non-data class)	data declaration of local variable
attribute (of data class)	structure field or two operators (get, set)
attribute type, parameter type	data sort of the SDL 2010 construct derived from the attribute
operation	procedure (local, exported, imported), signal definition, operator
parameter of an operation	parameter of (procedure or signal or operator)
simple inheritance	inheritance
association, link	channel
association name	name of channel
multiplicity of aggregations	number of instances (of agents), array (of data)
role name	gate, variable name
aggregation	structure for agents, local variables in an agent, two operators (or a struct) within data

Constructs of the UML notation not mentioned above are not usually used.

Further guidelines can be found in [b-Guo] and [b-Witaszek], but recognizing that the notation in these articles was subsequently incorporated into [b-OMG UML].

15.1 Structure steps (S-steps)

Structure steps are used to define (in SDL-2010) the external and internal interfaces of the system and static partitioning of the system into SDL-2010 blocks.

15.1.1 Step S:1 – Boundary and environment of the system

Instructions

1) Identify the boundaries between the system to be described and its environment.
2) Find a suitable name for the system.
3) Draw an SDL-2010 system diagram with an identified name and explain the system and its relation to the environment informally in a comment within the SDL-2010 system diagram.

Guidelines

This step might seem trivial, but the importance of the preparation for this step should not be underestimated. Analysis or Draft Design done before this step will help in choosing the right boundary and an appropriate name. Do not take this step until there is a reasonable understanding of the requirements for the system.

The system name and the comment description can probably be derived directly from the classified information model. The system name and the comment description will appear in the formal SDL-2010 description and should therefore be appropriate for the complete system.

The classified information can contain descriptions of classes both inside and outside the system. There can be one class that is easily identifiable as suitable for the SDL-2010 system, or it may be necessary to identify a set of classes that are interfaced to form the system.

When the classified information contains a behaviour aspect for every class and no interfaces identified as linked to the environment, the SDL-2010 system may be a closed system with no external interfaces. In this case, the SDL-2010 system describes the behaviour (of concern) of the application and the classes that communicate with it. These classes are informative.

A "context diagram" (produced using SDL-2010 informally as a draft design – see 14.2) and Message Sequence Chart diagrams can be useful in determining the boundary between the SDL-2010 system and the environment. In considering the suitability of an SDL-2010 draft design "context diagram", particular attention is given to the system name and descriptive comment. How much of the environment is in the formal SDL-2010 description is also considered. The normative interfaces for the application may need to be internal interfaces of the SDL-2010 system. If parts of the environment are included as part of the SDL-2010 system, make sure they are clearly identified (by name and/or comment) environment parts.

Message Sequence Chart diagrams are often produced as draft designs after this step has been done. If Message Sequence Chart diagrams have already been produced, distinguish between the axes corresponding to entities considered to belong to the system and the axes for entities that belong to the external environment. The latter then have no formal description in the SDL-2010 system (or are provided only as informative parts of the system). It is possible to consider the system as composed of communicating SDL-2010 systems, but in this case the system should be described as a single SDL-2010 system with blocks representing the communicating parts. Communicating SDL-2010 systems may be considered when the collected requirements are relevant to a set of related applications.

Consider whether there are any useful types in a library, which relate to the system or the environment interfaces and which might be put into an SDL-2010 package.

If the system has any timers, the unit of time needs to be defined. Normally suitable values are either the time unit equals 1 millisecond or the time unit equals 1 second. It is also useful to define SDL-2010 synonyms or textual macros for useful multipliers such as "seconds" and "hours".

Rule 13 – The system boundary shall be identifiable as communicating only by means of exchanged discrete messages.

Rule 14 – The unit for time data should be recorded in a comment in the system diagram.

Result: The result is a meaningful name for the system and a description in a comment.

15.1.2 Step S:2 – Discrete system communications

Instructions

1) Identify the information flow in terms of discrete messages to be communicated between the system and its environment.
2) Model these messages by signals definitions.
3) State the relation between each signal and objects external to the system in a comment.
4) State the purpose for each signal in a comment with the signal definition.
5) Group related (often all) signals for one object in one direction in an interface definition.
6) Include any signal definition only used in one interface definition in the interface definition.
7) Include the remaining signal definitions and interface definitions in the system diagram, and add these signals to the interface use list for each interface definition where it is used.

Guidelines

The signals to and from the environment can usually be identified in the classified information as interface aspects in the outermost class(es). Related information aspects describe the content of the signals.

The "context diagram" or Message Sequence Chart diagrams at the system level allow the signals to be identified. The number of signals may be reduced by qualifying signals with parameters (see step D:1). The signals for external communication are defined on the system level and correspond to discrete events between the system and its environment. These may be well defined in the information view as protocol description units or information flow. There may also be calls of remote procedures that can be identified to or from the environment, in which case the remote procedure signatures should be defined and included similarly to the signals for external communication. Each remote procedure call corresponds to a pair of signals (see clause 10.5 of [b-ITU-T Z.102]). Similarly, the use of remote variables (see clause 10.6 of [b-ITU-T Z.102]) may also be identified, defined and included.

If the three-stage methodology (see clause 13.1) is being used as a framework and a stage one description has been produced, then the signals needed to communicate with the user are the signals needed to communicate with the environment. Sometimes, particular behaviour is required from the user, in which case part (or all) of the user behaviour can be modelled within the system.

A system that includes the subject of the application and all communicating objects is a closed system, and has no signals to communicate with the environment.

When a signal with a parameter is introduced, identify or name a corresponding sort of data. The sort of data should correspond to a named ASN.1 type in the information view.

The signal definitions and interface definitions usually are too large to include in a text symbol on the first SDL-2010 page of the system diagram in the documentation. Additional pages should be added to the system diagram to contain these textual descriptions. Lengthy statements about the relationship between signals and external classes, or descriptions that contain informal drawings should not be placed in the SDL-2010 diagram. Such statements should be referenced from the SDL-2010. Shorter statements should be placed with the appropriate SDL-2010 definitions.

NOTE – It is recommended to place the textual parts of the SDL-2010 inside SDL-2010 diagrams. Placing the text in a text symbol on a diagram clearly identifies the text as SDL-2010 and also identifies the diagram where the text belongs.

A **signallist** definition is a legacy, shorthand form of interface definition that excludes package use, interface specialization, and the inclusion of signal, remote procedure and timer definitions in the interface. The long form of interface definition (with the keyword **interface**) is preferable. Both forms are shown below.

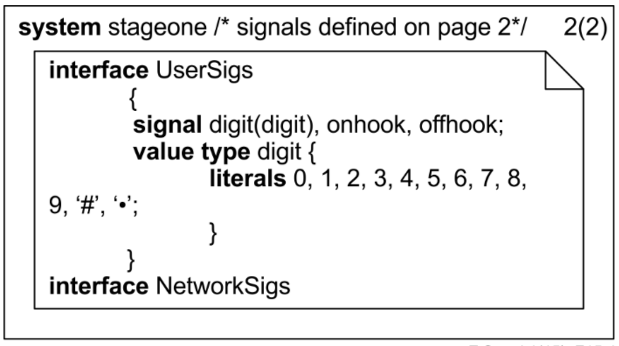
To assist documentation, **signallist** definitions are placed before the definition of signals used in the signallist. Data definitions are placed after signal definitions. The definitions are grouped into logical pieces by placing related definitions in one text symbol.

Rule 15 – The textual definitions of a diagram should be placed in text symbols on diagrams.

Rule 16 – If textual definitions occupy more than 50% of a diagram, the diagram should be split into two or more pages with the textual definitions in logical groups in separate text symbols.

Result: The result is the "alphabet" of the system: the signals defined at the system level to communicate with the environment, although these signals may later need to be refined.

Example



```
system stageone /* signals defined on page 2*/     2(2)
  interface UserSigs
          {
          signal digit(digit), onhook, offhook;
          value type digit {
                  literals 0, 1, 2, 3, 4, 5, 6, 7, 8,
  9, '#', '•';
                  }
          }
  interface NetworkSigs
```

Figure 15-1 – Interface definition from step S:2

NOTE – ASN.1 has not been used for 'digit' as it was intended that the names be 0, 1, 2 … "#" and "*".

Using the legacy **signallist** for the interface definitions and a legacy **newtype** data definition, the text box contents would be:

> **signallist** UserSigs = digit, onhook, offhook;
>
> **signal** digit(digit), onhook, offhook;
>
> **signallist** NetworkSigs = ringing, dialtone, ringtone, speech;
>
> **signal** ringing, dialtone, ringtone, speech;
>
> **newtype** digit **literals** 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, "#", "•"
>
> **endnewtype** digit;

15.1.3 Step S:3 – Externally identifiable parts

Instructions

1) Identify the main parts within the system and draw them as the agents (usually blocks, but possibly processes) in the system.
2) Find a suitable name for each agent and describe the agent and its relation to its environment (its enclosing structure) informally in a comment within the block.

NOTE – Steps S:3 to S:6 are used recursively when an agent is decomposed into subagents. When the steps are reapplied, the "system" being considered is actually an agent, and the word "system" should be considered replaced by the phrase: "enclosing agent". The difference between an enclosed agent and the system agent is that the signals and data used to communicate with the surrounding structure have to be defined **outside** the agent for an enclosed agent, but can be defined **inside** the system agent for a system. In both cases the signals and data can alternatively be defined in a package the agent uses.

Guidelines

Classified information contains component classes that correspond to the contained agents (connected by channels) and type definitions in SDL-2010. The component classes of the system (or agent when this step is used recursively) correspond to agents and channels. Some classes with no behaviour aspect may represent objects in the environment.

Any architectural requirements, such as splitting the system into separate protocol planes, are used to help identify system parts.

The internal agents are handled in this step, and the channels, in step S:4. The agents are objects that contain a behaviour aspect with internal states. The states may correspond to data described in the information aspect. The engineer has to decide whether a class should be described in SDL-2010 as a block type, process type, procedure, signal, timer, or sort of data. In most cases the choice of an SDL-2010 type corresponding to the classified information is obvious, but in some cases it might depend on the intended abstract presentation of the formal specification. Object behaviour can be described in a state based way (process, procedure) or operators in data type definitions). In this particular case a state based description is recommended, except if the behaviour clearly corresponds to data functions.

Structure within draft designs can also be used to determine the agent structure.

Agents delimit visibility. For this reason, signals, sorts and types that are used only within an agent should be defined within that block, so that information is hidden from higher levels and therefore makes these levels easier to understand. This principle also applies for these items in steps S:6 and S:7, and is expressed as a general principle for information hiding in Rule 18.

An agent is "informative" if it has no behaviour that is to be normative. The purpose of an informative agent is to make the system complete, so that the function of the system can be:

– understood by its use of and interaction with these informal parts;

– executed and analysed, leading to a better quality product and supporting Validation.

An agent can be informative only if all the agents, composite states and procedures (including remote procedures) it encloses are informative. Once an agent (or composite state or procedure) is marked informative it is implied that all the enclosed agents, composite states and procedures are informative and it is not necessary to mark these agents, composite states and procedures as "informative". Types (agent types, state types and data types) enclosed in an informative item are also informative.

NOTE – "informative" is not the same as "informal". In the formal SDL-2010 description, both informative and normative parts should be formal (that is, expressed formally).

When there are large number of agents directly enclosed within the system (or an agent) some of them are grouped together and encapsulated in a block as in step S:6.

The enclosed agents each represent a set of one or more agent instances. If it is obvious that two (or more) of these agent instance sets behave in the same way (for example, the terminations in a symmetrical end-to-end system), an agent type can be defined and a type based agent definition used.

Rule 17 – There should be no more than five blocks at the system level (or directly enclosed within an agent).

Rule 18 – A definition shall have the smallest scope that includes all uses of the defined item.

Rule 19 – If an agent (composite state or procedure) is informative and not enclosed itself in an informative agent (composite state or procedure), it shall have the annotation "informative" in the diagram referencing it or in its referenced diagram or in both places.

Result: The result is a diagram containing a set of agents and possibly identification of types for these agents.

Example

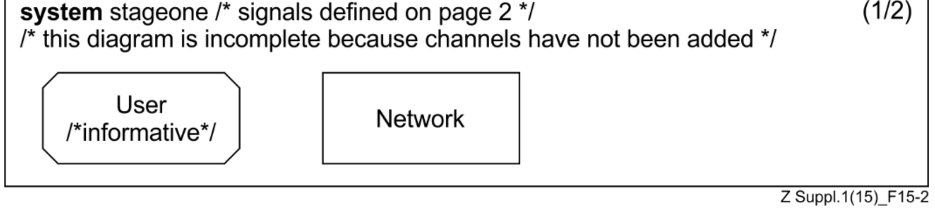

system stageone /* signals defined on page 2 */
/* this diagram is incomplete because channels have not been added */
(1/2)
User /*informative*/
Network
Z Suppl.1(15)_F15-2

Figure 15-2 – user and network for an ITU-T I.130 [b-ITU-T I.130] stage 1

15.1.4 Step S:4 – Communication paths between parts

Instructions

1) Identify the channels needed between agents and the boundary of the diagram and between agents within the diagram.
2) For each channel, identify the direction(s) of communication.
3) Associate an interface (possibly defined by a **signallist**) with each direction of the channel.
4) Choose an interface (**signallist**) name related to the function/usage of the communication.

Guidelines

The relevant description of how the internal agents within a diagram are connected is in the interface aspects of the class that corresponds to the diagram in the classified information. The channels might be represented explicitly as classes in the classified information, particularly if there are specific requirements on the channel.

The interfaces identified in step S:2 correspond to the ends of a channel at the system boundary. Each point on the boundary represents the communication with a different external object (a channel or agent external to the enclosing agent). There can be one or more channels connecting such a point to the agents in the SDL-2010 agent diagram. For the system agent each channel groups some flows of the system behaviour in the "context diagram" in the draft designs. Realistic modelling requirements such as independent interfaces are taken into consideration. If the interface name from S:2 seems inappropriate, rename it.

If the enclosing agent contains a single agent, this agent is connected to the boundary of the enclosing agent through channels. Otherwise, the contained agents at the diagram level are also connected by channels according to the flows of the information between the agents. There are no good reasons for an agent diagram to contain a single agent. An agent (even the system agent) that contains a single agent can be replaced by that agent.

Typical communication cases represented in Message Sequence Chart diagrams help to identify the channels.

Special effects that depend on the delay or non-delay of each channel should not normally be used. If only one channel is used between two agents, signals sent from one agent to the other arrive in the order they were sent at the other agent. Channels should be assumed to have a delay, unless there are requirements to communicate without delay.

If the communication on a channel needs to have specific messages with a specific format and encoding according to the classified information and collected requirements, the channel is "normative". Channels that are internal to the system and do not identify a particular interface for product testing or other purposes are usually informative. The communication on informative channels contributes to the behaviour of the system, but there could be an alternative system with different channels or communication that has the same behaviour.

Rule 20 – There should be only one channel between two agents.

Rule 21 – Every channel that is normative shall have the comment "normative" attached.

Result: The diagram is defined with a set of communication paths corresponding to the external interfaces and internal interfaces between the directly enclosed agents.

Example

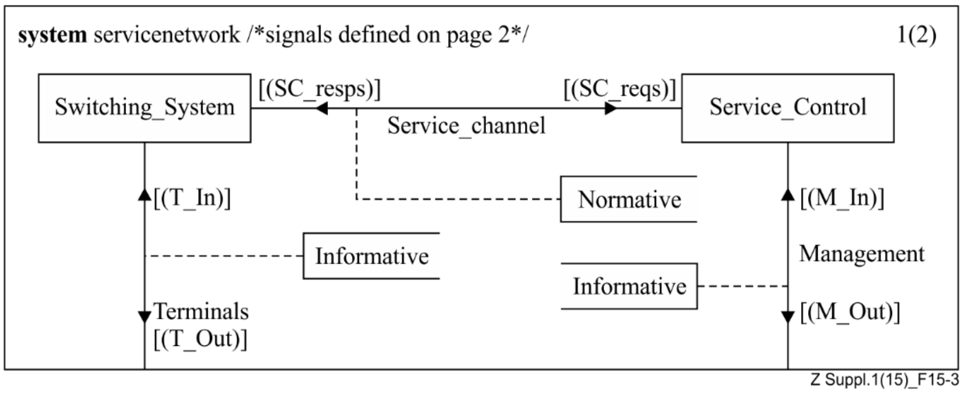


Figure 15-3 – A model for a network to handle services

15.1.5 Step S:5 – Associating signals to communication paths

Instructions

1) For each new interface name, identify the appropriate signals.

2) Name and define each new signal.

Guidelines

The association of signals to interfaces used between agents can require the definition of new signals in the diagram containing the agents.

The relevant description is in the internal interface aspects of the system in the classified information, or (if the channels are described as part-objects) information about the signal given by the corresponding channel in the classified information.

If a draft design has been produced for the SDL-2010 diagram, then reconsider each interface associated with a channel direction. Check also the messages between the relevant axes of each Message Sequence Chart.

Consider whether to redefine the interface associated with a channel at higher levels, making use of the new interface. For example, if a higher-level interface (named Hli) uses A, B, C, D and E, and the new interface (named Ni) uses A, B and C, Ni can be defined at the higher level and Hli can use Ni. This can improve the structure and clarity of the SDL-2010. The interface definition needs to be within the highest-level diagram in which it is used.

Although SDL-2010 allows the signal list symbol (that is, [...]) by a channel on a diagram to contain signals directly, it is not recommended to list the signals in the symbol. The list of signals is usually needed in more than one place, and it is easier to modify the list if it is defined once by an interface definition.

Rule 22 – No more than three signals (or interface names) should be listed in a signal list symbol, instead use a single interface name attached to the channel.

Rule 23 – The signals and data used in the external communication of a system should be defined in one (or more) text symbol(s) in interface definitions separate from other definitions.

Result: The signals are associated with interface names (that is, indirectly to channels).

15.1.6 Step S:6 – Information hiding and substructuring

Instructions

1) Consider each block agent within the current diagram in turn and decide whether the block should contain blocks or processes.

2) If the agent is to contain blocks, recursively apply the sequence of steps S:3 to S:6 to the block, regarding it as a (sub) system on its own and introducing new required signals, blocks, channels and interfaces.

3) Otherwise, apply step S:7 to divide the block into processes.

Guidelines

If the system is large, some block agents can be considered a system on their own, and can be further partitioned according to the rules given for the system. This results in the nesting of blocks. Each block can then be elaborated as described above. This step can be derived from the nesting of classes in classified information. A class that has behaviour as the main aspect and is not further partitioned is a good candidate to be a process; therefore, the directly enclosing object is a block. A class that contains further partitioned classes is usually a block to be partitioned.

It may not be clear whether the contents of a block should be blocks or processes, in which case a division into blocks is initially attempted. If it is difficult to partition a block further, it should probably be a process.

In SDL-2010 the only real distinction between a block agent and a process agent is the scheduling of contained state machines. Within a process only one state machine can be interpreting a transition at any one time and interpretation of contained state machines (of the agent and other contained process agents – block agents are not allowed within a process agent) is interleaved. Within a block each contained agent can (subject to resource constraints) logically be interpreted concurrently. Any agent (system, block or process) can contain a state machine definition. A block agent might contain a state machine that handles access to shared resources. It is therefore suggested to use blocks primarily for structuring, minimizing the need for state machines within blocks, and to use processes primarily for containing the main state machines of the system.

In SDL-2010, a process does not have to be encapsulated in a block to fit into the diagrams.

It is possible to have a block agent that contains alternative internal structures of contained agents and channels, with each alternative selected by (mutually exclusive) option areas and typically one alternative directly containing process agents and the other alternative directly containing further block agents. While it may be useful to use this feature during Draft Design, it is not recommended to use it for Formalization.

Recursive application of step S:6 results in a number of levels. Even if the system is complex, there should not be more than three levels of blocks. If each level has three to four blocks with two to five processes in each leaf block, then there will be over 100 processes in the whole SDL-2010 system.

Enclosed block agents are drawn as block references (directly nested diagrams are not allowed in SDL-2010).

If there are more than five obvious process definitions within a block, then the block should probably be divided into two or more blocks with fewer processes. The interactions in Message Sequence Chart diagrams between axes corresponding to processes can help to identify natural "clusters" of processes for each unpartitioned block.

An agent tree diagram should be drawn.

Rule 24 – The diagrams are be nested by reference.

Rule 25 – The number of channels from each block should be no more than five.

Result: The result is a set of agent diagrams and an agent tree that has the system as the root and blocks that are not further partitioned into blocks for each leaf.

Example

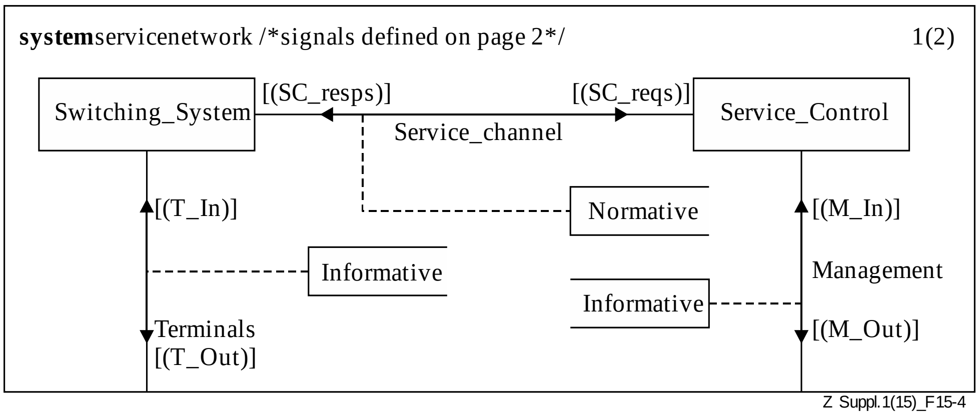


Figure 15-4 – A model for a network to handle services

15.1.7 Step S:7 – Block constituents

Instructions

1) Identify the separate objects with behaviour for the agent that has been chosen in step S:6 to be divided into processes (or agents with explicit state machines), and define these objects as the processes (or agents with explicit state machines) of the enclosing agent.
2) Find a suitable name for each process (or enclosed agent with an explicit state machine) and describe it and its relation to its environment (the enclosing agent) informally in a comment within the enclosed agent reference.
3) For each agent, define its initial and maximum number of instances.
4) Use channels to connect the process sets to channels at the block boundary.

Guidelines

The channels within the enclosing agent can be derived in a similar way to the channels in steps S:3 and S:4.

The methodology sometimes results in a single process encapsulated in a "dummy" block: the block containing a single process description of just one type and the relation to its environment is derived from the external interface of the block. In this case, the "dummy" enclosing block can be removed and the process directly placed where the "dummy" block was.

Consider object models with associations between classes corresponding to agents that have an explicit state machine (usually processes, but also may be blocks with state machines). The respective multiplicity (1:1, 1:many, many:many) of the association helps to define the initial and maximum number of instances for those agents. If the number of instances should not vary the minimum number of instances should be given (the same value as the maximum and initial number of instances).

Rule 26 – For each agent, at least one enclosed agent with an explicit state machine shall have its initial number of instances greater than zero, so that it can create other instances in the agent.

Rule 27 – The number of agent definitions within each enclosing agent should be no more than five.

Result: Identification of agents as leaves in the block tree.

15.1.8 Step S:8 – Local signals in a block

Instructions

1) Identify the signals between the agents local to the enclosing agent.

2) In the enclosing agent (usually a block) define any of these signals that are additional (not defined external to the enclosing agent).

3) Identify any imported and exported procedure belonging to agents (usually processes) in the enclosing agent (usually a block), and the corresponding remote procedure definition, and define these in the enclosing agent (or agent type).

Guidelines

The relevant description is in the internal interface aspects of the class corresponding to the enclosing agent (usually a block). If the channels are described as part objects, information about the signal on the channel is given by the corresponding local channel in the classified information. In the special case with a "dummy" block (a block containing a single process), there will be no additional signals at the block level.

SDL-2010 allows three ways (other than signal passing) that the value in data variables in one process can be used in another process: calling a remote procedure in the other process, import and export, and accessing data variables defined in an enclosing agent. All of these mechanisms are open to the possibility of designing a system that contains inherent errors. They must be used with care, and simulation of the interaction is recommended. Remote procedures are permitted. Import and export should be replaced by remote procedure calls. Sharing data variables in an enclosing block is a shorthand for accessing remote get and set procedures in the state machine of the block, and it is clearer to show these procedures explicitly. Sharing data variables in an enclosing process does not need remote procedures, because interpretation of the state machine is interleaved with other state machines.

Rule 28 – Remote procedures should be used for data import/export, and for sharing data in enclosing blocks.

Result: The result is a definition of each signal and remote procedure definition at the block (or block type) level.

15.2 Behaviour steps (B-steps)

These steps are to describe the way components behave in terms of communication, but without providing a formal definition of the data covered by data steps. This subclause describes these steps for an SDL-2010 **process** agent, but both these steps and the data steps are also generally appropriate for defining the way an SDL-2010 **block** agent (with explicit state machine) or **procedure** behaves.

15.2.1 Step B:1 – Set of signals to a process agent

Instructions

Identify the input alphabet of the process agent, called the signalset of the process, which is done by identifying any:

– signal that can be received by the process agent where the process may or may not send a response;

– exported procedure of the process agent. This is a case where the process acts as a server in a client-server model where the corresponding signal is implicit but the remote procedure name can be considered as part of the alphabet.

Guidelines

Although the signalset can usually be simply derived from the enclosing agent (usually block) diagram, the purpose of this step is to review the signalset considering only the current process. If it is decided to change the set of signals from other processes, the corresponding agent (usually block) diagrams must be changed. The processes that send the signals will need to be updated (probably at some other time), if they have already been formalized. One source of signals not necessarily shown on the block diagram is the process sending signals to itself or other instances of the same process definition (that is, the same agent instance set).

The signalset is used when deciding the behaviour of the process in each state in step B:4. It is suggested that a record is kept of the signalset if this cannot be derived automatically from tools used for SDL-2010.

For a procedure, the signals of the enclosing process are used, but new signals may be identified that need to be added to the process signalset (identified in this step but for the enclosing process).

Result: The result is the definition of the signalset and set of exported procedures of each process.

15.2.2 Step B:2 – Skeleton processes

Instructions

1) Produce Message Sequence Chart diagrams for at least the "typical" use cases if these have not been produced as draft designs.

2) Produce a skeleton process by mapping from the Message Sequence Chart diagrams considering only "typical" uses:

   2.1 Starting from the start symbol of the process, build a tree of states by considering "normal" state changes of the process.

   2.2 Build the process tree by branching at each state based on each input that is consumed but not ignored in the Message Sequence Chart diagrams and following each by a transition (including outputs and creates) to different states.

   2.3 Identify a state as different from other states if it has a different set of signals that it consumes or saves, or if it has a different response to a signal.

   2.4 Include time supervision (**set** and **reset**) and the corresponding timer input.

   2.5 As the tree is drawn, identify where the process returns to the same state and make the tree into a graph.

3) Draw this graph as a process diagram.

4) Determine whether the process has two or more disjoint sets of interfaces for different behaviour parts of the process that can be interleaved, then if the behaviours are independent, divide the process into multiple processes.

NOTE – If the behaviours are coupled by the use of common data, dividing the process into processes requires one or more remote procedures to access the data, or enclosing the divided process in an enclosing process containing the shared data. It is also possible to divide such a process using an SDL-2010 composite state aggregation.

Guidelines

The Message Sequence Chart diagrams are produced as part of Draft Design.

Message Sequence Chart diagrams can lead semi-automatically to a skeleton process. The order of events on the vertical axis in the Message Sequence Chart diagrams gives one ordering of events in a process for which a skeleton can be produced. In practice this is done by taking typical use cases in the Message Sequence Chart diagrams and then writing the process definition from these use cases (the "simple and usual cases"). Dynamic process creation can also be deduced from the Message Sequence Chart diagrams.

Time supervision can be shown on Message Sequence Chart diagrams. It is used to model a time span, to supervise the release of a resource and to supervise replies from unreliable sources.

Sometimes the Message Sequence Chart diagrams show a message sent to a process, requiring a response that is based on information not accessible to the process or in the message. If the requesting process has the information, then usually the solution is to ensure it is sent in a signal parameter (and to update definitions as appropriate). This can be in the message requiring the response or an earlier related message. If the information is in another process or external to the system, then the responding process must communicate with the owner of the information by signals or a remote procedure. Update the Message Sequence Chart diagrams or annotate them as incomplete and simplified. When a remote procedure is used, consider which timer should be used (if any) at each procedure call.

A skeleton process is expected to exhibit the essential action of a process, but not the complete set of actions. Other draft designs (and the Message Sequence Chart diagrams) should be compared with the skeleton process to check the consistency of the process with these other designs.

Two processes with N and M states respectively, when combined as one process has N × M states. Splitting such a process therefore produces two much simpler processes that are easier to understand.

When a process (or procedure) is informative, there may be spontaneous transitions starting with **none** that model user behaviour or other unpredictable events.

Rule 29 – Spontaneous transition should not be used in normative parts of the standard.

Rule 30 – The Message Sequence Chart diagrams either shall be correct traces of the handling of messages by the SDL-2010 system, or shall be clearly annotated how and why they differ from the way the SDL-2010 behaves.

Result: The result is a skeleton process in the form of a state-overview diagram including timing, and consistent typical Message Sequence Chart diagrams.

Example

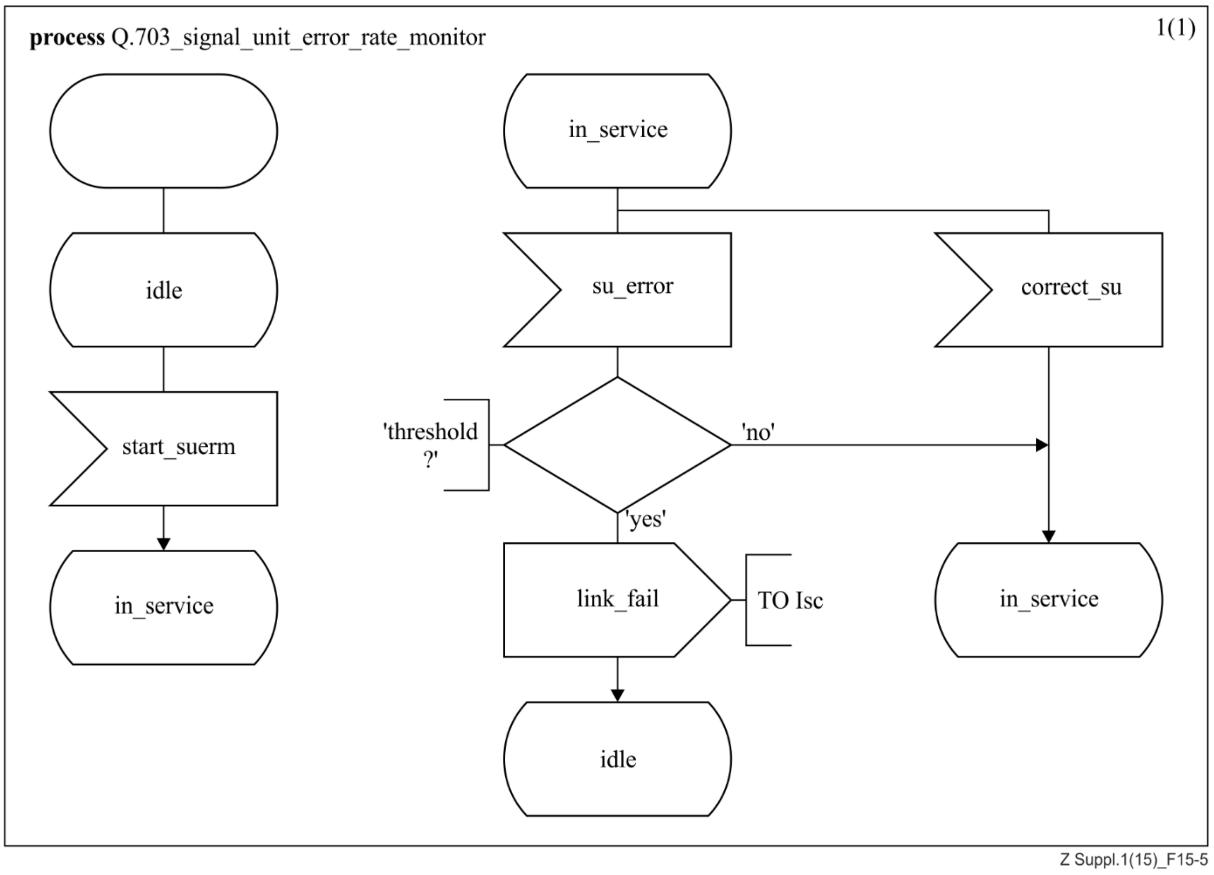


Figure 15-5 – A skeleton process

15.2.3 Step B:3 – Informal processes

Instructions

1) Identify combinations of uses.
2) Identify what information the process stores and consider whether this is implicit in the process states or whether internal data is needed.
3) Use this information to define the internal actions of each process.
4) Add tasks or procedures and possibly more decisions in transitions, but use only informal text in tasks and decisions.
5) If a procedure is used, give it a meaningful name and (later) define it with steps B:1 to D:8.

Guidelines

Use existing Message Sequence Chart diagrams and generate new Message Sequence Chart diagrams to identify combinations of uses.

Decide whether to use a task or procedure to describe the information processing in a transition, although this may be changed later. Use a procedure if it is anticipated that the information is needed from another process, or if the task is likely to be complex, or to depend on several stored data values; otherwise, choose a task. If a procedure is chosen, it may be appropriate to consider the parameters (number and sort).

Sometimes it is obvious that the same actions are done in several places in the process. These actions can be collected and made into a procedure.

Result: An informal process that covers all the cases in the Message Sequence Chart diagrams.

Example

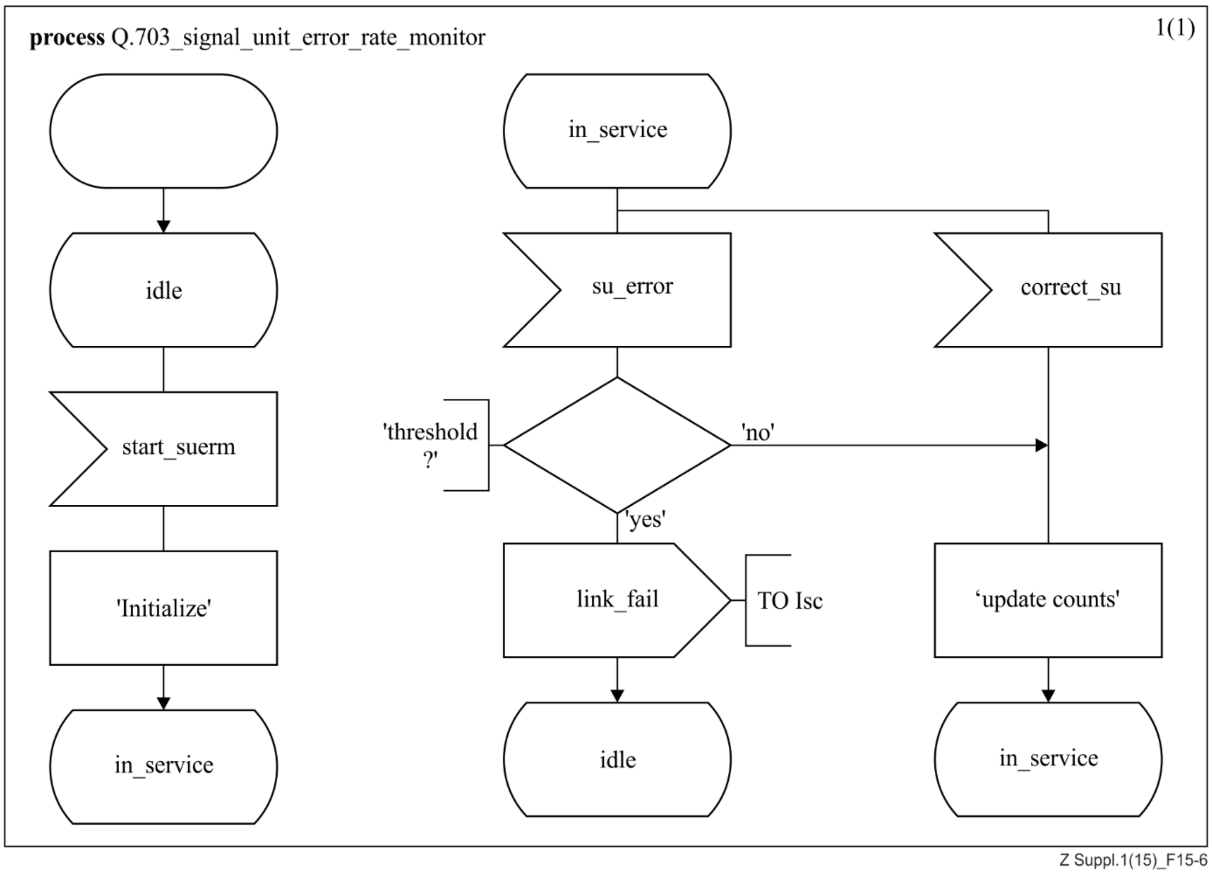


Figure 15-6 – The informal version of the same example as step B:2

15.2.4 Step B:4 – Complete processes

Instructions

1) Identify at each state and for each item in the signalset (signal or remote procedure) whether the signal (or remote procedure call) is input by the process or saved.

2) If the item is input, determine the transition taken (it may be an implicit transition back to the same state).

3) Continue instruction 1 and instruction 2 until there are no states at the end of a transition that have not been considered so that all items in the signalset have been considered for every state.

4) Analyse first each process and then combinations up to the whole SDL-2010 system to check for unwanted properties, and redesign to avoid them if necessary.

Guidelines

A state-signal matrix (see clause 14.6.3) can be used to check the action for each signal in each state. As a new state is identified, the matrix is extended. This matrix can also help identify when two of the defined states lead to the same behaviour and can be combined, or two signals have the same next states. In these cases it may be possible to reduce the number of signals or states.

Sometimes the same signal or the same interface occurs on different channels leading to the process. If the handling of these signals is different depending on which channel the signal arrived, a **via**-path is used in the signal input to distinguish the channel (or gate) leading to a particular transition.

A state-overview diagram (see clause 14.6.2) can be drawn to get an overview of the behaviour of the process. If the process is not normally intended to terminate, then usually every state should be reachable from every other state, but this need not be the case as there may be initial states for the start-up of the process and final states for termination.

There is a limit to how much investigation is practical on a single process. More interesting results from investigation are obtained as more processes are "complete" and can be analysed in combination in a block or as the whole system. The unwanted properties that can be detected (such as unreachable states, deadlock and live-lock) will depend on the complexity of the system and the tools available.

For investigation it can be assumed that every decision contains the SDL-2010 **any** value construct.

Investigation of anything other than a simple process is difficult without tools to generate the state space and then check it. Even tools have difficulty with large numbers of processes or a very complex process. This is therefore not a trivial step, but investigation at this stage saves a lot of wasted time and effort if redesign is found necessary, and is one of the major benefits of using SDL-2010.

Rule 31 – All states in a process shall be reachable from the start of the process.

Rule 32 – A procedure that is exported by a process (to be used as a remote procedure) shall not be saved in every state of that process.

Rule 33 – Each signal received by the process should have at least one input leading to a non-empty transition.

Result: At this stage the process (procedure or composite state) definition is complete but informal.

Example

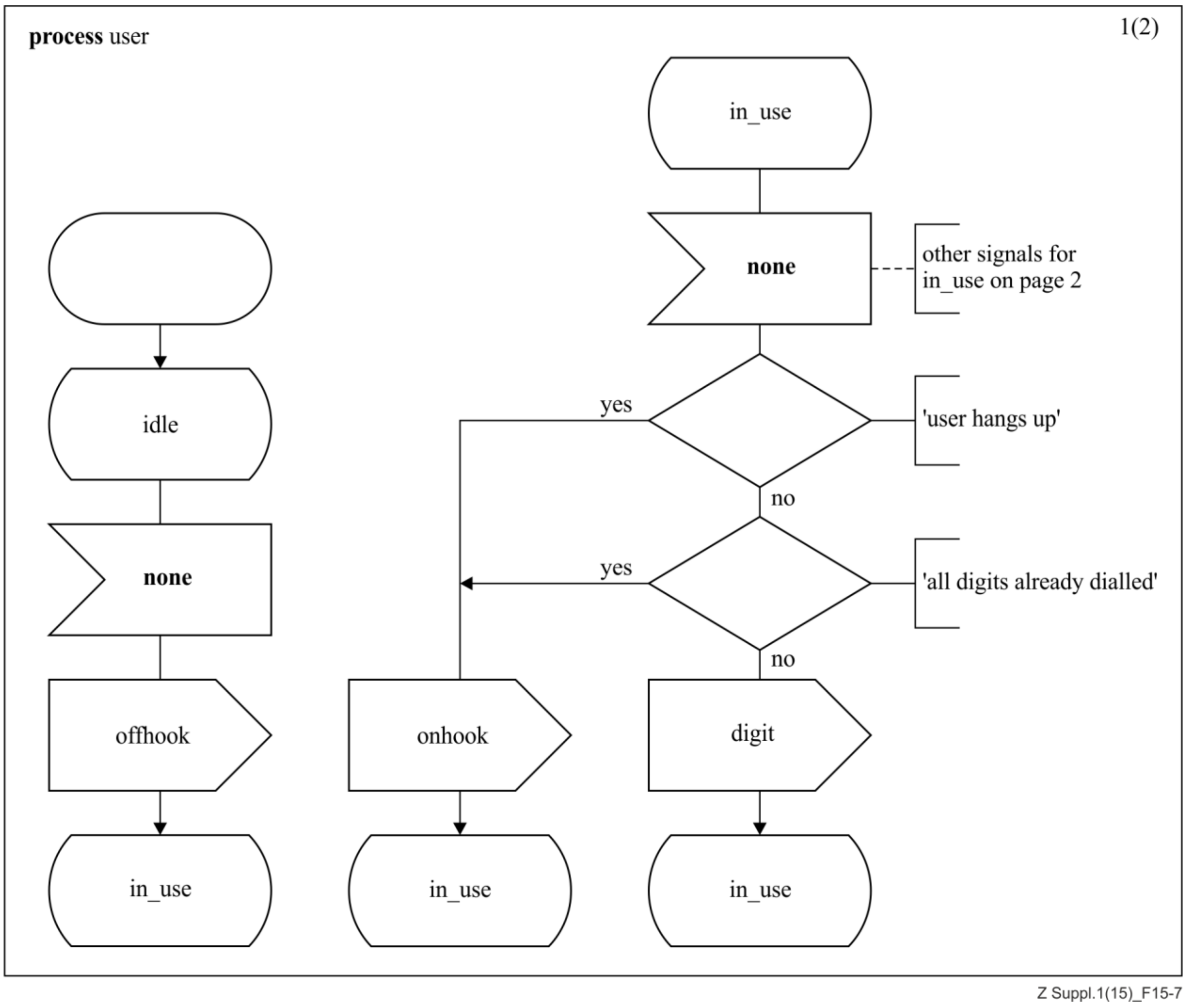


Figure 15-7 – Part of a complete but informal process

15.3 Data steps (D-steps)

The purpose of the data steps is to provide a formal definition of the data used in the agents. Without formal data any decision in transitions and operators used in expressions have uncertain results.

The first two steps (D:1 to D:2) concern interface values. The next three steps (D:3 to D:5) concern variables within agents. The remaining steps (D:6 to D:8) concern the formal definition of new sorts of data. These last steps are needed if new sorts of data have new operators. For this reason steps D:6 to D:8 are only occasionally necessary, and should be unnecessary if ASN.1 has been used to define data. The main advantage of ASN.1 is standardized encoding. Alternatives to ASN.1 are the native data concrete syntax defined for SDL-2010 in [b-ITU-T Z.104], or the data binding to a subset of [b-ISO 9899] (C Programming Language) concrete syntax defined in [b-ITU-T Z.104C]. Each of the data concrete syntaxes (ASN.1, native SDL-2010, or supported C data language subset) is mapped to a well-defined abstract grammar for data that defines the semantics of the language.

Step D:1 is applied to the system as a whole, whereas steps D:2 to D:8 can be applied to each agent with a state machine, state diagram or procedure.

Rule 34 – The sorts of data used shall preferably be defined using SDL-2010 combined with ASN.1 as defined in [b-ITU-T Z.105], alternatively using native SDL-2010 data [b-ITU-T Z.104] or the supported C data language subset [b-ITU-T Z.104C].

15.3.1 Step D:1 – Signal parameters

Instructions

1) Identify the values to be conveyed by signals, starting with signals at the system level.
2) Look for predefined sorts of data to represent the identified values.
3) Extend the signal definitions with the sort of data.
4) Identify and define new sorts of data if necessary.

Guidelines

The classified information description of the intended communication provides a source for the names of sorts of data.

ASN.1 provides a formal description of the data used for signals, and therefore the corresponding sort of data can be used directly. If ASN.1 has not been used, a decision must be made whether to define the sorts of data in ASN.1 or SDL-2010. ASN.1 should be used if the encoding of data is important or if the data is on a normative interface with the environment. SDL-2010 may be used if no particular encoding is required and if the sort of data is used only within the SDL-2010 system.

Rule 35 – If bit tables have been used to define data, these shall be converted to ASN.1 or native SDL-2010 data or the supported C data language subset.

Rule 36 – The names of existing sorts of data should not be used for new sorts of data even if they have different scopes.

NOTE – If the sorts of data have the same scope, SDL-2010 language rules do not allow the names to be the same.

Result: The signals have parameters with corresponding defined sorts of data.

Example

From Figure 15-5 the signal "digit" can be extended to "digit(digit)", where the sort of data is defined as shown in Figure 15-1.

15.3.2 Step D:2 – Process and procedure parameters

Instructions

1) Identify the sorts of data needed for parameters of agents (or procedures or composite states).
2) State the role of each parameter in a comment in the heading of the agent (or procedure or composite state).

Guidelines

Within an agent, a parameter is treated as a variable. The only difference between an agent parameter and a variable with a default value is that an agent parameter can receive a different value for each instance of the agent. Typically this value is the identity of some other entity such as a Pid or an equipment number.

The classified information and draft designs concerning creation and deletion give guidance on the role of the parameters to pass to a process when it is created. The invocation of a procedure can also correspond to creation (and the procedure return to deletion).

*Rule 37 – An agent parameter should not contain the parent Pid as this is available as the **parent**.*

Result: The agent (or procedure or composite state) parameters have been formalized.

15.3.3 Step D:3 – Input parameters

Instructions

1) Add parameters to inputs according to signal definitions.
2) Define variables as required to receive the input values.
3) State the role of each variable in a comment.

Guidelines

Check that each parameter defined in a signal definition is used in at least one input or is required for communication with the environment.

Rule 38 – A signal parameter should not contain the sender Pid as this is available as the ***sender****.*

Result: The input interfaces have been formalized.

15.3.4 Step D:4 – Formal transitions

Instructions

1) Replace the informal text in tasks, decisions and answers with formal assignments, formal expressions, formal range expressions and procedure calls, and identify any new operators used in the formal expressions to be added to the sorts of data in step D:6.
2) Define additional variables and synonyms as required.
3) Define additional procedures as required.
4) Add parameters to procedure calls (and composite state invocations) according to procedure (and composite state) definitions.

Guidelines

The previous informal text in symbols may be useful as comments attached to the symbols.

If the value of a function used in an expression depends only on the actual parameters, there is a choice of making the function an operator or a procedure. If the result depends on other data, it must be a procedure. If the function behaviour depends on data from another process, it may be appropriate to make it a remote procedure, or decide how this information is obtained. To obtain the information may require additional parameters to existing signals and storing this information, or communication in the procedure, possibly with additional signals.

The definition of a procedure (or a state for a composite state) is treated in a similar way to an agent through steps B:1 to D:8, including the possibilities of having a procedure called (or composite state invoked) internally and a procedure (or composite state) nested within the procedure (or composite state).

When the normative behaviour is independent of some data in informative agents or procedures, the **any** construct can be used to create random values for this data or random decisions in informative agents or procedures. In this way the informative parts can model situations where data and behaviour are unpredictable. Spontaneous transitions are used if the time ("when") something happens is unpredictable (see clause 15.2.2), and **any** value is used if what happens is unpredictable.

Rule 39 – A value or decision that is non-deterministic (using ***any****) shall not appear in a normative part of the system unless it is explicitly marked informative.*

Rule 40 – A value or decision that is non-deterministic (using ***any****) shall always have a comment attached explaining how the choice is made.*

Result: At the end of this step, the informal text has been eliminated.

Example

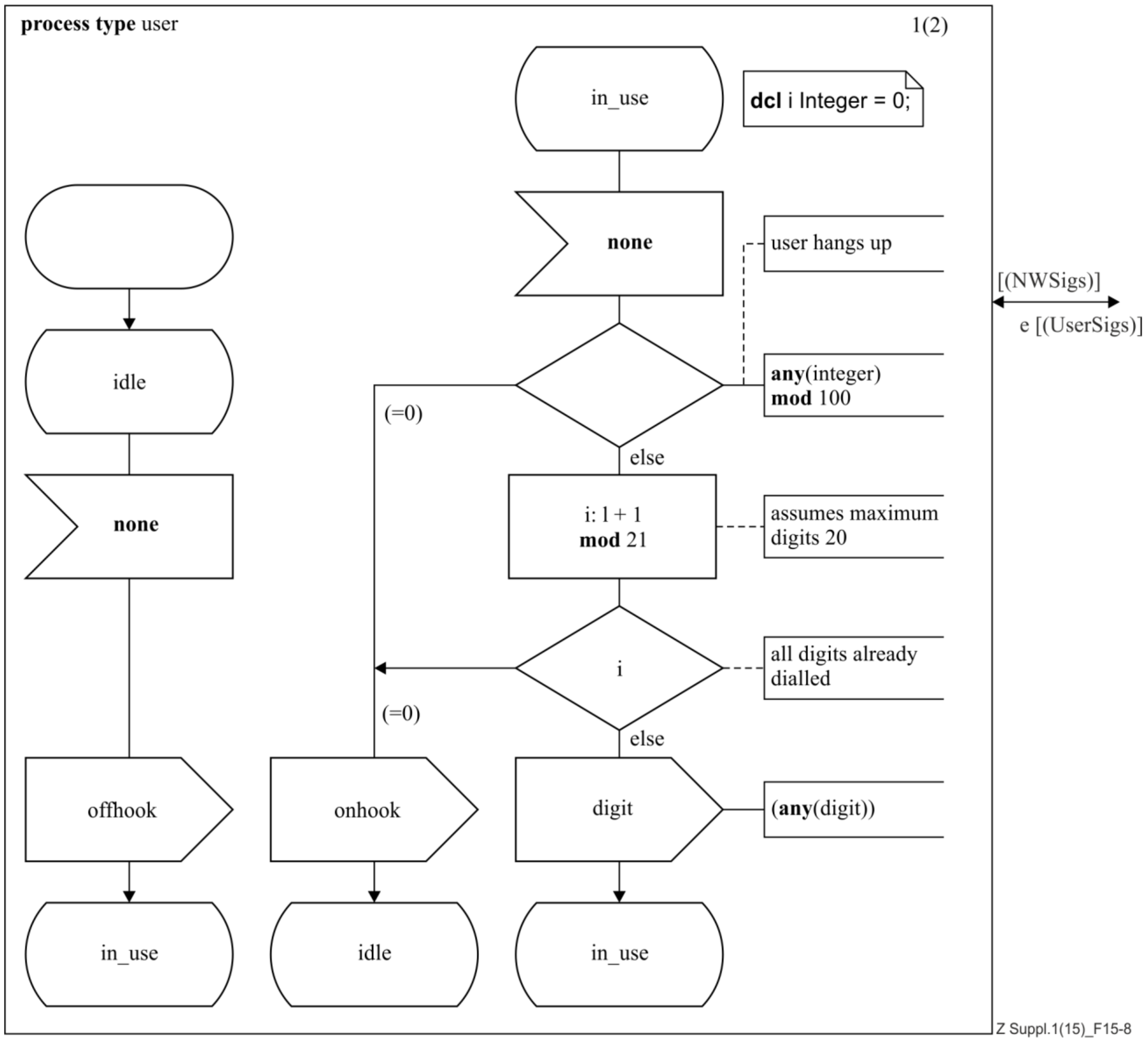


Figure 15-8 – Part of a complete formal process agent for a user

This example makes the user process in Figure 15-5 formal and also illustrates the use of the **any** construct.

15.3.5 Step D:5 – Output and create parameters

Instructions

1) Add expressions to outputs using introduced variables and synonyms.

2) Add actual parameters to create actions.

Guidelines

Check that each parameter sent in an output is used in at least one possible input or is required for communication with the environment. As compared with the check that a signal definition parameter is used in step D:3, this is a check that a signal instance parameter may be used.

Rule 41 – If a parameter is omitted in an output; there shall not be a corresponding input that expects a value for this parameter.

Result: The SDL-2010 description is now formal, except for the behaviour of expressions, which depend on data types.

15.3.6 Step D:6 – Data signatures

Instructions

1) Identify and define all the sorts of data and values (synonyms) that need to be defined.

2) If some values are identified as dependent on the actual system installation, express them by using external synonyms.

3) For each required sort of data define a data type or syntype with a meaningful name or define an ASN.1 type.

NOTE – Data types are defined using either **value type** { … } or **newtype** … **endnewtype**; syntax.

4) Identify any new operators that need to be defined.

NOTE – If there are no new operators, the formalization process is complete.

5) For the data types with new operators, list the signatures (the literals and the operators with parameter sorts) and identify (in a comment) a set of operators and literals that can be used to represent all possible values (the "constructors").

Guidelines

External synonyms can be used to:

– provide dimensions for the number of agent instances, the number of data items in an array, etc.;

– make choices in transition options or selects providing optional functionality and alternative behaviour.

Inherit as much as possible from [b-ITU-T Z.104] **package** Predefined data, which includes ASN.1 "useful types" with appropriate operators. The objective is to avoid having to define the behaviour for new operators. If new operators are defined, they should be defined using operation definitions (textually in an operation definition or graphically in an operation diagram).

The introduction of new operators can be avoided by:

– using the operators defined in [b-ITU-T Z.104] **package** Predefined;

– using a literal data type constructor (or ASN.1 ENUMERATED) for any enumerated type;

– using **struct** (or ASN.1 SEQUENCE) for records;

– using **choice** (or ASN.1 CHOICE) for any data type with mutually exclusive alternatives;

– using the **package** Predefined parameterized data types:

  – Array – elements of one sort index by another;

  – Vector – elements of one sort index by an Integer in the range 1 to a specified maximum;

  – String – ordered sequence of elements of the same sort with string operators;

  – Powerset – a mathematical set with elements of one specified sort (no value repeated);

  – Bag – an unordered collection with elements all of one sort (any number of each value);

– using a **syntype** if the set of data values is intended to be subset of and compatible with a sort of data;

– giving a sort a new name with **syntype** if the purpose is to introduce a more meaningful name;

– giving a sort a new name with **newtype** with **inherits all** if the purpose is to have a sort of data with the same properties as an existing sort of data, but not compatible with the existing sort, so that values of one sort cannot be used where values of the other sort are needed.

The operators are usually listed in the data type definition for the sort of data corresponding to the result of the operator. Sometimes an operator produces a value of a sort of data defined in a different context, for example, an Integer or Boolean. In this case the operator is listed under the data type of (one of) the parameter sorts of data.

A set of operators and literals that can be used to represent all possible values is a set of constructors of the sort of data. These constructors are used in later steps if (and only if) it is necessary to define operator properties. The set of constructors is not usually unique, nor is it always obvious. The chosen set should be just sufficient to define all values so that if one constructor is deleted then the set of values is different. The choice can be difficult!

Although this step produces a formal SDL-2010 description, if new operators are introduced, the functionality is not formally defined until the operators are defined. Step D:7 corrects this deficiency, but makes interpretation depend on informal text. Step D:8 makes the description formal.

Result: At this stage the system is completely formally defined and is complete if no new operators have been introduced.

15.3.7 Step D:7 – Informal data description

Instructions

Add informal descriptions of any new operators in the form of informal text in the data type definitions or in dummy operation definitions.

Guidelines

The names of the operators should correspond to their function and therefore assist the description of the function.

Static properties (ignoring operator results) are analysed for correctness. For example, it can be analysed (usually by the tool being used) whether operator parameters are the correct data type. The dynamic properties depend on the interpretation of the operator body, which can only be done informally. However, in an actual support environment, some additional features or assumptions may make this level of description sufficient.

Result: The informal descriptions provide a record of what is intended by the data type definition, without formally defining operators.

15.3.8 Step D:8 – Formal data description

Instructions

For each operator signature, add an algorithmic operator definition in the form of an operation definition or operation diagram.

Guidelines

An operation definition is textual and concise. An operation diagram may be easier to understand for complex operators.

An alternative is to invoke an external operation (implemented in some other language), but this has the disadvantage that the SDL-2010 is not then completely formally defined.

Rule 42 – The use of external operator definitions should be avoided.

Result: If all the above steps have been followed completely, at this stage there will be a completely formal specification in SDL-2010, including complete and unambiguous definitions for all the sorts of data used in the specification.

15.4 Type steps (T-steps)

The type steps are used to define the pieces of the system as types so that they can be reused.

To get the maximum benefit from the application of these steps, in larger systems application of these steps in parallel with the preceding steps is recommended. This is because as one branch of the system hierarchy is developed, it can lead to types that may be reused in another branch.

The most important steps are T:1 and T:2. Steps T:3 to T:5 may be considered optional. A good knowledge of SDL-2010 and object orientation is advisable before using steps T:3 to T:5. In some cases, parameterization as covered by step L:2 can be used instead of specialization.

15.4.1 Step T:1 – Identification of SDL-2010 types

Instructions

1) Modify instance definitions used in the system to use types.
2) Where two instances (for example, blocks) have the same behaviour, use the same type for both instances.

Guidelines

This allows the types to be clearly recognized and can allow some unnecessary repetition in the SDL-2010 system to be removed. A typical situation is where an "end-to-end" system is defined and the termination at each end is described by agents with the same behaviour. The definition must be repeated for both agents unless an agent type is used, in which case the definition of the agent type can be used for both agents.

The definition of a system type is optional, but allows the system definition to be reused as the basis of other similar standards. This can be particularly useful if there is a set of related standards.

Adding block or process agent types requires gates to be added to identify how the connection to the definition using the type corresponds to the communication paths of the type.

A process or block agent that is only used once should initially be modelled as a type because it may be possible to replace this by a type based on an existing type at a later step. Although it is easier to initially produce diagrams without using types, the introduction of types allows reuse that can then save a lot of repeated description and can save engineering effort.

It should be noted that all procedure and signal definitions are type definitions.

Rule 43 – Two blocks (or processes) that model instances of the same thing shall be modelled by block (process) definitions based on a common block (process) type definition.

Result: Defined types for the system, blocks, and processes.

Example

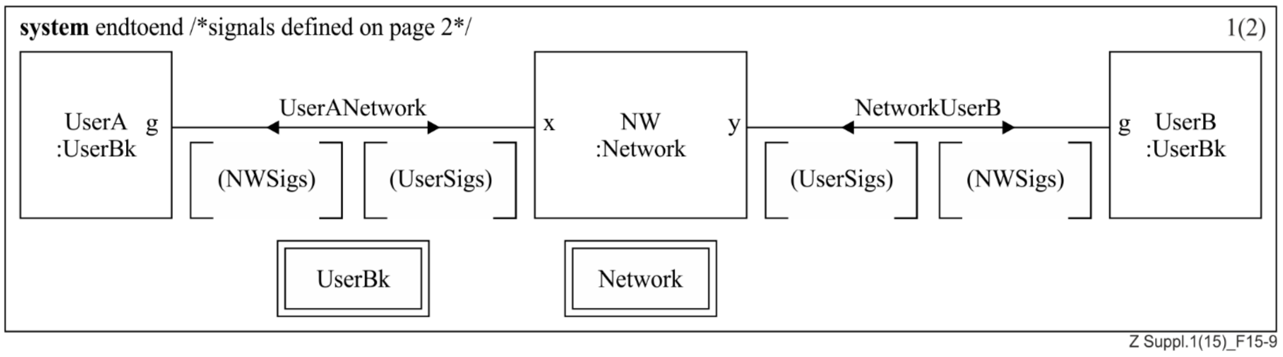


Figure 15-9 – A system definition using types

15.4.2 Step T:2 – Specialization of types by adding properties

Instructions

1) Identify cases where one type has all the behaviour of another type when presented with any stimulus (signal or remote procedure call) that this second type might receive, but also handles additional stimuli.

2) Define the first type as a subtype that **inherits** the properties of the second type and just **adding** the properties to handle the additional stimuli.

Guidelines

A subtype is a type created by specializing another type (called the supertype of the subtype). Specialization can be done in two ways: adding properties to the supertype or changing some of the properties of the supertype in the subtype. This step considers adding properties to the supertype.

The properties that can be added are limited only by the properties already defined for the type so that the added properties do not conflict with the inherited definition. Channels, block (type) definitions, process (type) definitions, procedures, state types and signal definitions can be added to a block (or system) type. All of these except block (type) definitions can be added to a process type. New transitions for new signals can be added to the state machine of an agent type and the transitions can lead to new states. New signals to/from the specialized type should be added to gates (existing or new) of the agent type.

Result: A new subtype with extended behaviour, but which is compatible with the previous behaviour.

Example

Two processes (suerm and suerma) are identical except that process (suerma) accepts an extra signal (abort_call). When abort_call is input it resets the process to the idle state regardless of the previous state of the process. The process type (SUerma) for the second process can be a subtype of the type (SUerm) the first process (see Figure 15-10).

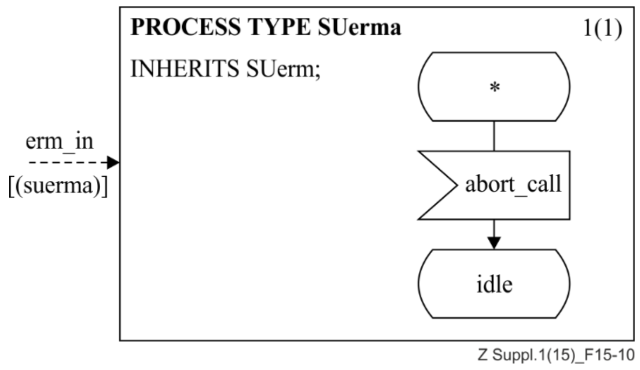


Figure 15-10 – A subtype created by adding a property

15.4.3 Step T:3 – Using "virtual" to generalize types

Instructions

1) The step T:3 is complete if there are no types that have almost the same internal structure and behaviour and handle slightly different situations (they are "similar").

2) Otherwise for each set of "similar" types (as in instruction 1), apply instructions 3 to 6.

3) Identify the common part and specify this as a general supertype to be used for the other types that become subtypes when step T:5 is applied.

4) If the general supertype is an agent type without an explicit state machine, identify the instances (blocks and processes) in the subtypes that have to be different, make all these type-based instances and define the types as virtual types in the general supertype.

5) If the general supertype is an agent type with an explicit state machine (or the general supertype is procedure or state type), then identify the partial action sequences that should be adaptable, and:

   5.1 if they are complete transitions, make them virtual transitions;

   5.2 otherwise, replace the partial action sequences by procedure calls and define corresponding virtual procedures;

   5.3 if there are other procedures in the general supertype, consider whether they should be virtual.

6) For each virtual type in the general supertype, identify or define a type that has the internal structure and parameters appropriate to all redefinitions and use this type as a constraint on the virtual type as elaborated in step T:4.

NOTE – The supertype defined by this step is not used within the SDL-2010 system until other steps are applied.

Guidelines

A general supertype is a supertype that has some parts that can be redefined. These parts are identified as being virtual. If the type is used without redefining these parts, then the definitions in the virtual parts apply. Therefore the definitions of the virtual parts in the general type are chosen so that they model the most common variation.

Be careful not to overgeneralize types. There is also a danger in generalizing when two types have similar functional behaviour, but very different concepts, because it can be difficult to understand the purpose of the general supertype as applied to each case. The general supertype should capture the essential features of a concept as behaviour that is not virtual.

A virtual transition allows the whole transition to be redefined. Consider making parts of the transition into virtual procedures, so that the common parts are fixed and the whole transition does not have to be virtual.

Rule 44 – The common part of a supertype should constitute at least 70% of the total of each subtype.

Result: A type generalized by the definition of some of the local types or transitions as virtual.

Examples

A typical case is two processes based on types that are identical except for the transition for one signal in one state. The general supertype is then a **process type** definition in which either the transition that differs is virtual or the transition that differs contains a call of one or more virtual procedures. Usually the virtual transition and/or procedure(s) define how one process behaves, and that process is based on the general supertype containing the virtual item(s). How the other process behaves is then based on a specialized supertype that inherits the general supertype and redefines the virtual item(s).

For another example, assume that a **system** "AtoB" that represents communication from UserA to UserB through a network NW as in Figure 15-11. Normally in such a model both ends would be the same, but for the example UserA and UserB are almost identical blocks each containing one process called User that behaves slightly differently in each case. When a UserA receives a ring signal it enters a ringing state, whereas a UserB enters the connected state. There is also extra initialization for UserB. The signals are defined on page 2 of the system.

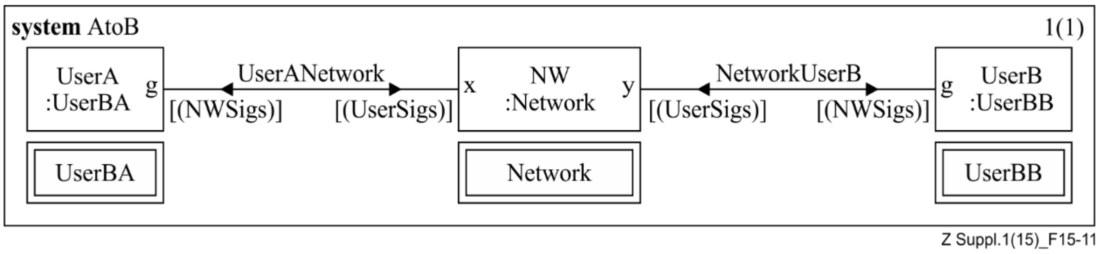


Figure 15-11 – The AtoB system

The **process** User is slightly different in each case. The general supertype containing the **process** User is defined as a **block type** UserBk as shown in Figure 15-12 where the **process** User is based on the **virtual process type** Usertype.

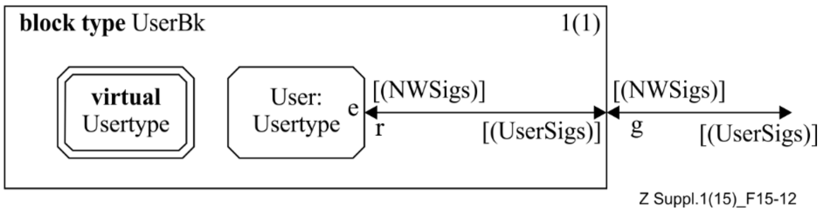


Figure 15-12 – The general super block type UserBk for the "User"

Since there is only one process in each block and this has different behaviour in each case, this process is based on the **virtual process type** Usertype. The **block type** UserBA and **block type** UserBB can be defined as inheriting the general supertype UserBk and redefining (if necessary) the **virtual process type** Usertype.

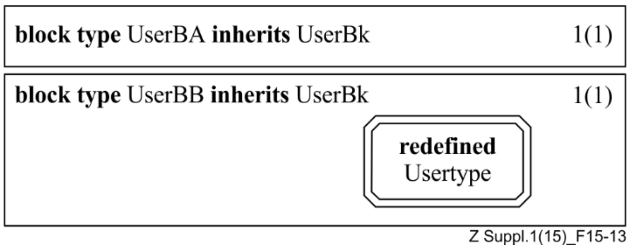

block type UserBA inherits UserBk
1(1)
block type UserBB inherits UserBk
1(1)
redefined Usertype
Z Suppl.1(15)_F15-13

Figure 15-13 – Definition of UserBA and UserBB using UserBk

UserBA simply inherits UserBk so it would be possible to just use UserBk instead of UserBA.

15.4.4 Step T:4 – Constraining virtual types

Instructions

1) Decide which properties of a virtual type must apply for all redefinitions of that virtual type.
2) Define (or identify) a type which has these properties (called the "constraint type").
3) Add **atleast** "constraint type" to the definitions of the **virtual** types in the general supertype.
4) Define a **redefined** type (based on the virtual constraint type) for each case where the virtual type will be used.

Guidelines

A general supertype encloses virtual types that can be different each time the general supertype is used to define a subtype. The definition of the virtual type needs to be enclosed in the general supertype, and a definition of a type (corresponding to the virtual type) enclosed in each subtype (of the general type). The constraint type:

– defines the common properties of the virtual type in the general supertype and each of the redefinitions of the virtual type;

– constrains the definitions so that each definition is a subtype of the constraint type and therefore inherits the constraint type.

If all the subtypes of the virtual type need to be subtypes of the virtual type, no constraint type needs to be specified as the constraint for a virtual type V without a constraint specification is V itself.

If the definition of the virtual type enclosed in the subtype is the same as the constraint type, this is the default and the redefinition in the subtype is omitted.

Rule 45 – The constraint type shall fix the properties for:

– *a virtual procedure – the parameters (so that all calls have the same number of parameters of the same type);*

– *a virtual agent type – parameters (so that all creates have the same number of parameters of the same type), gates, common internal agents (including channels to these common agents), common explicit state machine (if there is one);*

– *a virtual composite state type – parameters (so that all invocations have the same number of parameters of the same type), gates (only used for a composite state that defines the state machine of an agent), and common state machine.*

Result: A type definition (the "constraint type") for each virtual type in the general supertype, and a subtype (of the constraint type) for each actual instance (or instance set in the case of agents).

Example

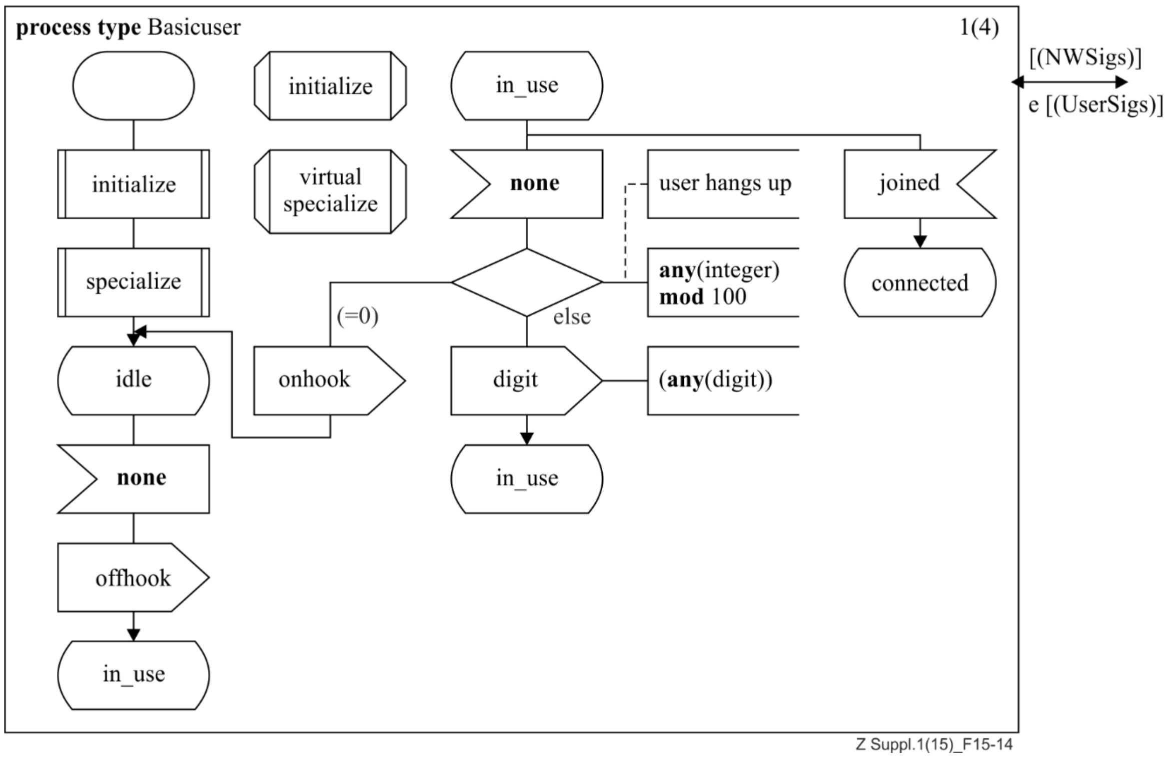
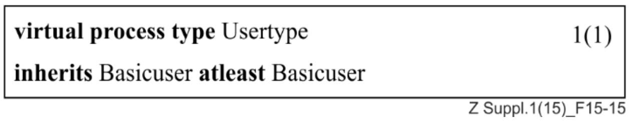


Figure 15-14 – The Basicuser process type

It is assumed that every time **block type** UserBk (see clauses 15.4.1 and 15.4.3) is used, the **process type** Users has at least the properties of a **process type** Basicuser. The definition of the **virtual process**s **type** Usertype in the general super **block type** UserBk is then defined to be **atleast** Basicuser.



virtual process type Usertype 1(1)

inherits Basicuser **atleast** Basicuser

Figure 15-15 – Definition of Usertype in UserBk

NOTE – In Figure 15-15 Usertype is the same as Basicuser, but additional transitions and states could also have been added.

15.4.5 Step T:5 – Specialization by redefining types

Instructions

1) Replace each subtype identified in step T:3 by a type that **inherits** from the general supertype.
2) In the subtype identified in step 1 above, redefine virtual transitions for the context.
3) In the subtype identified in step 1 above, redefine virtual types for the context using the **redefinition**s from step T:4.

Guidelines

The step is necessary to produce a type that describes the required behaviour from the more abstract types generated by previous T steps.

Result: A system based on type definitions.

Examples

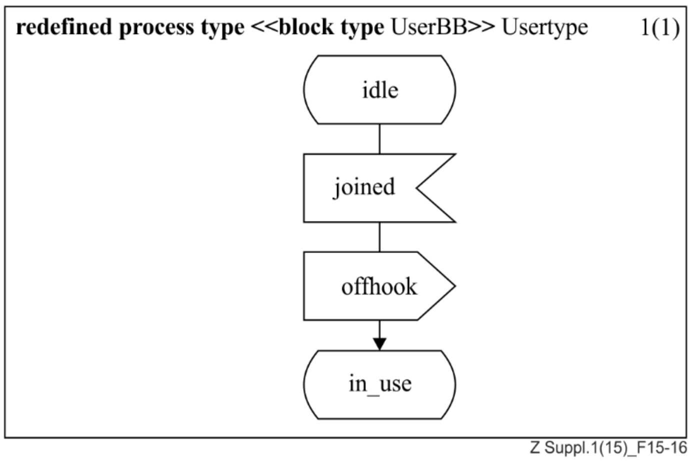


Figure 15-16 – virtual process type Users and a redefinition of Users

NOTE – A qualifier is now needed for the identifier because the name Usertype is reused as a process type in different contexts. A qualifier is also needed for the definition of Usertype in UserBk and (if redefined) in UserBA.

15.5 Localization steps (L-steps)

The purpose of localization steps is to ensure that types are defined at the best places to prevent the unnecessary definition of new types.

15.5.1 Step L:1 – Non-parameterized types

Instructions

1) Find the appropriate scope unit for the definition of each type.
2) Move each type (and related definitions) to this scope unit or (if possible) to a package.
3) Add a **use** for each new package in package reference text symbol of the system diagram.

Guidelines

A type that is not dependent on any definition can be moved to a package. Be careful to identify mutually dependent types or definitions that could be moved to a package together.

At the same time, the scope unit for other definitions can be reviewed. Each definition is placed so that it can be reused in different places but also with the smallest visibility for all its uses. It is a good idea to move the definitions of the signals for the system to a package, because this makes the system more like a block.

A type needs to be local if it depends on definitions or other types in the same scope unit (assuming it is not appropriate and possible to move all the related definitions). A type must be local if its visibility needs to be restricted to the local context. The need to restrict the visibility of a type often occurs because of redefinition of a virtual type with the same name. A type should be local if it is only meaningful in this context.

It is useful to place a collection of definitions in a package so that it can be reused in different systems. In SDL-2010 a package use clause can be associated with any diagram.

SDL-92 (and some tools) only allowed package use clauses to be attached to the system diagram. The use of a package with SDL-92 or some tools therefore conflicts with restricting the local scope of definitions. Localization into packages is considered in more detail in step L:3.

Rule 46 – The intended scope of use of a package shall be added as a comment to the package reference clause.

Result: All definitions have an appropriate local scope or are defined in a package with a comment on their intended scope.

15.5.2 Step L:2 – Defining context parameters

Instructions

1) Identify when of two or more type definitions are identical except for the use of some named items (such as signals or procedures).
2) Determine what constraints (minimum common definition) there are on the types of the named items (for example, a signal may be constrained that it shall have two integer parameters) and identify or define a type to be used in an **atleast** clause.
3) Define a new type definition with the named items as context parameters and **atleast** constraint clauses on the parameters.
4) Replace the uses of the almost identical types by uses of the new type with context parameters, so that the actual parameters are the original named items.
5) Place a reference to the new type definition in the scope unit that includes all the uses.

Guidelines

Context parameters are parameters of types that are replaced by actual parameters that are definitions of items. The replacement is static and takes place before interpretation of the SDL-2010 system. The actual parameters given in the context of the use of the type define a type with the parameters replaced.

The other parameters in SDL-2010 (for example, parameters of processes, procedures and signals) are replaced by the actual parameters that are values; the replacement is dynamic and takes place when the system is interpreted. The values for these parameters are passed to instances of the type.

The use of context parameters is sometimes an alternative to making some parts of a type virtual so that it can be reused. In the case of timer, variable and synonym context parameters, there is no choice: SDL-2010 does not have virtual types for these items. For procedures it is recommended to use a virtual procedure if the actual procedure is always local. It is recommended to use a procedure context parameter if the actual procedure is sometimes a more global one. For an agent (block or process) it is preferable to use an agent context parameter of the enclosing agent, if the agent types for the actual parameters can be usefully defined external to the enclosing agent. System agents cannot be context parameters.

Table 4 – Use of context parameters

	Can be context parameter	**Can have context parameter**	**Can be virtual type**
system	no	yes	yes
block, process (type)	yes	yes	yes
procedure	yes	yes	yes
signal	yes	yes	yes
timer	yes	yes	no
sort (that is, a data type)	yes	yes	yes
variable, synonym	yes	no	no

Sometimes there is no constraint to be placed on the context parameter, in which case the **atleast** clause is omitted.

Rule 47 – Types with context parameters should only be used when an understanding of the system is increased by the reuse of concepts.

Result: A less context-dependent type included in the system, and uses of this type replacing two or more context-dependent definitions.

15.5.3 Step L:3 – Package definition

Instructions

1) Identify the groups of related items that are not specific for the system and that can be separated out into a package suitable for general use.
2) Define a package with the identified items represented by types.
3) Record some information to enable the package to be found during searches for suitable types.
4) Record in the library the use of the package.
5) Whenever a package is reused, record the use in the library, and make a special record of any changes to the package to make it reusable.

Guidelines

Packages are particularly useful where there is a set of SDL-2010 systems that are related, such as descriptions of different supplementary services in different related standards. It is assumed that packages are stored in a reuse library for future use.

In searching the reuse library for definitions, the classified information is used as the basis of search keys to match information in the library. The information (defined names, key words) can also be used to recognize groups of related concepts in the library. To make a collection of useful types for an application domain, it is recommended to use a package rather than enclose these types in (for example) a block type and then specialize the block type for each use. A block type is more appropriate for a functional piece of a system.

Sometimes a package in the library is nearly what is required, but does not quite match the classified information. In this case, the package in the library may be modified to cover the new system. Other uses of the package should be checked before it is modified to avoid introducing incompatibilities for these cases.

Although the handling of a package is not completely defined by the SDL-2010 standard, the support environment is expected to handle it so that it is available for any other package or system specification.

Result: Packages of type definitions for reuse in the library and used in the SDL-2010 system.

Bibliography

[b-ITU-T I.130]	Recommendation ITU-T I.130 (1988), *Method for the characterization of telecommunication services supported by an ISDN and network capabilities of an ISDN*.
[b-ITU-T Q.65 1988]	Recommendation ITU-T Q.65 (1988), *Stage 2 of the method for the characterization of services supported by an ISDN*.
[b-ITU-T Q.65 2000]	Recommendation ITU-T Q.65 (2000), *The unified functional methodology for the characterization of services and network capabilities including alternative object oriented techniques.*
[b-ITU-T Q.1200]	Recommendation ITU-T Q.1200 (1997), General *series Intelligent Network Recommendation structure*.
[b-ITU-T Q.1228]	Recommendation ITU-T Q.1228 (1997), *Interface Recommendation for intelligent network Capability Set 2*.
[b-ITU-T X.200]	Recommendation ITU-T X.200 (1994), *Information Technology – Open Systems Interconnection – Basic Reference Model: The basic model.*
[b-ITU-T X.210]	Recommendation ITU-T X.210 (1993), *Information Technology – Open Systems Interconnection – Basic Reference model: Conventions for the definition of OSI services.*
[b-ITU-T X.219]	Recommendation ITU-T X.219 (1988), *Remote Operations: Model, notation and service definition.*
[b-ITU-T X.290]	Recommendation ITU-T X.290 (1995), *OSI conformance testing methodology and framework for protocol Recommendations for ITU-T applications – General concepts*.
[b-ITU-T X.291]	Recommendation ITU-T X.291 (1995), *OSI conformance testing methodology and framework for protocol Recommendations for ITU-T applications - Abstract test suite specification*.
[b-ITU-T X.680]	Recommendation ITU-T X.680 (2008), *Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation*.
[b-ITU-T X.681]	Recommendation ITU-T X.681 (2008), *Information technology - Abstract Syntax Notation One (ASN.1): Information object specification*.
[b-ITU-T X.682]	Recommendation ITU-T X.682 (2008), *Information technology - Abstract Syntax Notation One (ASN.1): Constraint specification*.
[b-ITU-T X.683]	Recommendation ITU-T X.683 (2008), *Information technology - Abstract Syntax Notation One (ASN.1): Parameterization of ASN.1 specifications*.
[b-ITU-T X.903]	Recommendation ITU-T X.903 (2009), *Information technology – Open distributed processing – Reference model: Architecture*.
[b-ITU-T Z.100]	Recommendation ITU-T Z.100 (2011), *Specification and Description Language – Overview of SDL-2010.*
[b-ITU-T Z.100a]	Recommendation ITU-T Z.100 (1993), *CCITT Specification and Description Language (SDL).*

[b-ITU-T Z.101]	Recommendation ITU-T Z.101 (2011), *Specification and Description Language – Basic SDL-2010*.
[b-ITU-T Z.102]	Recommendation ITU-T Z.102 (2011), *Specification and Description Language – Comprehensive SDL-2010*.
[b-ITU-T Z.103]	Recommendation ITU-T Z.103 (2011), *Specification and Description Language – Shorthand notation and annotation in SDL-2010*.
[b-ITU-T Z.104]	Recommendation ITU-T Z.104 (2011), *Specification and Description Language – Data and action language in SDL-2010*.
[b-ITU-T Z.104C]	Amendment 1 (10/12) to Recommendation ITU-T Z.104 (2011), *Specification and Description Language – Data and action language in SDL-2010: Replacement Annex C on language binding*.
[b-ITU-T Z.105]	Recommendation ITU-T Z.105 (2011), *Specification and Description Language – SDL-2010 combined with ASN.1 modules*.
[b-ITU-T Z.106]	Recommendation ITU-T Z.106 (2011), *Specification and Description Language – Common interchange format for SDL-2010*.
[b-ITU-T Z.107]	Recommendation ITU-T Z.107 (2012), *Specification and Description Language – Object-oriented data in SDL-2010*.
[b-ITU-T Z.109]	Recommendation ITU-T Z.109 (2012), *Specification and Description Language – Unified modeling language profile for SDL-2010*.
[b-ITU-T Z.110]	Recommendation ITU-T Z.110 (2008), *Criteria for use of formal description techniques by ITU-T*.
[b-ITU-T Z.120]	Recommendation ITU-T Z.120 (2011), *Message Sequence Chart (MSC).*
[b-ITU-T Z.151]	Recommendation ITU-T Z.151 (2012), *User Requirements Notation (URN) – Language definition.*
[b-ITU-T Z.161]	Recommendation ITU-T Z.161 (2012), *Testing and Test Control Notation version 3: TTCN-3 core language.*
[b-ITU-T Z.161.1]	Recommendation ITU-T Z.161.1 (2012), *Testing and Test Control Notation version 3: TTCN-3 language extensions: Support of interfaces with continuous signals.*
[b-ITU-T Z.162]	Recommendation ITU-T Z.162 (2007), *Testing and Test Control Notation version 3: TTCN-3 tabular presentation format (TFT).*
[b-ITU-T Z.163[	Recommendation ITU-T Z.163 (2007), *Testing and Test Control Notation version 3: TTCN-3 graphical presentation format (GFT).*
[b-ITU-T Z.164]	Recommendation ITU-T Z.164 (2012), *Testing and Test Control Notation version 3: TTCN-3 operational semantics.*
[b-ITU-T Z.165]	Recommendation ITU-T Z.165 (2012), *Testing and Test Control Notation version 3: TTCN-3 runtime interface (TRI).*
[b-ITU-T Z.165.1]	Recommendation ITU-T Z.165.1 (2012), *Testing and Test Control Notation version 3: TTCN-3 extension package: Extended TRI.*
[b-ITU-T Z.166]	Recommendation ITU-T Z.166 (2012), *Testing and Test Control Notation version 3: TTCN-3 control interface (TCI).*
[b-ITU-T Z.167]	Recommendation ITU-T Z.167 (2012), *Testing and Test Control Notation version 3: TTCN-3 mapping from ASN.1.*

[b-ITU-T Z.168]	Recommendation ITU-T Z.168 (2012), *Testing and Test Control Notation version 3: TTCN-3 mapping from CORBA IDL.*
[b-ITU-T Z.169]	Recommendation ITU-T Z.169 (2012), *Testing and Test Control Notation version 3: TTCN-3 mapping from XML data definition.*
[b-ITU-T Z.170]	Recommendation ITU-T Z.170 (2012), *Testing and Test Control Notation version 3: TTCN-3 documentation comment specification.*
[b-ITU-T Z.500]	Recommendation ITU-T Z.500 (1997), *Framework on formal methods in conformance testing*.
[b-ISO 9899]	ISO/IEC 9899:2011, *Information technology – Programming languages – C.*
[b-ISO 90003]	ISO/IEC 90003:2014, *Software engineering – Guidelines for the application of ISO 9001:2000 to computer software*
[b-Belina et al.1]	Belina, F., Hogrefe, D., Trigila, S. (1988), *Modelling OSI in SDL* (in Turner: Formal Description Techniques), North-Holland.
[b-Belina et al.2]	Belina, F., Hogrefe, D., Sarma, A. (1991), *SDL with Applications from Protocol Specification*, Prentice Hall.
[b-Bræk]	Bræk, R., Haugen, Ø. (1993), *Engineering Real Time Systems: an object-oriented methodology using SDL, Prentice-Hall.*
[b-Guo]	Guo, F., Mackenzie, T.W. (1995), *Translation of OMT to SDL-92.* In SDL '95 with MSC in CASE, Proceedings of the Seventh SDL Forum, Elsevier, North Holland.
[b-Hogrefe]	Hogrefe, D. (1996), *Validation of SDL Systems*, Computer Networks and ISDN Systems, Elsevier, North Holland.
[b-Miga et al.]	Miga, A., *et al* (2001), *Deriving Message Sequence Charts from Use Case Maps Scenario Specifications* in SDL 2001 Meeting UML, LNCS 2076, Springer.
[b-Mitschele-Thiel]	Mitschele-Thiel, A. (2001), *Systems Engineering with SDL – Developing Performance-Critical Communication Systems*, Wiley.
[b-Olsen]	Olsen, A., *et al* (1994), *Systems Engineering Using SDL-92*, Elsevier, North Holland.
[b-OMG UML]	OMG (2011), *OMG Unified Modeling Language (OMG UML), Superstructure*. Version 2.4.1, document no. formal/2011-08-06.
[b-OOSE]	Jacobson, I., *et al* (1992), *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley.
[b-Reed]	Reed, R. (editor) (1993), *Specification and Programming Environment for Communication Software*, Elsevier, North Holland.
[b-Rumbaugh]	Rumbaugh, J., *et al* (1991), *Object-Oriented Modeling and Design*, Prentice-Hall.
[b-TIMe]	SINTEF Telecom and Informatics, *The Integrated Method (TIMe).* <http://time.sintef9013.com>
[b-Witaszek]	Witaszek, D., *et al* (1995), *A Development Method for SDL-92 Specifications based on OMT*. In SDL '95 with MSC in CASE, Proceedings of the Seventh SDL Forum, Elsevier, North Holland.

SERIES OF ITU-T RECOMMENDATIONS

Series A	Organization of the work of ITU-T
Series D	General tariff principles
Series E	Overall network operation, telephone service, service operation and human factors
Series F	Non-telephone telecommunication services
Series G	Transmission systems and media, digital systems and networks
Series H	Audiovisual and multimedia systems
Series I	Integrated services digital network
Series J	Cable networks and transmission of television, sound programme and other multimedia signals
Series K	Protection against interference
Series L	Construction, installation and protection of cables and other elements of outside plant
Series M	Telecommunication management, including TMN and network maintenance
Series N	Maintenance: international sound programme and television transmission circuits
Series O	Specifications of measuring equipment
Series P	Terminals and subjective and objective assessment methods
Series Q	Switching and signalling
Series R	Telegraph transmission
Series S	Telegraph services terminal equipment
Series T	Terminals for telematic services
Series U	Telegraph switching
Series V	Data communication over the telephone network
Series X	Data networks, open system communications and security
Series Y	Global information infrastructure, Internet protocol aspects and next-generation networks
Series Z	**Languages and general software aspects for telecommunication systems**

Printed in Switzerland
Geneva, 2015