INTERNATIONAL TELECOMMUNICATION UNION

ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

Z.100

Supplement 1

(05/97)

SERIES Z: PROGRAMMING LANGUAGES

Formal description techniques (FDT) – Specification and Description Language (SDL)

SDL+ methodology: Use of MSC and SDL (with ASN.1)

ITU-T Recommendation Z.100 – Supplement 1

(Previously CCITT Recommendation)

ITU-T Z-SERIES RECOMMENDATIONS

PROGRAMMING LANGUAGES

FORMAL DESCRIPTION TECHNIQUES (FDT)	Z.100–Z.199
Specification and Description Language (SDL)	**Z.100–Z.109**
Application of Formal Description Techniques	Z.110–Z.119
Message Sequence Chart	Z.120–Z.129
PROGRAMMING LANGUAGES	Z.200–Z.299
CHILL: The ITU-T high level language	Z.200–Z.209
MAN-MACHINE LANGUAGE	Z.300–Z.499
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.399
QUALITY OF TELECOMMUNICATION SOFTWARE	Z.400–Z.499
METHODS FOR VALIDATION AND TESTING	Z.500–Z.599

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

SUPPLEMENT 1 TO ITU-T RECOMMENDATION Z.100

SDL+ METHODOLOGY: USE OF MSC AND SDL (WITH ASN.1)

Summary

This Supplement is published as a Supplement to Recommendation Z.100 (SDL), but is also relevant to Recommendations Z.105 (SDL with ASN.1) and Z.120 (MSC). It outlines a methodology for the use of these languages in combination.

This Supplement covers the following topics:

1) Terminology definitions and background material on the use and application of SDL;

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 SDL+ (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. It is intended to be incorporated by the SDL+ users in their overall methodologies, and tailored 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 should be partially covered by a separate document 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, 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.

Source

Supplement 1 to ITU-T Recommendation Z.100, was prepared by ITU-T Study Group 10 (1997-2000) and was approved under the WTSC Resolution No. 1 procedure on the 6th of May 1997.

FOREWORD

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

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

The approval of Recommendations by the Members of the ITU-T is covered by the procedure laid down in WTSC Resolution No. 1.

In some areas of information technology which fall within ITU-T's purview, the necessary standards are prepared on a collaborative basis with ISO and IEC.

NOTE

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

INTELLECTUAL PROPERTY RIGHTS

The ITU draws attention to the possibility that the practice or implementation of this Recommendation may involve the use of a claimed Intellectual Property Right. The ITU takes no position concerning the evidence, validity or applicability of claimed Intellectual Property Rights, whether asserted by ITU members or others outside of the Recommendation development process.

As of the date of approval of this Recommendation, the ITU had not received notice of intellectual property, protected by patents, which may be required to implement this Recommendation. However, implementors are cautioned that this may not represent the latest information and are therefore strongly urged to consult the TSB patent database.

CONTENTS

SDL+ METHODOLOGY: USE OF MSC AND SDL (WITH ASN.1)

(Geneva, 1997)

1 Context

This Supplement is intended for those persons responsible for methodology in areas where the Z.100-Series Recommendations are used.

The objective of this Supplement is to provide a methodology for the effective use of the Z.100-Series Language Recommendations: Z.100, CCITT Specification and Description Language (SDL) [1]; Z.105, SDL combined with ASN.1 (SDL/ASN.1) [2]; and Z.120, Message Sequence Chart (MSC) [3]. The application of this methodology should lead to a well-defined specification of the target application in SDL combined with ASN.1 and MSC.

When Recommendation Z.105 is used, SDL is used to define behaviour and ASN.1 is used to define data. It is also possible to use SDL to define data as in Recommendation Z.100. This Supplement covers the use of SDL without ASN.1 as well as combined with ASN.1 as in Recommendation Z.105.

MSC is used to describe sequences of events and does not normally cover every possibility. MSC is used to describe what happens for particular sequences of stimuli, whereas SDL defines behaviour to all stimuli for every possible status. MSC without SDL would not usually give a full description of system behaviour. In the methodology described in this Supplement, SDL would not normally be used without MSC. The Supplement does not describe a general methodology for the use of MSC either alone or with languages other than SDL (with and without ASN.1).

Since the methodology assumes the use of SDL with ASN.1 and MSC, the term "SDL+" is used throughout the Supplement to mean "SDL (Z.100) with ASN.1 (Z.105) and MSC (Z.120)".

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 the behaviour of an entity 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 behaviour 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 (Recommendations X.208 [4] and X.680 [5]).

2.9 **association**: a dependency relationship between two or more classes (or two or more objects) in the Object Model Notation (see 13.2 and reference [11]), and shown as an annotated line between class (or object) symbols.

2.10 **behaviour**: sequence of actions with stimulus and responses 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 4.1/X.903).

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 4.1/X.903).

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 4.1/X.903).

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 which conforms to Recommendations Z.100 [1], Z.105 [2] and Z.120 [3], is consistent and unambiguous without any SDL "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, deviations include ambiguity, use of informal text, and non-conformance with Recommendation Z.100 [1]; for ASN.1, deviations include use of "any" in description of a data type (MSC 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 4.1/X.903).

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 [6] and [7]). 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, possess properties and can be managed, and characterized by:

– its state (all its properties);

– its behaviour (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 vocabular*, 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 development of a formal SDL description.

2.35 **reuse library**: storage of application concepts which 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 1 – 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.

NOTE 2 – **SDL service** (Z.100 [1]): an alternative way of specifying a part of the behaviour of a process (a communicating extended finite state machine).

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

NOTE – **SDL specification** (Z.100 [1]): a definition of the requirements of a system. A specification consists of general parameters required of the system and the functional specification of its required behaviour.

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 realised.

2.42 **technology view**: a view of a system and its environment that focuses on the choice of technology in that system. (see 4.1/X.903).

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 **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 with ASN.1 and MSC

It is probably widely accepted that the key for the success of a system is a thorough system specification and design. This requires, on the other hand, a suitable specification language, satisfying the following needs (the result of the system specification and design 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, analyze 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. 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 analyze, 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, phenomena are described as they are perceived by an observer. The natural language description of the system makes use of concepts that are derived directly from the application and implementation of the system.

When a system is specified using 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 analyze it. When designing a specification language, evidently a trade-off must be made between these two properties. In addition, it should be stressed that a model always represents a simplified view of reality, and has consequently always limitations.

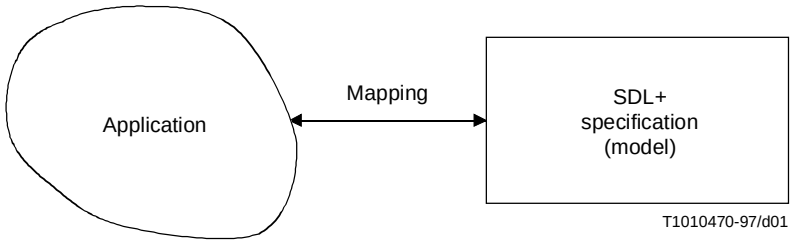


Figure 3-1/Suppl. 1 to Rec. Z.100 – How to understand an SDL+ specification

3.2 The application area of SDL+

Although 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+ 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+ 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+ specification to a programming language is possible in many cases. The amount of change required to a specification in SDL to produce an implementation described in SDL can be quite small in some cases.

3.3 Relation to implementation

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

If a decision is made to represent only the behaviour of the system (as seen at its boundary), then people normally talk about a *specification*, and the given structure of the model is only an aid to structure the specification into manageable pieces. If a decision is made to represent also the internal structure of the system, then people normally talk about a *description*. This is the reason why Recommendation Z.100 does not make any distinction between specification and description.

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

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 to 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 a 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 of 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) which 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 different roles undertaken by engineers is outside the scope of this Supplement. An engineer often has more than one role.

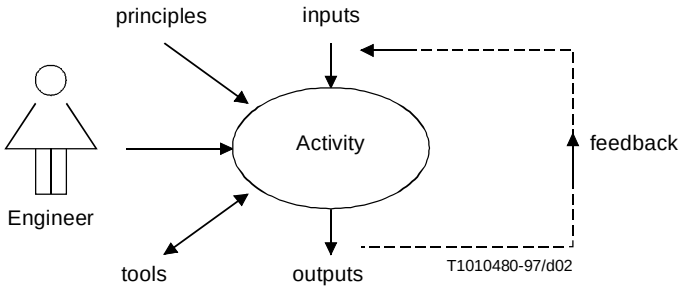


Figure 4-1/Suppl. 1 to Rec. Z.100 – An activity

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 [8] has five different views: 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.

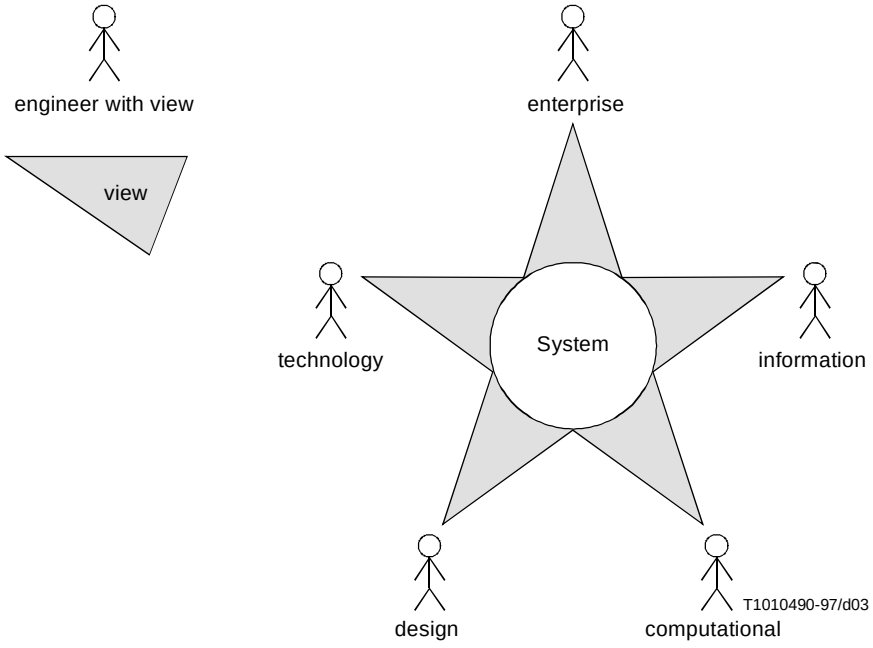


Figure 4-2/Suppl. 1 to Rec. Z.100 – The ODP views of a system

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. The enterprise view is captured by requirements, and should be used within the scope description of the application and for Validation of the product.

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). 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 processes using data defined in either ASN.1 or SDL. The system is decomposed so that it can be distributed.

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 structure shall match the requirements. In all other cases, the SDL 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 understanding 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, supplemented by MSC 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.

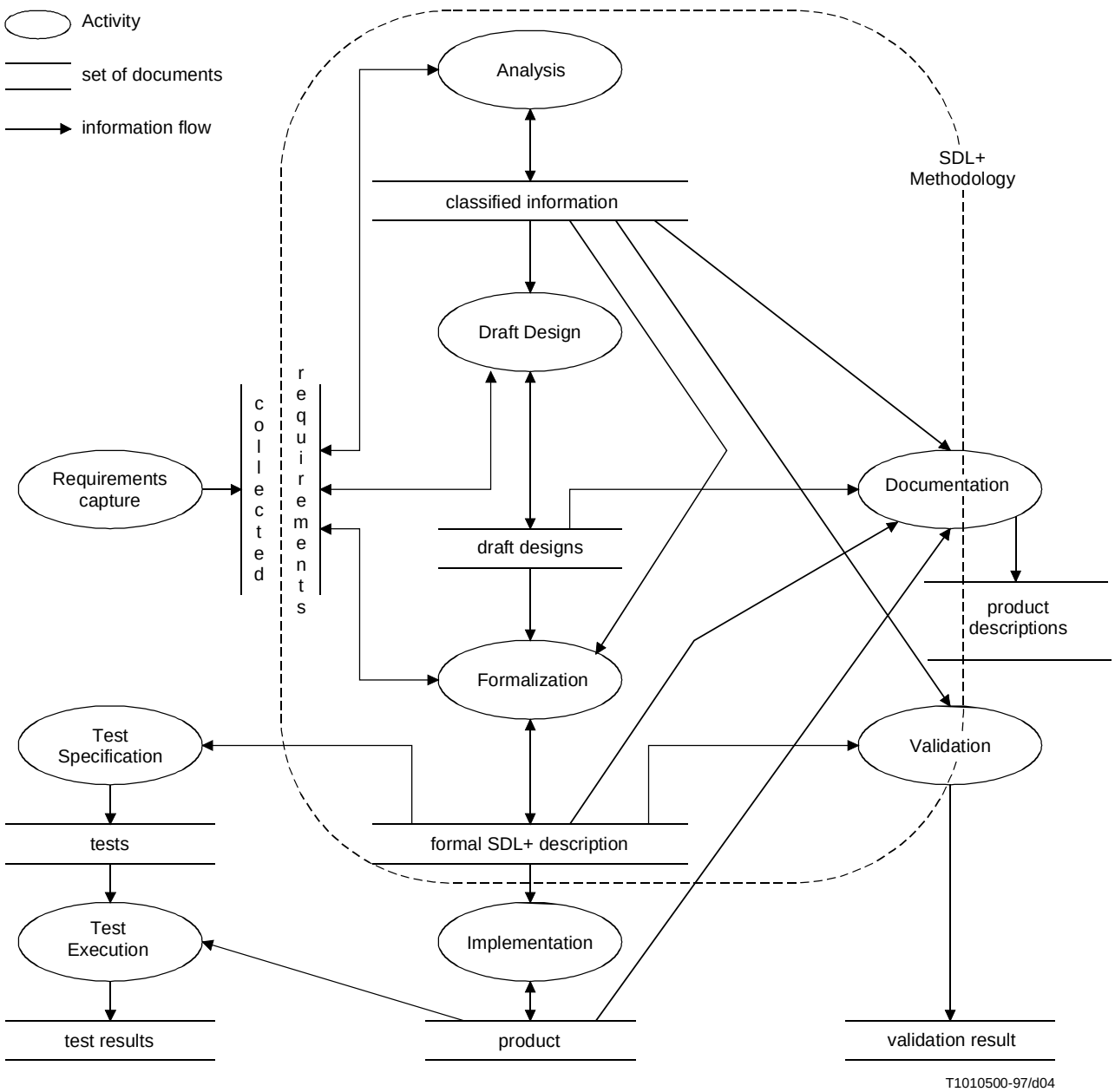


Figure 4-3/Suppl. 1 to Rec. Z.100 – The SDL+ methodology

The entire set of the seven activities (Requirements Collection, Analysis, Draft Design, Formalization, Validation, Documentation and Implementation) are 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, 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 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 requirements capture activity.

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 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 shown 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) the design views.

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.

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 Charts are 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.

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 diagrams can be considered as a Draft Design as they often contain informal parts or are incomplete according to SDL 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 sense), so that the behaviour is determined only by the formal semantics of SDL+ and does not depend on human interpretation.

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 during Draft Design is a valuable feature of the SDL language.

The complete result of the approach contains all three levels of description as compared with the formal SDL 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 walk-throughs and inspection with a check list 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 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 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. Recommendation Z.500 – Framework on formal methods in conformance testing [9] – 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.

Recommendation Z.500 defines a framework for the use of formal methods in conformance testing. It is intended for implementers, testers, and specifiers involved in conformance testing to guide 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 Supplement process as well as the development of tools for computer-aided test case generation. Recommendation 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 Recommendations X.290-X.294, OSI Conformance Testing Methodology and Framework for Protocol Recommendations for CCITT Applications [10], which are applicable to a wide range of products and specifications, including specifications in natural language. Recommendation Z.500 interprets conformance testing concepts in a formal context.

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

Testing is not further describe in this Supplement. For further information, see [9] and [10].

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 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 description are:

– draft designs such as Message Sequence Charts;

– 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 structured according to a set of principles.

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 analyzed, and draft designs have to be modified at the same time as Formalization is taking place. It is also quite usual to split up 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 application concept-modelling process.

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 is 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) OMT [11], OOSE [12] or SOON [6]. 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 which is still an informal specification that structures the information in the collected requirements and supports further development, operation and maintenance work on the system.

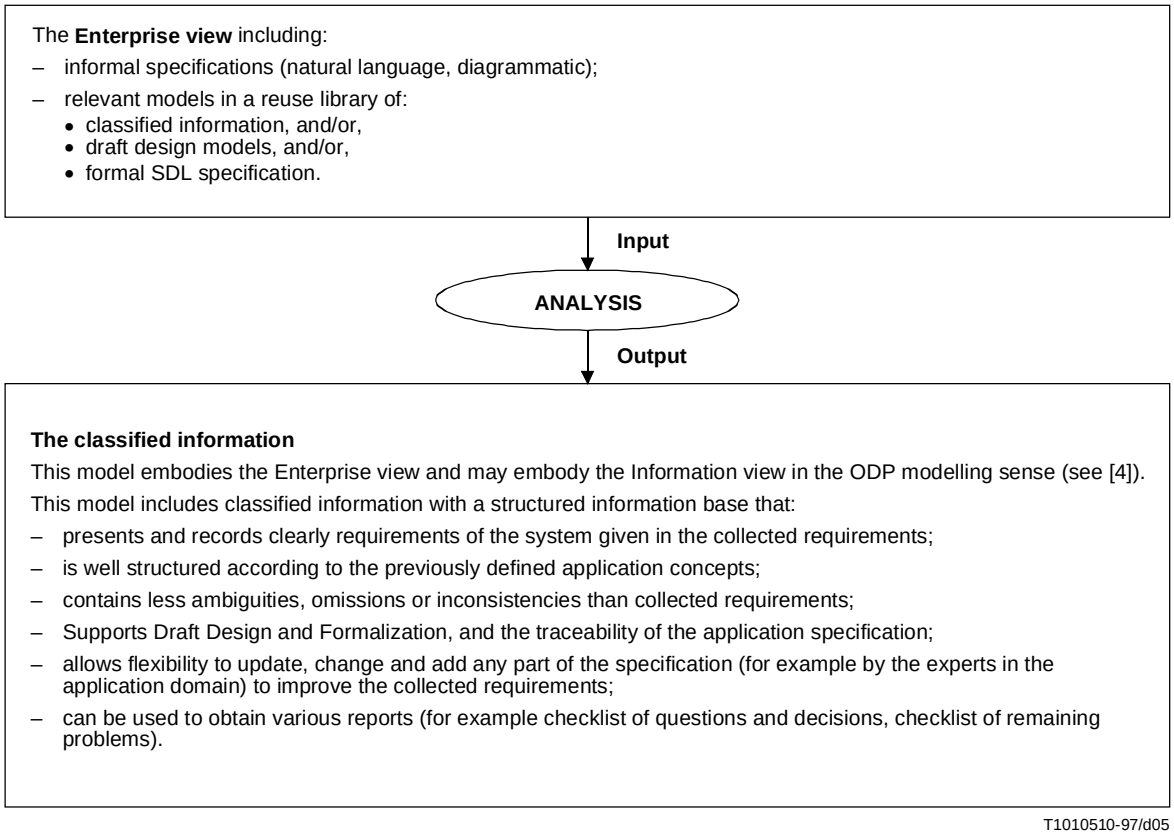


Figure 5-1/Suppl. 1 to Rec. Z.100 – Input and output of Analysis

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.

Collected requirements are well structured and Analysis may be skipped, when 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 which 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.

5.2 Questions during Analysis

In Figure 4-3, the arrow from Analysis to collected requirements represents questions that arise during 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 which 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 MSC.

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 MSC. There should be one (composite) object that represents the system and one or more objects for the environment. There can be several sequence cases in MSC 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 MSC use sequences which may be used when skeleton processes are produced in Formalization.

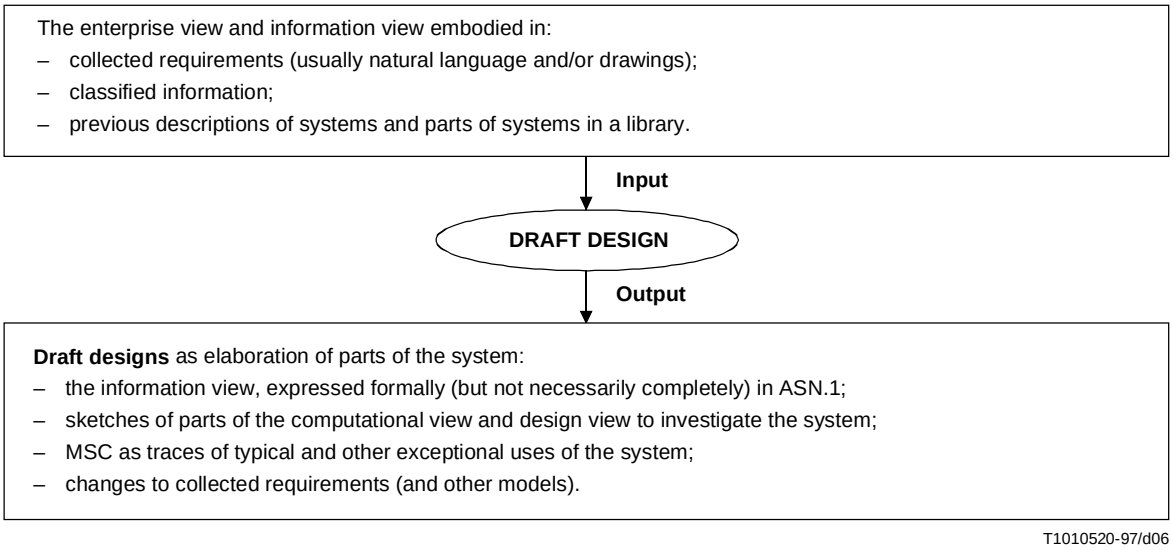


Figure 6-1/Suppl. 1 to Rec. Z.100 – Input and output of Draft Design

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 probably 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 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 analyzed;

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 MSC invoked from Formalization when 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 a block structure in SDL.

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 MSCs that describe use sequences, that is, 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 MSC 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 (that is, 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 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.

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 description are suggested in guidelines in the detailed Formalization steps in clause 15.

The engineer formalizes and generates refined designs to identify, understand and analyze 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).

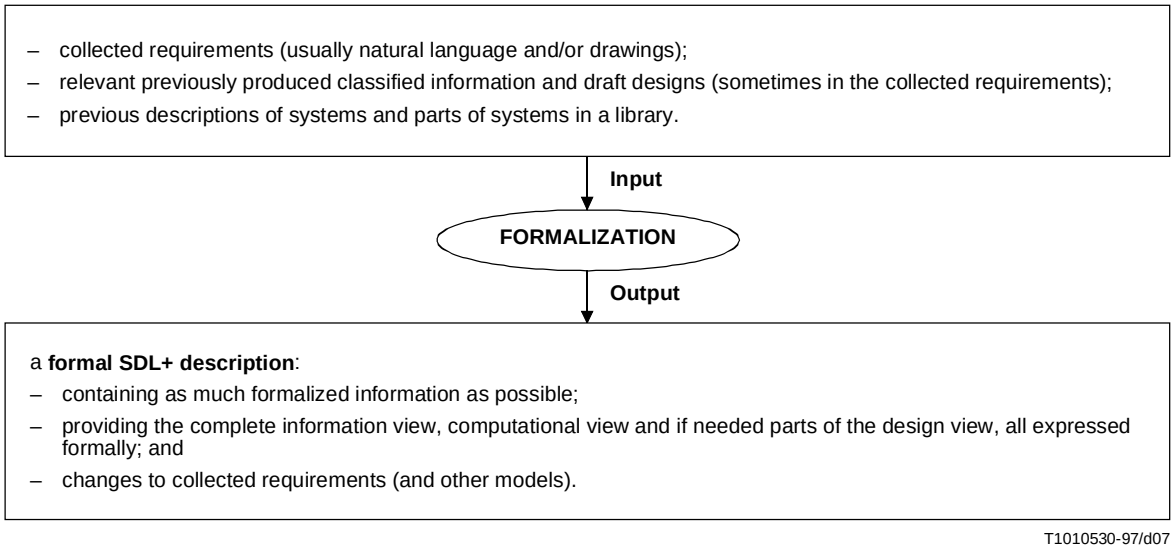


Figure 7-1/Suppl. 1 to Rec. Z.100 – Input and output of Formalization

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 creating 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 Q.1200-Series standards [13].

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 MSC have not already been produced during Draft Design, then they will be needed to help the SDL 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 as some of the descriptions may only need to be checked (and changed if needed) to be correct according to the language rules and any rules given for each step. Usually, however, such 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 MSC) 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-92 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 Recommendations Z.100 [1], Z.105 [2] and Z.120 [3].

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 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.

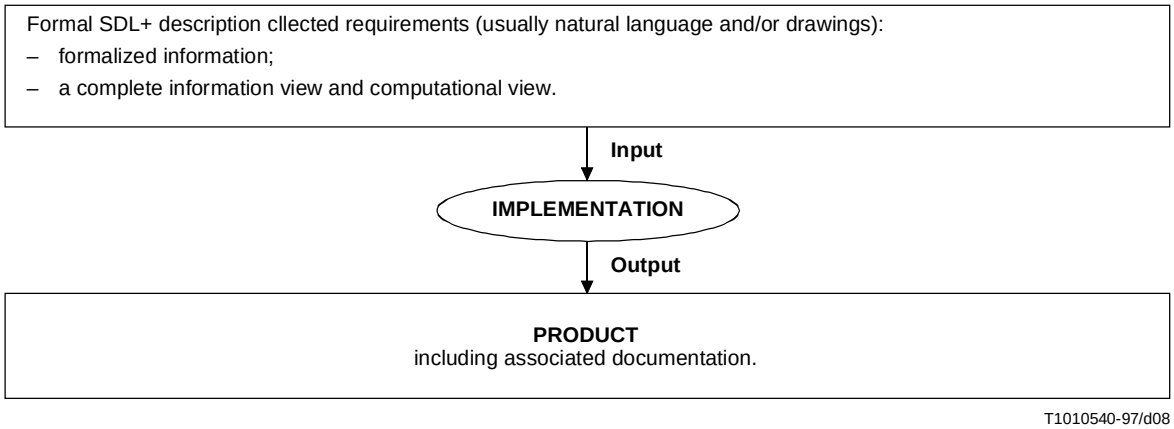


Figure 8-1/Suppl. 1 to Rec. Z.100 – Input and output of Implementation

There will be some items constraints (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 even be the case that some of the SDL is not implemented 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 can automatically translate SDL into programming languages whilst maintaining the SDL 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 description to match the distribution of the SDL across physical units. This may require turning some blocks in the SDL system into cooperating SDL systems. A run-time system to support the SDL will be needed, and there needs to be a mechanism (possibly part of the run-time system) that converts external events into SDL signals.

More detailed guidelines can be found in [6].

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, MSC, ASN.1, 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 have 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 subclause 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 also Recommendation Z.400 [14]).

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).

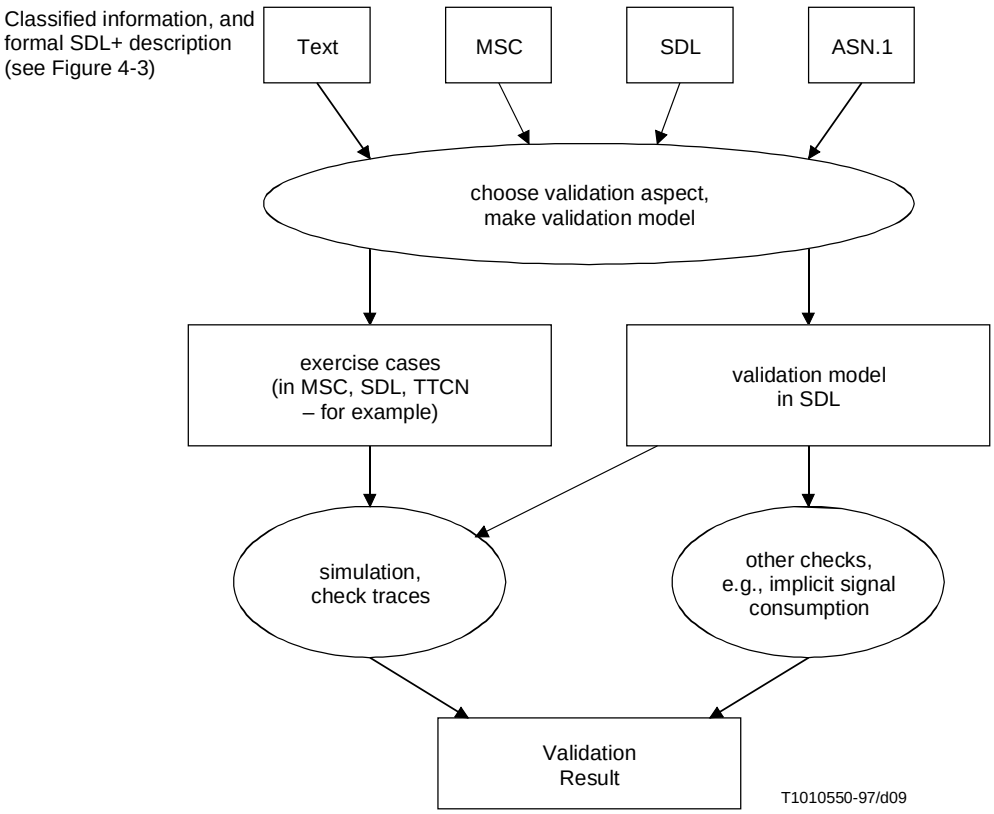


Figure 9-1/Suppl. 1 to Rec. Z.100 – General Validation scheme

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 include 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 necessary to make the model executable. These modifications can be drawn from the formalized model, assuming 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, then 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 for executability.

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) identifing 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 [15].

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 Recommendation I.130 [16] methodology 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)

Because Recommendation 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 Recomendations I.130 [16] and Q.65 [17] 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 Recommendation I.130. 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 I.130 method, SDL 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" (3.1/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 (3.2/I.130);

– in stage 3, where "SDL diagrams from the stage 2 form the basis" (3.3/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 SDL 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 SDL description appears in a separate standard.

When Recommendation I.130 was first drafted, Recommendation Z.120 [3] was not a standard; therefore, "information flows" in Recommendation I.130 should now be replaced with the use of MSC.

The I.130 approach is consistent with this methodology. Recommendation I.130 defines what descriptions should be provided and this methodology gives detail of how to produce these descriptions. The stage 1 description for Recommendation I.130 can be produced by producing a context diagram and then elaborating this with a simple SDL description that can be considered as a draft design for the functional entity SDL in stage 2. The stage 2 SDL can be produced by applying all activities in this methodology. The stage 3 SDL can be produced by a re-application of this methodology, using the stage 1 and stage 2 as part of the collected requirements.

Recommendation Q.65 gives more detail for the stage 2 method. In step 2.1, a functional model is drawn that shows the relationship between the functional entities. This should be re-drawn as an SDL system diagram with the functional entities as block types, functional entity instances as blocks and the relationships as channels. Because the SDL in a standard should be complete, the SDL version of the diagram will appear in the standard. 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 diagram and therefore gives a more abstract view of the system. The information flow diagrams from Recommendation Q.65 are MSC with a functional entity instance (SDL block) for each instance axis.

When a formal SDL description is produced for the stage 2, the functional entity actions are usually redundant as the functions are completely described by the SDL. These functional entity action descriptions may still be useful to help explain the SDL and because some existing standards use them extensively. If they are used, then they should be clearly marked as informative.

10.2 Relationship with OSI Layered modelling

This subclause updates SDL-92 material that was first published in [18], [19] and in Appendix I clause Z.100 [1].

The OSI concepts used in this subclause are defined in [20] and [21]. In order to make the subclause more self-contained, an explanation is given of the key concepts.

A *layer service* is offered by a *service provider* for a given layer. The service provider is an abstract machine offering a communication facility to *users* in the next higher layer. The service is accessed by the users at *service access points* by means of *service primitives* (see Figure 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/Suppl. 1 to Rec. Z.100 – A layer service provider

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

Except for the highest layer, the users of a layer service are *protocol entities* of the next higher layer, which cooperate in order to enhance the features of the layer service, thus providing a service of the next higher layer. The cooperation is carried out in accordance with a predefined set of behaviour rules and message formats, which constitute a *protocol*. According to this view, protocol entities of (N)-layer and the (N-1)-service provider together provide a refinement of the (N)-service provider (see Figure 10-2).

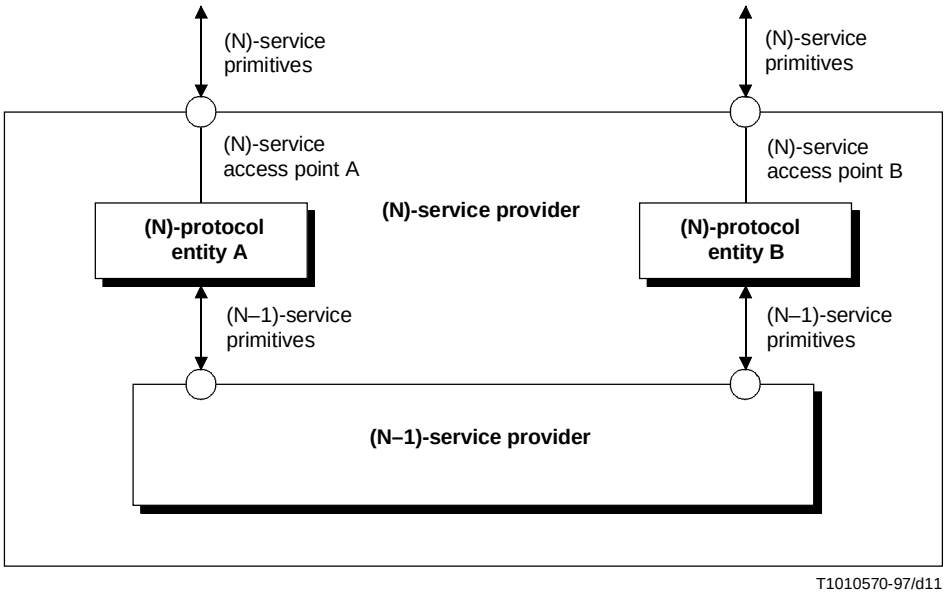


Figure 10-2/Suppl. 1 to Rec. Z.100 – Refinement of the (N)-service provider

The refinement of the (N)-service provider shown in Figure 10-2 may of course be much more complicated. For example, there may be (N)-relay protocol entities which are not connected to any protocol entity of the (N+1)-layer. Such cases are not consider 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 re-order messages, in which case the protocol should contain mechanisms of error detection and correction, resynchronization, retransmission etc., in order to provide a reliable and usually more powerful service to the next higher layer.

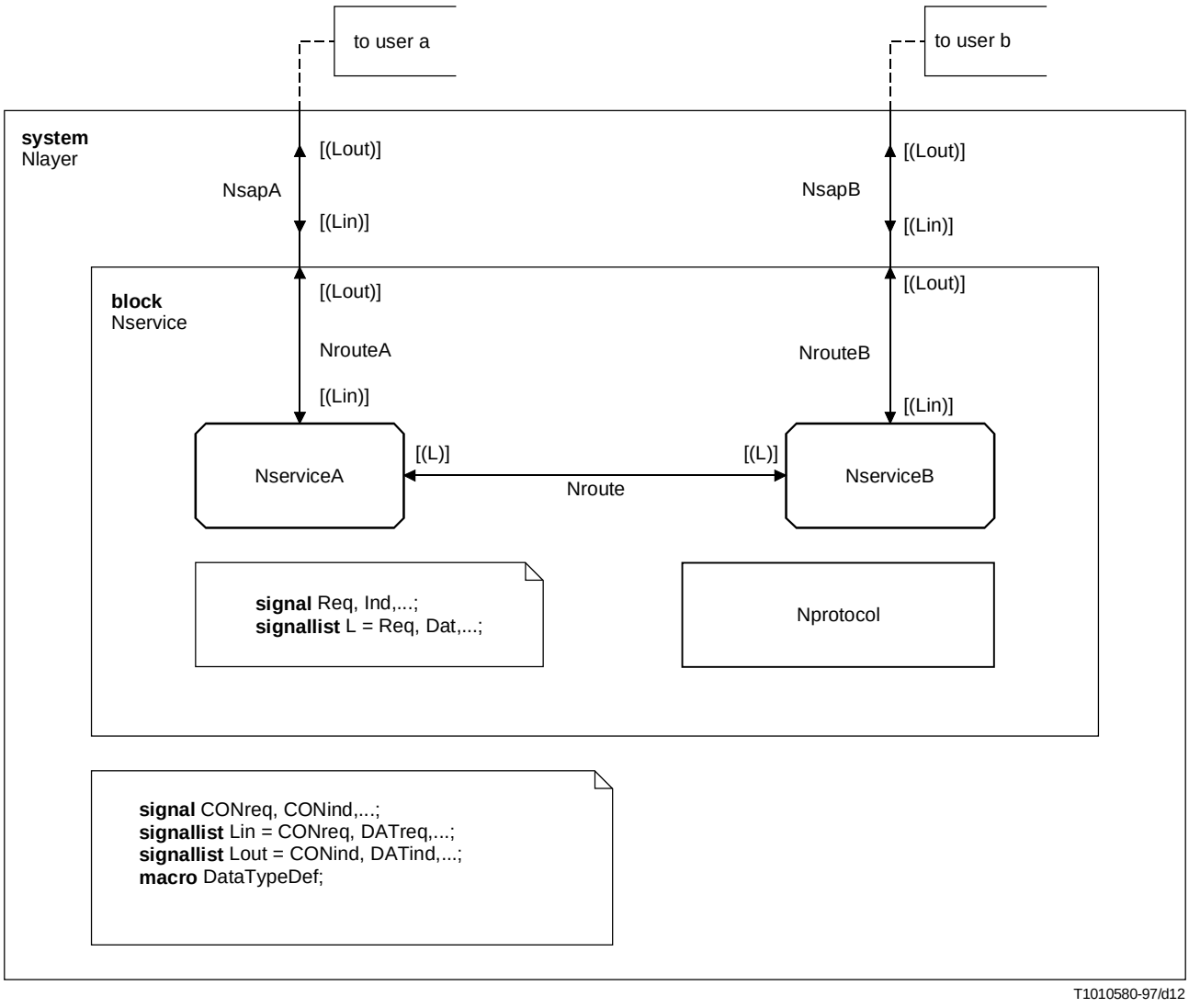
The OSI architectural concepts can be modelled in SDL in a number of alternative ways, mainly depending on what aspect should be emphasized. First, a basic approach is described, then other approaches are outlined as variants of the basic approach.

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

10.2.1 Basic approach

Service specification

A service specification for layer N can be modelled in a straightforward manner as a block *Nservice* containing two processes *NserviceA* and *NserviceB* (see Example 10-1).



Example 10-1/Suppl. 1 to Rec. Z.100 – (N)-service specification in SDL

In Example 10-1, the users of this service are in the environment of the system and can be considered as processes, capable of communicating with the system on terms of this system.

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

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

In Example 10-1, some examples of signal specifications are shown. As suggested by some signal names, a connection oriented service is assumed. In the case of a connectionless service, a great deal of simplification can be made. However, this case will not be treated further, for the sake of brevity.

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

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

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

Protocol specification

The protocol specification for layer N is modelled by the substructure *Nprotocol* of the block *Nservice*, see Example 10-1.

In the *block diagram* of Example 10-1, a block substructure reference (a block symbol containing the name *Nprotocol* of the block substructure) has been introduced. The specification of the block substructure is given in a remote *block substructure diagram* (see Example 10-2) containing three blocks: *NentityA*, *NentityB* and *N_1service*. The first two blocks represent (N)-protocol entities, while the block *N_1service* represents the (N-1)-service provider. The specification of *N_1service* is analogous to the specification of *Nservice*, and is not shown in this diagram (being a remote specification).

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

10.2.2 Alternative approach using channel substructure

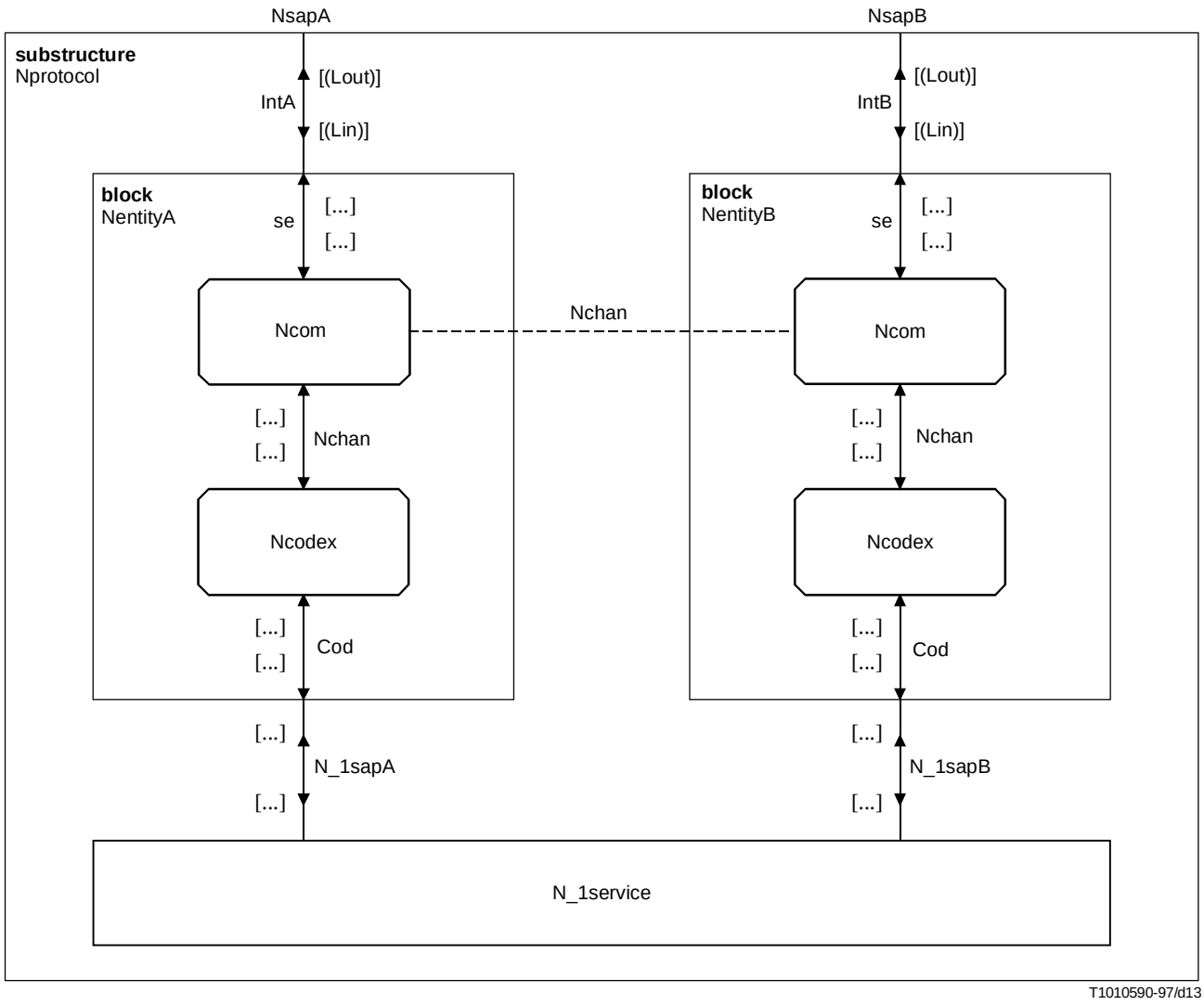
This approach is obtained from the basic approach in Example 10-2 by grouping the processes differently, introducing the real channel *Nchan* and using channel substructure (see Example 10-3). The channel *Nchan* conveys protocol data units, indicated by the signal list *Npdu*. This approach emphasizes the protocol view and the horizontal orientation of OSI.

Note that in this approach the blocks within a channel substructure do not represent protocol entities, and overlap two adjacent layers. The service primitives are hidden in these blocks, and are conveyed on the signal routes *N_1sapA*, *N_1sapB*, *N_2sapA*, *N_2sapB*, etc. However, the chosen highest (N)-layer must be treated separately, as indicated in the example. Note also that the system diagram (Example 10-1) is not affected by this approach.

10.2.3 Symmetrical OSI architecture

When the OSI architecture is symmetrical, that is, when entities of the two sides of the OSI architecture are mirror images of each other, the specifications of these entities are identical except for the entity name. The common specification can be given by instantiation of a type, the process type *Nservice* in the block *Nservice*. The process type, *Nservice*, is then instantiated into *NserviceA* and *NserviceB* (see Example 10-4). *x* and *z* are actual parameters corresponding to the gates (not shown here) of the *Nservice* process type. In this example only the block diagram of *Nservice* in Example 10-1 is shown. Note that service specifications are always symmetrical; only protocol specifications can be asymmetrical.

An alternative approach is to represent only one side (see Example 10-5), which is a modification of the system diagram in Example 10-1. The channel *NsapB* has been replaced by the channel *Nchannel*, conveying the internal signals *L*. These signals are now on the system level, and their specifications have been moved accordingly. Note that this approach cannot be used in combination with channel substructuring (shown in Example 10-3).



Example 10-2/Suppl. 1 to Rec. Z.100 – (N)-protocol specification in SDL

10.3 Relationship with Q.1200-Series (IN) architecture and SIBs

The Q.1200-Series [13] "Intelligent Network" (IN) architecture does not in itself present any basic difficulties in mapping to SDL, but there are a number of different ways that SDL can be used to support Service Independent Building Blocks (SIBs). However, the linking of basic call processing to the service logic is a significant modelling issue that needs to be considered when the IN approach is used. The major difference between this approach and the I.130 approach is the introduction of a global functional plane above the (distributed) functional plane and the use of SIBs.

The distributed functional plane can be modelled in the same way as stage 2 in Recommendation I.130.

To support IN using SDL, there needs to be an SDL model for the global functional plane with:

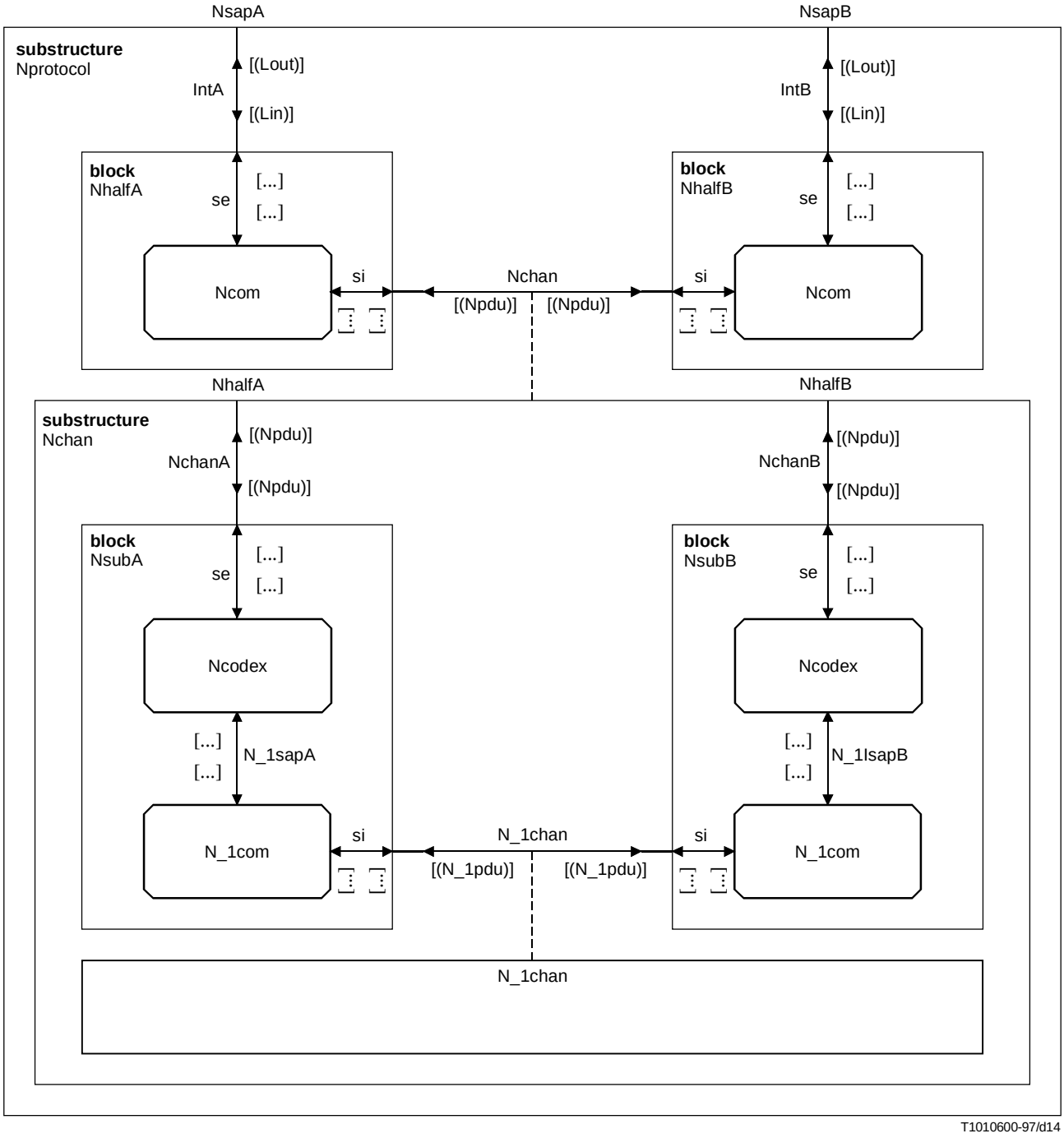
– a well defined global basic call processing (BCP), preferably in formal SDL but at least with clearly defined interfaces for the calling of the first SIB in a chain;

– a way that the invocation of a new service can be added to the global BCP;

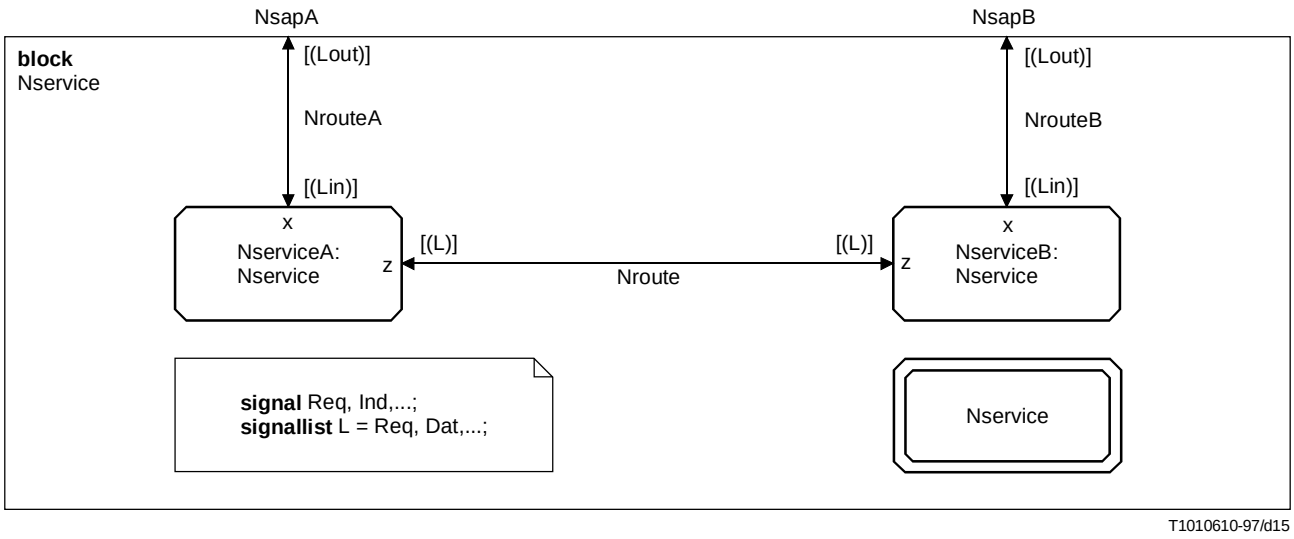
– a way that SIB can be specified in SDL and chained together;

– a way of using SDL so that the global BCP and service logic can be in different standards.

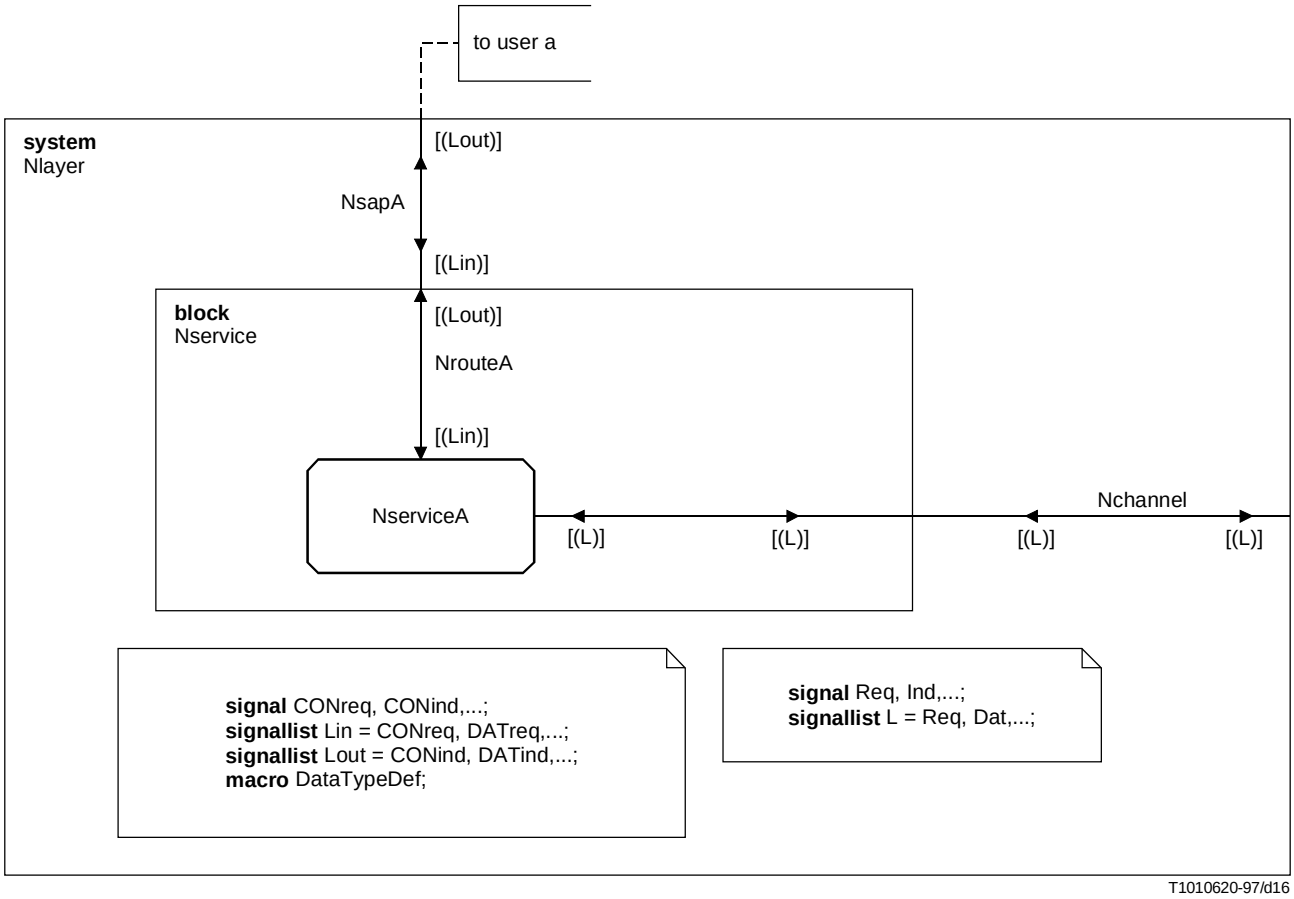
SDL-92 provides mechanisms that can be used to support the IN approach.



Example 10-3/Suppl. 1 to Rec. Z.100 – The protocol view of OSI



Example 10-4/Suppl. 1 to Rec. Z.100 – Using process type to represent a symmetrical OSI architecture



Example 10-5/Suppl. 1 to Rec. Z.100 – Representing only one side of a symmetrical OSI architecture

To call SIBs from BCP, BCP itself should be defined as a process type. The points at which SIBs are likely to be called should either be in virtual procedures or in virtual transitions in the BCP. The call of the first SIB can then be added to the BCP by changing the virtual part of the process. The actual process definition then depends on the service interactions provided by BCP.

The SIBs themselves can be modelled in several ways. They can be procedures of BCP, but if so they cannot run in parallel with BCP or each other. Alternatively, the SIBs can be remote procedure calls to some service logic process(es), or an SDL process. The calling of a SIB can be done by remote procedure calls or creating a SIB process or by signal passing. The points of return to BCP need special attention because there may be more than one. This suggests that a procedure interface may not be suitable, unless the procedure return is always followed by a decision based on a returned value from the procedure.

The global data and call data needs to be well defined. There needs to be a mechanism for handling the database for this data.

Once these architectural issues are fixed, a formal SDL description for BCP and for SIBs can be produced, using the general architecture as part of the collected requirements and a constraint on design. The methodology of this standard can then be followed.

10.4 Relationship with X.219 Remote operations (RO and ROSE)

X.219 [22] 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 remote procedure. If the operation never has a response (class 5), it can be mapped onto an SDL 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 process behaviour). Similarly, association is not mapped onto any explicit SDL 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

```
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.

10.5 Relationship with Recommendation X.722 (GDMO)

NOTE – Although this is potentially important for the design and perhaps the standardization of telecommunications systems where the resources are treated as managed objects, further study is needed to agree on a method for combining GDMO with the standards for the resources.

GDMO [23] uses ASN.1 as a basis. The GDMO template provides a BEHAVIOUR keyword after which the behaviour of an object can be defined, but the definition is in the form of a string that usually contains natural language. The described behaviour is only the behaviour of the managed object that is relevant to management. Typically, the object has other behaviour that is standardized by a service or protocol standard. The benefits of describing the management behaviour of an object in SDL are:

– The management behaviour of the object would be better defined, and for complex managed objects this behaviour can be quite complex.

– It allows better checking of the consistency of the management description and functional description.

– The management description and functional description can have some common parts and a basis for merging the two in an implementation.

The ASN.1 basis of GDMO provides a basis for defining the managed object in SDL.

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+. It is based on several years of research and experience. Some of the material on which the Supplement has been based can be found in [24] and [25].

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.

2) There are many different ways of implementing a product from an SDL specification.

3) Time to market is often more important than execution efficiency and SDL 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).

It is expected that as more standards are written using SDL in a formal way, the SDL descriptions in standards will be used as the basis of procurement, product implementation and product testing. Special attention is therefore given to the combination of the methodology in this Supplement with other techniques used for writing standards.

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 SDL users asked for this approach and the steps presented in Appendix I/Z.100 [1] 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 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 difference between a *guideline* and a *rule* is that the 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 Q.65)

The relationship between the three-stage methodology of Recommendations I.130 [16] plus Q.65 [17] and the methodology in this Supplement is described in subclause 10.1. Here the mapping of Q.65 onto SDL constructs is explained.

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

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

Structure

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

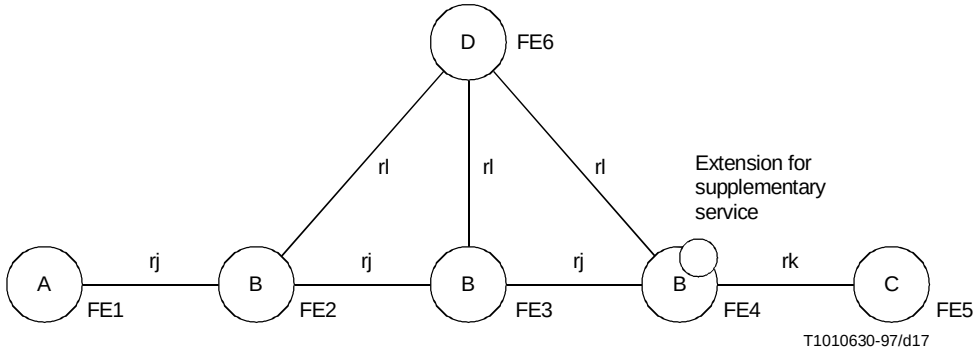


Figure 12-1/Suppl. 1 to Rec. Z.100 – Example of a functional model according to Recommendation Q.65

The elements of the figure are:

– Names within circles (e.g. *A*): types of functional entities;

– Names in upper case adjacent to circles (e.g. *FE1*): names of functional entities (instances);

– Names in lower case between circles (e.g. *rj*): relationships between types of functional entities;

– Added shadowing circle: extension for supplementary service.

Note that this model clearly distinguishes between types of functional entities (e.g. *A*) and their instantiation (e.g. *FE1*) when describing the structure of the system. Recommendation Q.65 also mentions the convenience of describing two functional entity types as subsets of the same single functional entity type, if the two functional entity types have much commonality. These features can be expressed by using the **type** features of SDL.

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

– Model → system;

– Functional entity → block with one process;

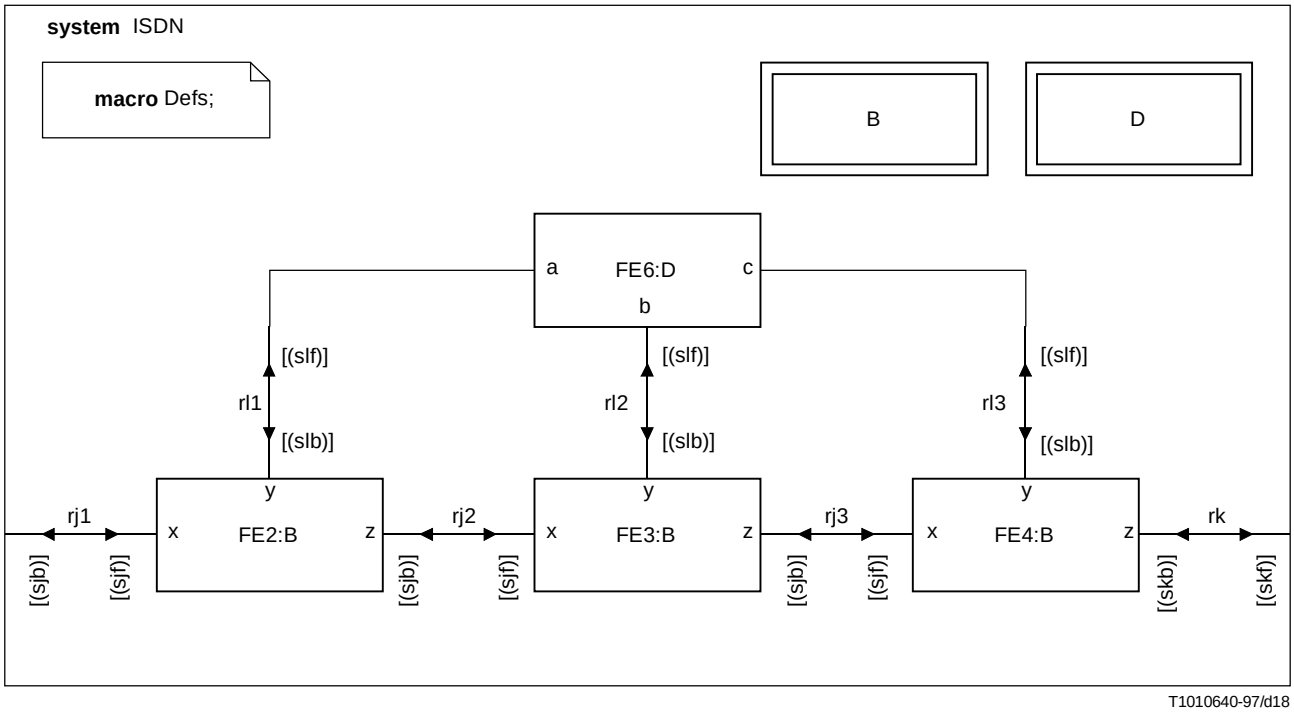
– Relationship between functional entities → channel.

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

– Type of functional entity → block type.

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

Example 12-1 shows the functional model of Figure 12-1 in SDL. Note that *FE1* and *FE5* are missing within the system. The communication with these functional entities is modelled as communication with the environment. Leaving *FE1* and *FE5* in the environment anticipates that one does not intend to describe the behaviour of *FE1* and *FE5*.



Example 12-1/Suppl. 1 to Rec. Z.100 – SDL version of the example in Figure 12-1

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

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

Behaviour

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

The information flow diagrams correspond to MSC [3]. The information flow diagrams are produced for "normal" cases, and SDL is then used to describe the behaviour of the functional entities. This occurs in the Draft Design and Formalization activities in the methodology in this Supplement.

Step 4 of Recommendation Q.65 is concerned with defining a number of basic actions for each functional entity. The idea is then to reuse these basic actions in different service specifications.

The elements of the diagram are:

– Names on top of columns (e.g. *FE1*): functional entities;

– Names in lower case between circles (e.g. *rj*): relationships between types of functional entities;

– Names on arrows between columns (e.g. *ESTABLISH X req.ind*): information flow;

– Names within brackets adjacent to information flow names (e.g. *location*): additional information conveyed by the information flow;

– Names within columns (e.g. *Action B, 100*): names of basic functional entity actions;

– Numbers within parenthesis [e.g. *(5)*]: labels to the SDL diagrams.

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

– functional entity name → process instance (one process instance per block);

– information flow → message;

– additional information → message parameters;

– basic functional entity action → action;

– labels to the SDL diagrams → comments.

The information flow diagrams recommended in Recommendation Q.65 thus basically contain the same information as the MSCs. The SDL specifications can in general be checked against the MSCs during simulation, and in some simple cases (no dynamic creation of process, no dynamic addressing of outputs) this check can easily be done without simulation. MSCs can be derived from SDL specifications, but normally only a subset of the behaviour ("the normal behaviour") is meant to be shown in an MSC, so some human interaction is needed to derive the appropriate MSCs from the SDL specification. Another possibility is to derive automatically skeletons of SDL process diagrams from the information flow diagrams. See 15.2.2, **Step B:2 – Skeleton processes**.

Data

Tasks and **decisions** are mainly informal in the existing SDL-diagrams for ISDN. Therefore, there is little need of defining operators for data types. Data items mainly appear 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.

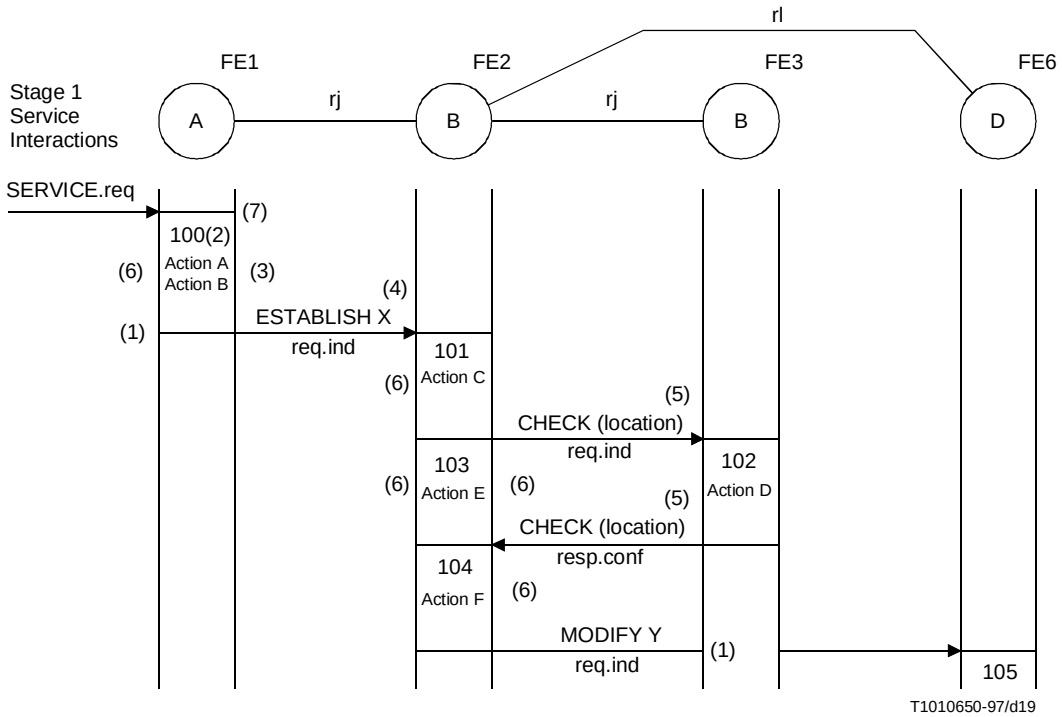
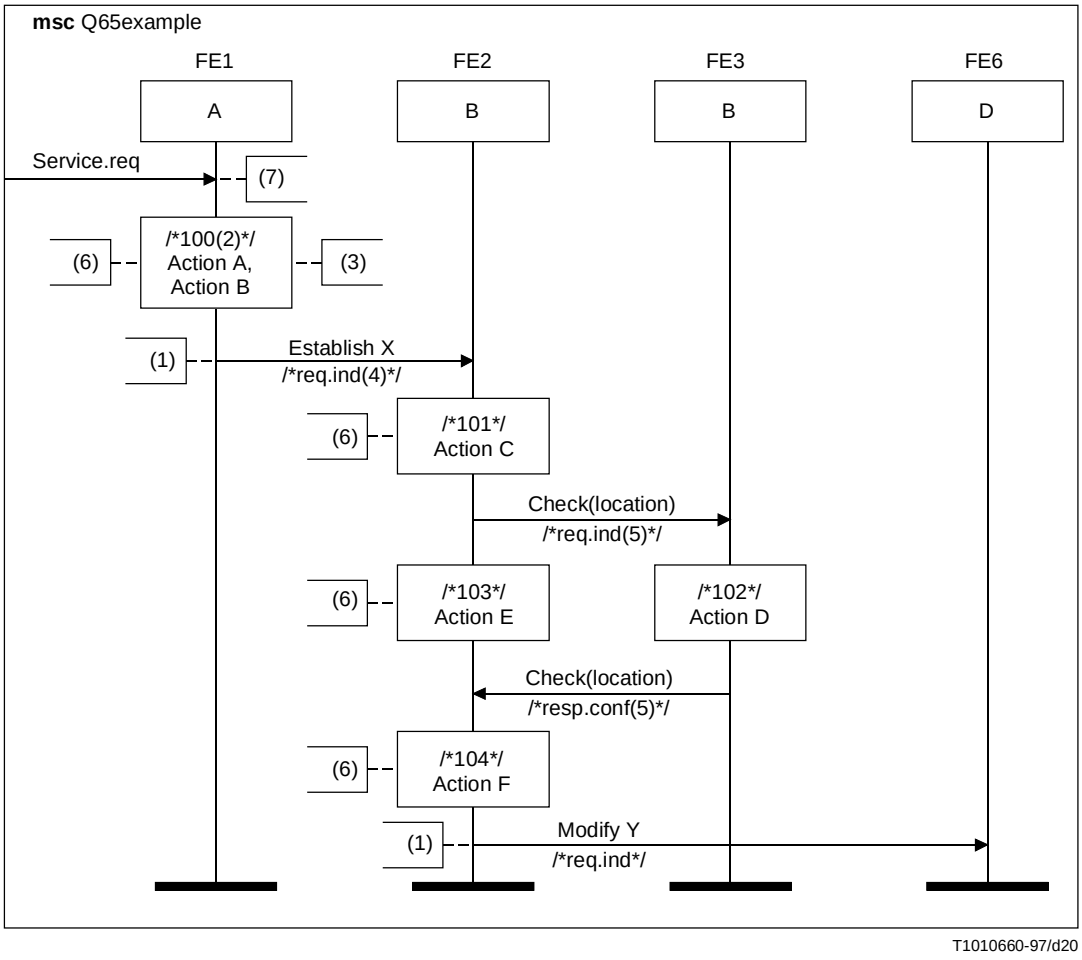
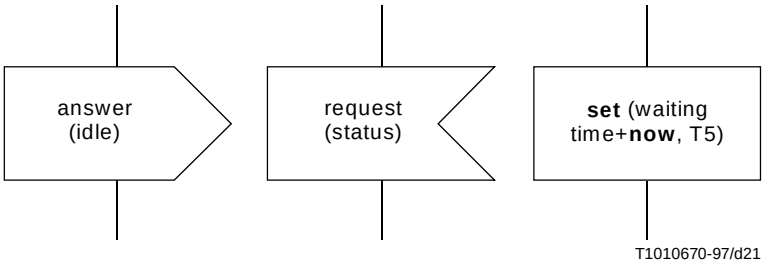


Figure 12-2/Suppl. 1 to Rec. Z.100 – Example of information flow diagram for Recommendation Q.65



Example 12-2/Suppl. 1 to Rec. Z.100 – MSC version of the example in Figure 12-2



Example 12-3/Suppl. 1 to Rec. Z.100 – Use of data

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

newtype **Indication**
literals **idle, busy, congestion;**
endnewtype **Indication;**
...
signal **Check location (Indication) /*resp.conf */;**
...
output **Check location (busy) /*resp.conf*/;**
...

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

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 lead to 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 a 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) techniques such as OMT [11], OOSE [12] or SOON [6] shall be preferably used. 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 close components are performed by introducing new appropriate components. As a consequence, OOA techniques are very useful to create and reuse domain components (that is, components appropriate to a domain). The Analysis steps are explained in detail by using the OMT approach, but one of the other OOA approaches could be used.

MSC 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 [12].

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 system 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 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 modelling entities of the environment (physical as well as logical) are called *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 relation between objects (physical or logical);

– use relation 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 and when information is generated or passed from place to place, nor concerned with how and 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) must 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 Object Model Notation of OMT [11] the basic elements of which are shown in Figure 13-1. Figure 13-2 shows a partial diagram where the application domain relates to banking operations with cash cards.

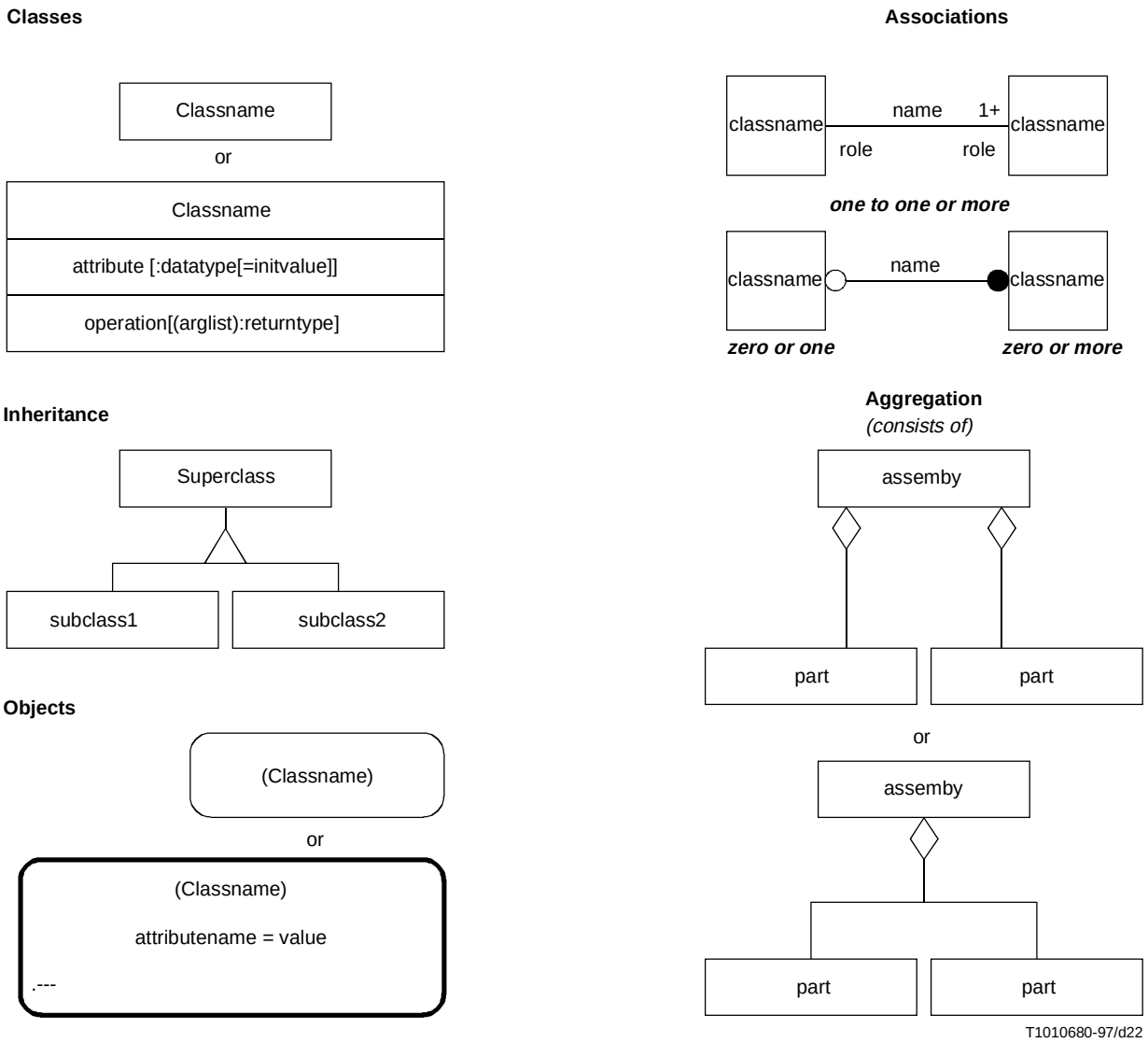


Figure 13-1/Suppl. 1 to Rec. Z.100 – Basic elements of the Object Model Notation

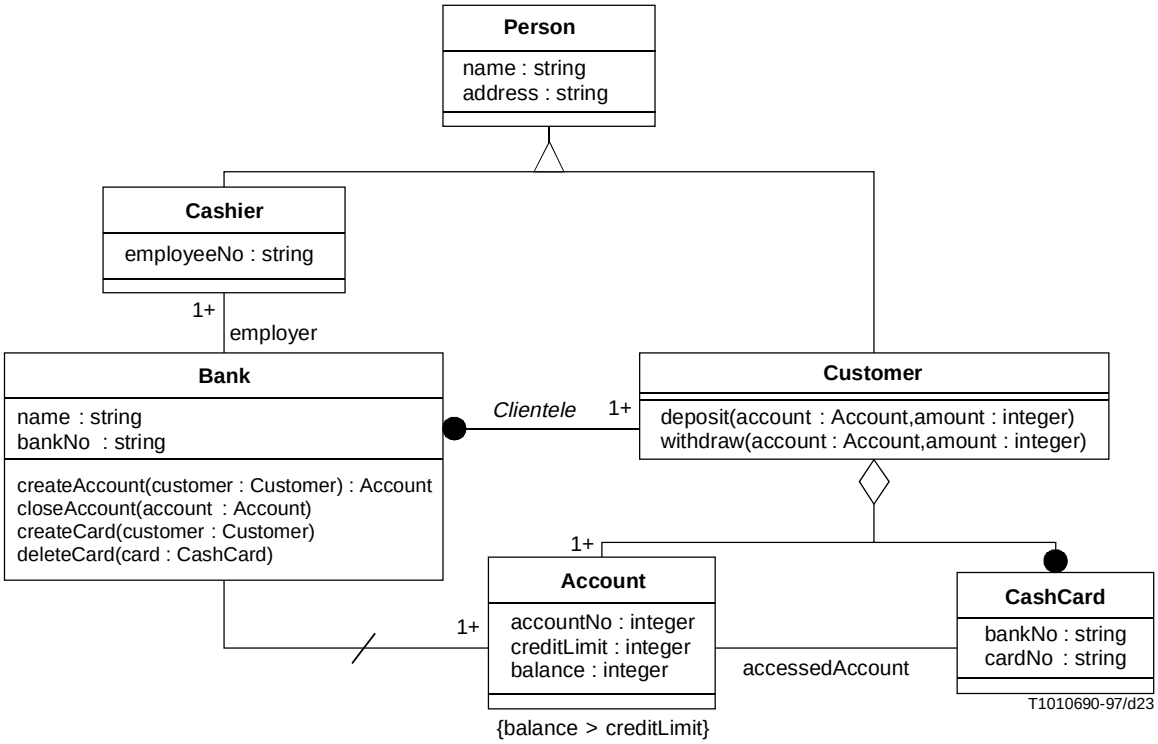


Figure 13-2/Suppl. 1 to Rec. Z.100 – Example of the Object Model Notation

– 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, the point of view of a service to be provided by the system. The collection of all these modules constitute 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 identify 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 which 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 data base 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 data base. Redundant classes (which represent the same concept, the same physical entity or which contain the same attributes) must be removed. Empty classes, i.e. classes without attributes and loosely coupled with the other classes of the model, must 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 which 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 which have partially common structure (attributes or associations) could be reorganized by introducing a new class which encapsulates this common sub-structure. 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 class 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 together 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 complete 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 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 MSC usually constituted by the typical use sequences (in fact the mission of the system) and by exceptional use sequences which 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 at classifying 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 agents involved in the communication with the system. These 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 MSCs to be created. The set of produced MSCs 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 a special attention to the structure of the MSC document.

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 MSCs (HMSCs).

3) For each low-level use sequence, draw the MSC 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 MSCs.

5) Verify that all the typical and exceptional use sequences collected in the requirements are covered by the MSC document.

6) Verify the consistency of the dynamic model, as described by the MSC document, with the object model.

Guidelines

Identifying the required use sequences

The use sequences to be produced are identified from the requirements which 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 which can occur and should be considered when nominal sequences are running.

In order to avoid redundancy between described sequences, and to ease reuse of parts of sequences, the expected behaviour should be divided into small and reusable MSCs. In MSC-96 there are several ways to combine MSCs.

– MSC references are used to refer to other MSCs from within an MSC.

– Complex use sequences should be described by High-level MSCs – a directed graph of MSC-references.

– Combination of MSCs can also be described as MSC expressions within MSC references, e.g. "MSC1 **seq** (MSC2 **alt** MSC3)".

– Small variations where, for example, there is a choice between alternative messages, are best shown by inline expressions.

Simple MSCs 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 of an MSC

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

The system appears as a vertical bar in an MSC. There may be as many instances of the system in the MSC 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 MSC contains several instances of the system. In the same way, an agent can be multi-instantiated in an MSC if this situation occurs in the real world. Different instances of the same agent should be carefully named in order to have an understandable MSC, for example "Caller" and "Called" in an MSC involving two subscribers in dialogue. It is usual to limit the number of instances in the MSC 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 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 MSC instances which are in dialogue. However, associations of the object model need not necessarily result in messages of the MSCs.

Verifying the completeness of the dynamic model

Each possible scenario should be covered by an MSC. 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) MSC 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 which do not want to directly access to "b" (e.g. 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 MSC 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 MSC.

Rule 5 – For showing the sequences of messages exchanged between the system and its environment against time in diagrams, the MSC 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 (MSC 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/Suppl. 1 to Rec. Z.100 – 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 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.

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

NOTE – Bit tables can be used to describe encoding.

14.1 Component relationship modelling

The objective of this step is to complete the class 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 class 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 class 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 class 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 OMT, sorts of data are declared at the model level only and then are accessible by all the classes; sorts of data cannot be declared locally to a class. 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 this 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 bi-directional and enable the access to class instances without knowing how it is possible. The Draft Design should explain how to access to 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 block in the context diagram into subblocks using the top level aggregation and decomposition structure from the classified information (that is, the class model).

3) Repeat decomposition for the subblocks until the behaviour of each block cannot sensibly be further divided or it is obvious that the block corresponds to a single SDL process.

Guidelines

The context diagram is an SDL system diagram containing a single block that represents the behaviour of the system. The channels leading to and from the block represent the information flow between the system and the 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 MSC 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 MSC). 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 SDL system connecting with the environment. 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 classes at the top level (not parts of other classes – that is, not aggregated) and are not agents (classes for objects in the environment) can be assumed to be aggregated to form the system.

The subblocks 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.

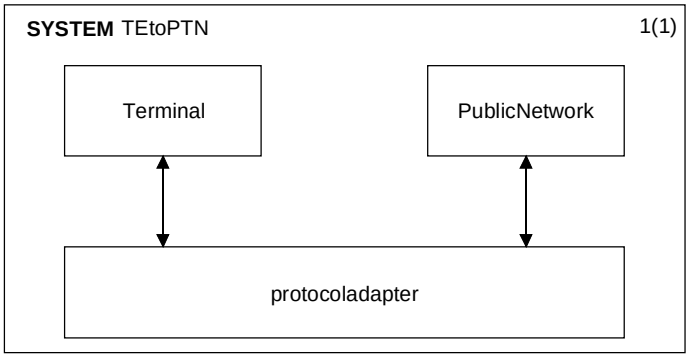
These descriptions use SDL to show data and control flow but do not show signal detail or process behaviour. The approach allows various decompositions to be investigated and to focus on the data flow and data repositories. Whenever a "data block" has connections to more than one block, the decomposition should be reconsidered, as the data should be contained in one process in the formalized model.

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.

Example



NOTE – For the context diagram, the SDL need not be complete: for example, the channels have no names and signals are not defined in this example.

Figure 14-1/Suppl. 1 to Rec. Z.100 – Context diagram for protocol adapter

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 the 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 in ASN.1. ASN.1 basic encoding (X.209 [10]) is not always suitable for a protocol; for example, it may be too inefficient. In this case, explicit encoding is used. A simplified form of the bit tables may be needed in the documentation to describe encoding.

SDL data 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 provides good support for the sorts of data needed.

Table 2/Suppl. 1 to Rec. Z.100 – 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" is implemented
        fdi     FixedIdentity OPTIONAL;             -- only present if "incoming call" is implemented
        bs      BasicService OPTIONAL;              -- only present if "incoming call" is 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

        cnal    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
        }
```

If there is no reason to prefer SDL 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 description, but encodes the interfaces so that the bit patterns 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.

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 data and control flow model is available and the use sequence model will be detailed in accordance to this SDL description.

The additional tasks mainly concern: the use of SDL entities instead of classes of the object model, the use of SDL 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, an information generated by an external entity was modelled as a unique abstract message, whereas in Draft Design, this information will be modelled by the set of interactions which occurs in the real world. These additional tasks lead also to introduce new MSC detailing the abstract exchanges.

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

Instructions

1) Follow instructions 1 to 5 of 13.3 to produce MSC but model MSC instances within the system that correspond to SDL entities (blocks or processes or services) internal to the system.

2) Check the consistency between the MSC at this level and the MSC in the classified information.

3) Produce (optionally) MSC for lower level interactions within SDL and check the consistency with MSC at higher levels.

Guidelines

Interfaces to the environment (or 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 item, the names of the instances on the chart can be the same as the names of the corresponding classes or SDL items. If there is more than one instance, they must have unique names and the class (or SDL 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 MSC 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. 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 charts into simple MSCs that are referenced from one main MSC, but this is useful when many sequences have the same "set up" or "clear down". When MSC references to simple MSCs and 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 MSC notation shall be used.

Example

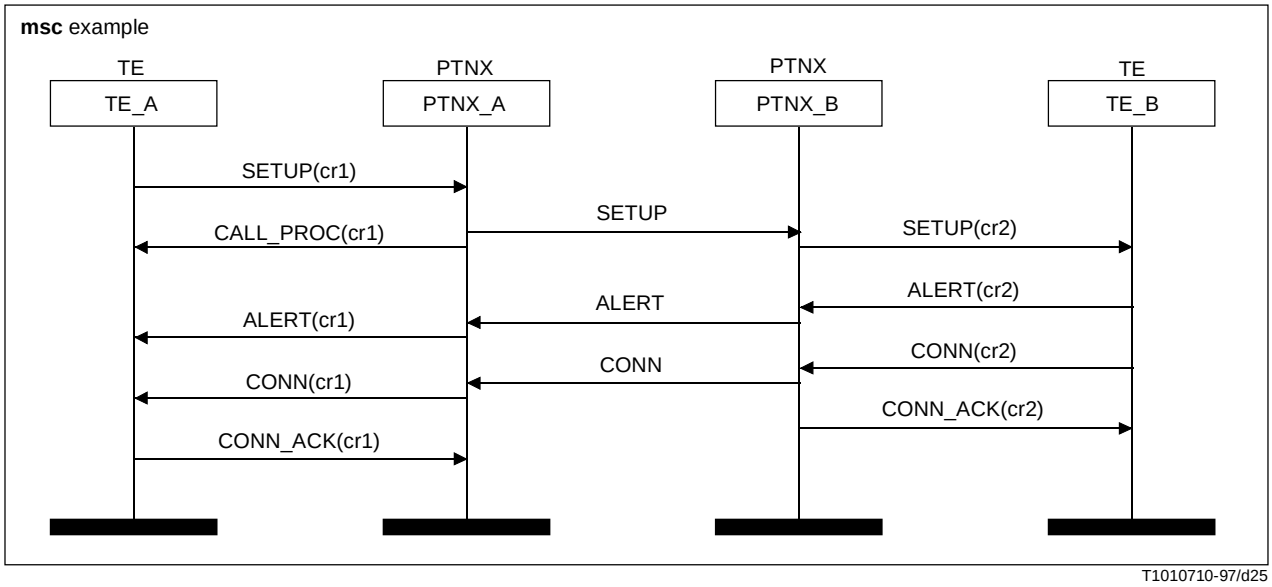


Figure 14-2/Suppl. 1 to Rec. Z.100 – En-bloc sending, successful call

14.5 Process behaviour modelling

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

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 diagrams may be informal, incomplete, inconsistent and invalid SDL. 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 State overview modelling

This modelling uses the SDL auxiliary diagrams to provide overviews of the system behaviour. Such diagrams are tree diagrams, state overview diagrams, and state signal matrix diagrams (see I.9.2/Z.100 to I.9.4/Z.100 [1]).

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

15 Formalization steps

NOTE – In all steps where a system, block, process 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 the OMT Object Model Notation has been used in Analysis, Table 3 gives a general guide for the derivation of SDL for the Object Model Notation.

Table 3/Suppl. 1 to Rec. Z.100 – Transformations of the OMT Object Model Notation into SDL

Object Model Notation	Possible SDL transformation
class	block (type), process (type), service (type), data declaration (or newtype or syntype)
abstract class	block type, process type, service type, (data) newtype (or syntype or generator)
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 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	signal route, channel
association name	name of (signal route or channel)
multiplicity of aggregations	number of instances of (blocks or processes), array (of data)
role name	gate, variable name
aggregation	structure for blocks and processes, local variables in a process, two operators (or a struct) within data

Constructs of the OMT Object Model Notation not mentioned above are not usually used.

Further guidelines can be found in [26] and [27].

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-92, the T- and L-steps should be applied at the same time as the S- and B-steps.

15.1 Structure steps (S-steps)

Structure steps are used to define in SDL the external and internal interfaces of the system and static partitioning of the system into SDL 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 system diagram with an identified name and explain the system and its relation to the environment informally in a comment within the SDL 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 a classified information. The system name and the comment description will appear in the formal SDL 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 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 system may be a closed system with no external interfaces. In this case, the SDL 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 informally as a draft design (see 14.2) and MSC can be useful in determining the boundary between the SDL system and the environment. In considering the suitability of an SDL 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 description is also considered. The normative interfaces for the application may need to be internal interfaces of the SDL system.

MSC for the system are often produced as draft designs after this step has been done. If MSC 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 system (or are provided only as informative parts of the system). It is possible to consider the system as composed of communicating SDL systems, but in this case the system should be described as single SDL system with blocks representing the communicating parts. Communicating SDL 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 package.

If the system has any timers, the unit of time needs to be defined. Normally either one unit equals 1 millisecond or one unit equals 1 second are suitable values. It is also useful to define SDL synonyms 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 defined in the system.

3) State the relation between each signal and objects external to the system.

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 into a signallist.

6) Include the signal definitions and signallists in the system diagram.

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 MSC 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.

If the three-stage methodology (see 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 signallist definitions usually are too large to include in a text symbol on the first SDL 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 diagram. Such statements should be referenced from the SDL. Shorter statements should be placed with the appropriate SDL definitions.

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

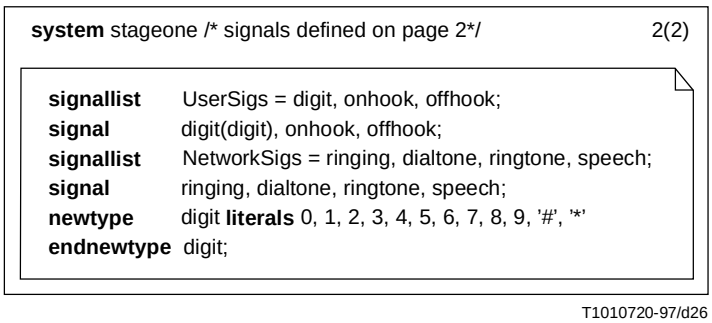
To assist documentation, signallist definitions are placed before the definition of signals used in the signallist. Data definitions are placed after signal definitions. The definitions are grouped into logical pieces by placing related definitions in one text symbol.

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



NOTE – ASN.1 has not been used for 'digit' because it was intended that the names of digits be 0, 1, 2 and so on

Figure 15-1/Suppl. 1 to Rec. Z.100 – Signallist from step S:2

15.1.3 Step S:3 – Externally identifiable parts

Instructions

1) Identify the main parts within the system and draw them as the blocks in the system.

2) Find a suitable name for each block and describe the block and its relation to its environment (its enclosing structure) informally in a comment within the block.

NOTE – Steps S:3 to S:6 are used recursively when a block is decomposed into subblocks. When the steps are reapplied, the "system" being considered is actually an SDL block, and the word "system" should be considered replaced by the phrase "enclosing block". The difference between a block and the system is that the signals (signallists and data) used to communicate with the surrounding structure are defined **outside** the block for a block and **inside** the system for a system.

Guidelines

Classified information contains component classes that correspond to the contained blocks (connected by channels) and type definitions in SDL. The component classes of the system (or block when this step is used recursively) correspond to blocks 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 blocks are handled in this step, and the channels, in step S:4. The blocks 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 as a block type, process type, procedure, signal, timer, or sort of data. In most cases the choice of an SDL type corresponding to a classified information is obvious, but in some cases it might depend on the intended abstract presentation of the formal specification. For example, object behaviour can be described in a state based way (process, procedure) or using axioms (operator and axioms in newtype definitions). In this particular case a state based description is recommended, except if the behaviour clearly corresponds to abstract data functions.

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

Blocks delimit visibility. For this reason, signals, sorts and types that are used only within a block 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 steps S:6 and S:7 for these items, and is expressed as a general principle for information hiding in Rule 18.

A block is "informative" if it has no behaviour that is to be normative. The purpose of an informative block 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 analyzed, leading to a better quality product and supporting Validation.

A block can be informative only if all the blocks, processes and procedures (including remote procedures) it encloses are informative. Once a block (process or procedure) is marked informative it is implied that all the enclosed blocks, processes and procedures are informative and it is not necessary to mark these blocks, processes and procedures as "informative".

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

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

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

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

Rule 19 – If a block (process or procedure) is informative and not itself enclosed in an informative block (process 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 blocks and possibly identification of types for these blocks.

Example

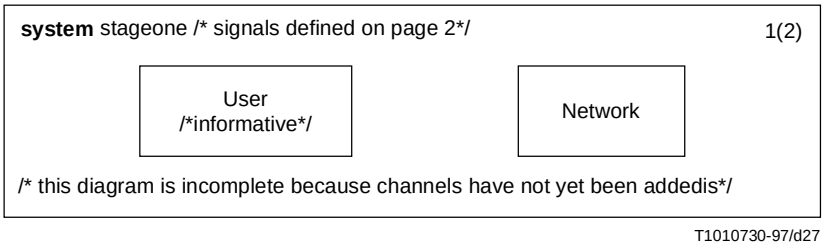


Figure 15-2/Suppl. 1 to Rec. Z.100 – The user and the network for an I.130 [7] stage 1 description

15.1.4 Step S:4 – Communication paths between parts

Instructions

1) Identify the channels needed between blocks and the boundary of the diagram and between blocks within the diagram.

2) For each channel, identify the direction(s) of communication.

3) Associate a signallist with each direction of the channel.

4) Choose a signallist name related to the function and usage of the communication.

Guidelines

The relevant description of how the internal blocks 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. A channel and its associated signallist(s) define the signature of an interface.

The signallists 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 block external to the SDL system). There can be one or more channels connecting such a point to the blocks in the SDL system diagram. 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 system contains a single block, it is connected to the environment through channels. Otherwise, the blocks at the diagram level are also connected by channels according to the flows of the information between the blocks. There are no good reasons for a block diagram to contain a single block.

Typical communication cases represented in MSC 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 blocks, then signals sent to one block arrive in the same order at the other block. 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 which has the same behaviour.

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

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 blocks.

Example

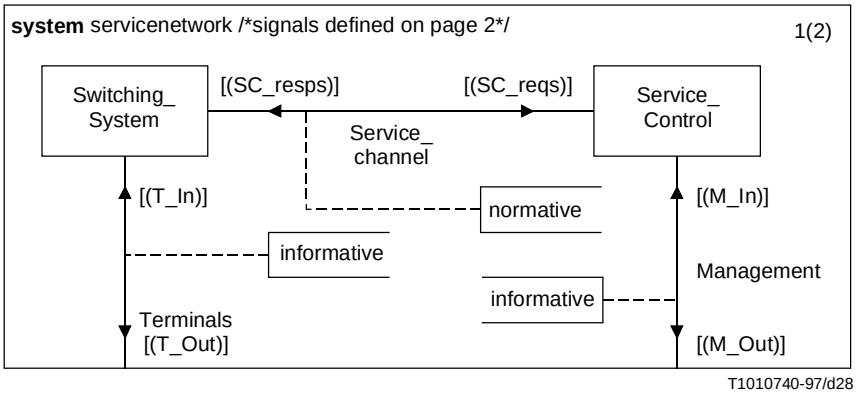


Figure 15-3/Suppl. to Rec. Z.100 – A model for a network to handle services

15.1.5 Step S:5 – Associating signals to communication paths

Instructions

1) For each signallist name, identify the appropriate signals.

2) Name and define each new signal.

Guidelines

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

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 diagram, then reconsider each signallist associated with a channel direction. Check also the messages between the relevant axes of MSC.

Consider whether to redefine the signallist associated with a channel at higher levels, making use of the new signallist. This can improve the structure and clarity of the SDL. The signallist definition needs to be within the highest level diagram in which it is used.

Although SDL 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 in a signallist.

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

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

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

15.1.6 Step S:6 – Information hiding and substructuring

Instructions

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

2) If the block 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 signallists.

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

Guidelines

If the system is large, some blocks can be considered a system on their own, and can be further partitioned according to the rules given for the system. This results in 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.

NOTE – **block** inner directly enclosed within a **block** outer is a shorthand notation for a **block** inner directly enclosed within block **substructure** outer enclosed within an otherwise empty **block** outer. See Figure 15-4.

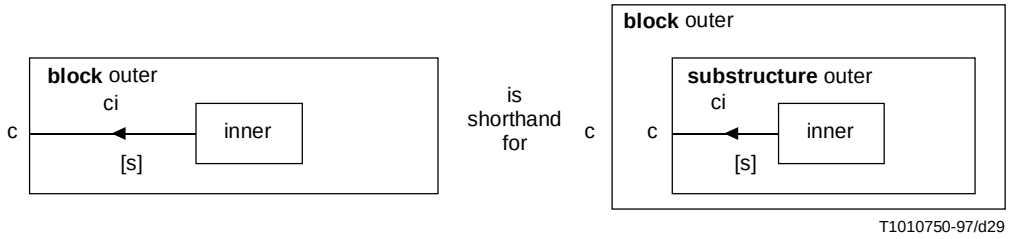


Figure 15-4/Suppl. 1 to Rec. Z.100 – Enclosed blocks shorthand

In some cases, a process will have to be encapsulated in a block to fit into the diagrams.

SDL also allows a block to contain both process descriptions and a substructure containing further blocks. It may be useful during Draft Design to use this feature, but it is not recommended to use this for Formalization.

Recursive application of this step 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 system.

Enclosed blocks should be drawn as block references rather than as block diagrams because directly nested diagrams soon become too large and complex to handle or understand. The same principle applies to all diagrams (block, process, procedure and type diagrams).

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

A block tree diagram should be drawn.

Rule 24 – The diagrams should be nested by reference rather than direct enclosure.

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

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

15.1.7 Step S:7 – Block constituents

Instructions

1) Identify the separate objects with behaviour for the block that has been chosen in step S:6 to be divided into processes, and define these objects as the processes of the block.

2) Find a suitable name for each process and describe it and its relation to its environment (the enclosing block) informally in a comment within the process reference.

3) For each process, define its initial and maximum number of instances.

4) Use signal routes to connect the process sets to channels at the block boundary.

Guidelines

The signal routes within the block can be derived in a similar way to the channels in steps S:3 and S:4.

In the case where a process has been encapsulated in a "dummy" block, the unpartitioned block contains a process description of just one type and the relation to its environment is derived from the external interface of the block.

Consider object models with associations between classes corresponding to processes. The respective multiplicity (1:1, 1:many, many:many) of the association helps to define the initial and maximum number of instances for those processes.

Rule 26 – For each block, at least one process shall have its initial number of instances greater than zero, so that it can create other instances in the block.

Rule 27 – The number of process definitions in each block should be no more than five.

Result: Identification of processes within the leaf blocks in the block tree.

15.1.8 Step S:8 – Local signals in a block

Instructions

1) Identify the signals between the processes local to the block.

2) Define at the block level any of these signals that are additional (not defined external to the block).

3) Identify any imported and exported procedure belonging to processes in the block, and the corresponding remote procedure definition, and define these at the block (or block type) level.

Guidelines

The relevant description is in the internal interface aspects of the class corresponding to the enclosing block. If the signal routes are described as part objects, information about the signal on the signal route is given by the corresponding local signal route 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 allows three ways (other than signal passing) that the value in data variables in one process can be used in another process: view and reveal, import and export, and by calling a remote procedure in the other process. 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. View and reveal can lead to non-deterministic behaviour. Import and export can be replaced by remote procedure calls.

Rule 28 – View and reveal shall not be used.

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 behaviour of components 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 **process**, but both these steps and the data steps are also generally appropriate for defining the behaviour of an SDL **procedure**.

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

Instructions

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

– signal that can be received by the process (an 'announcement' in ODP terminology) where the process may or may not send a response;

– exported procedure of the process. This is a case where the process acts as a server in a client-server model (an 'interrogation' in ODP terminology) 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 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 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 shown on the block diagram is the process sending signals to itself or other instances of the same process definition.

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.

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 MSC for at least the "typical" use cases if these have not been produced as draft designs.

2) Produce a skeleton process by mapping from MSC 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 MSC 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. It is also possible to divide such a process into SDL services.

Guidelines

The MSC are produced as part of Draft Design.

MSC can lead semi-automatically to a skeleton process. The order of events on the vertical axis in the MSC 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 MSC and then writing the process definition from these use cases (the "simple and usual cases"). Dynamic process creation can also be deduced from the MSC.

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

Sometimes the MSC 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 request message or an earlier

related message. If the information is in another process or external to the system, then the process must communicate with the owner of the information by signals or a remote procedure. Update the MSC or annotate them as incomplete and simplified.

A skeleton process is expected to exhibit the essential behaviour of a process, but not the complete behaviour. Other draft designs (and the MSC) 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 (for example in Figure 15-5).

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

Rule 30 – The MSC either shall be correct traces of the handling of messages by the SDL system, or shall be clearly annotated how and why they differ from the SDL behaviour.

Result: The result is a skeleton process in the form of a state overview diagram including timing, and consistent typical MSC.

Example

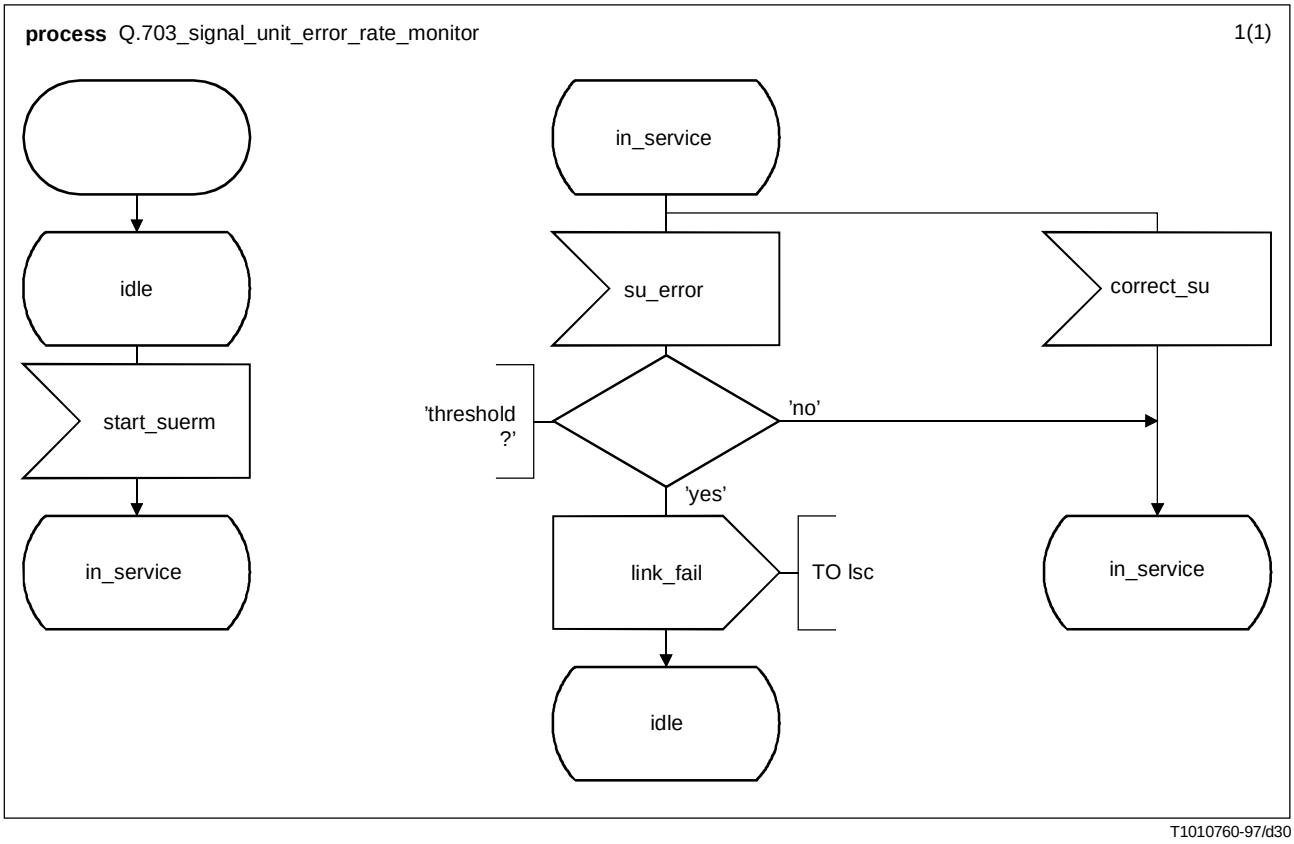


Figure 15-5/Suppl. 1 to Rec. Z.100 – A skeleton process

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) use steps B:1 to D:9 to define it.

Guidelines

Use existing MSC and generate new MSC 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: The result is an informal process that covers all the cases in the MSC.

Example

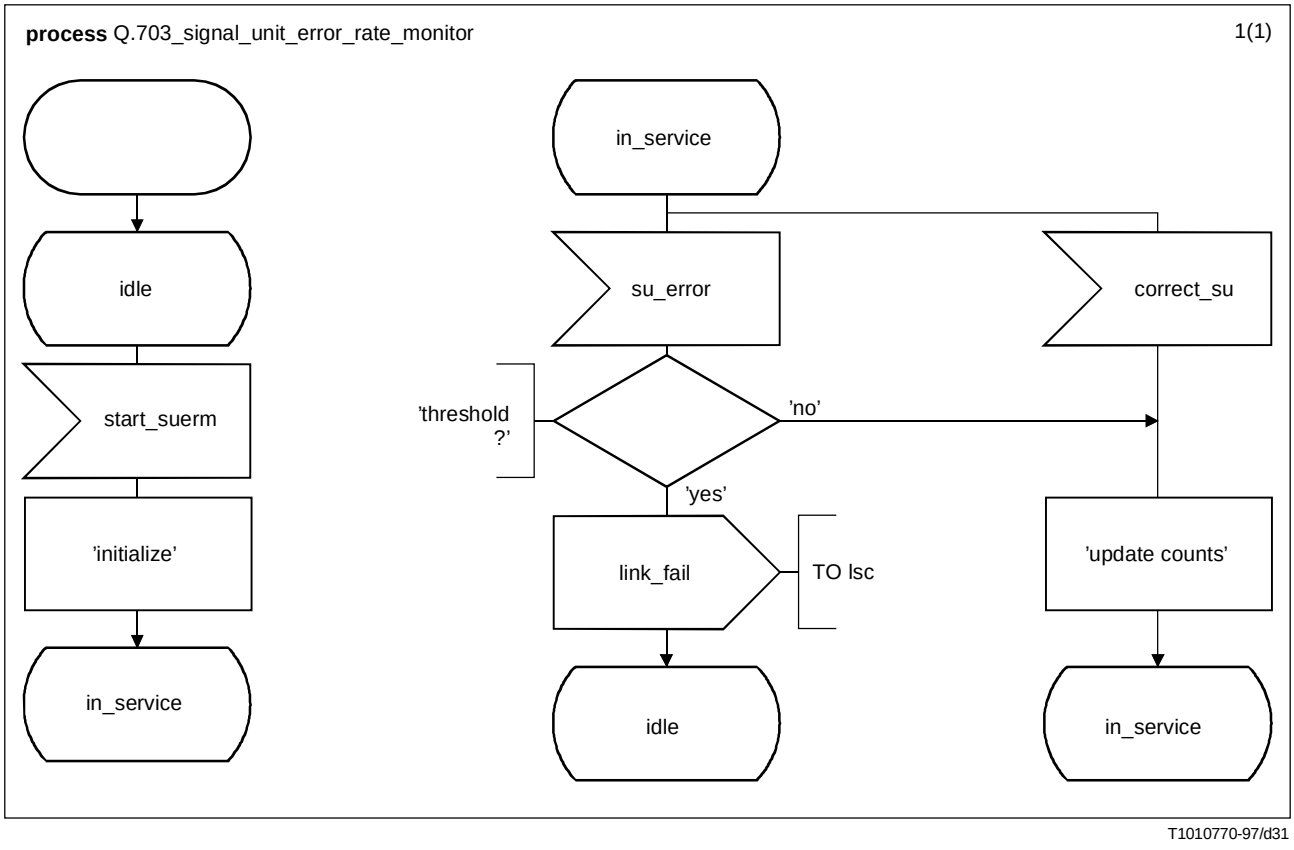


Figure 15-6/Suppl. 1 to Rec. Z.100 – The informal vesion of same example as step B:2

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

Guidelines

A state signal matrix (I.9.4/Z.100 [1]) 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.

A state overview diagram (I.9.3/Z.100 [1]) 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 analyzed 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 **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.

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

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.

15.3 Data steps (D-steps)

The purpose of the data steps is to provide a formal definition of the data used in the processes. Without formal data any decision in the behaviour 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 processes. The remaining steps (D:6 to D:9) 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:9 are only occasionally necessary and should be unnecessary if ASN.1 has been used to define data.

Step D:1 is applied to the system as a whole, whereas steps D:2 to D:9 can be applied to each process or procedure.

Rule 34 – The sorts of data used shall be defined using SDL combined with ASN.1 as defined in Recommendation Z.105 [2]).

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.

Example

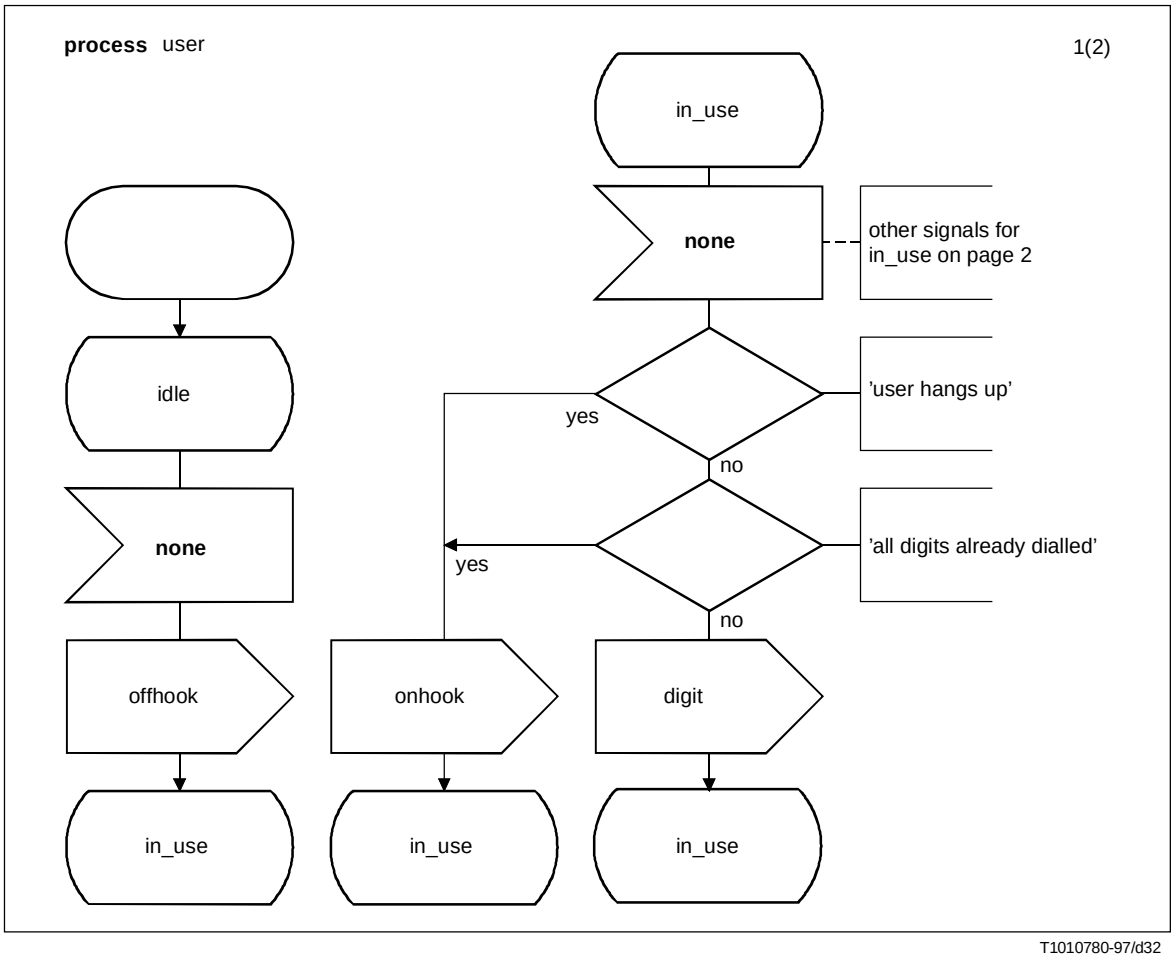


Figure 15-7/Suppl. 1 to Rec. Z.100 – Part if a complete but informal process

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. ASN.1 should be used if the encoding of data is important or if the data is on an interface with the environment. SDL may be used if no particular encoding is required and if the sort of data is used only within the SDL system.

Rule 35 – If bit tables have been used to define data, then these shall be converted to ASN.1 or SDL.

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 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 the process (or procedure).

2) State the role of each parameter in a comment in the heading of the process (or procedure).

Guidelines

Within a process, a parameter is treated as a variable. The only difference between a process parameter and a variable with a default value is that a process parameter can receive a different value for each instance of the process. 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 – A process parameter should not contain the parent Pid as this is available as the ***parent****.*

Result: The process (or procedure) 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 according to procedure 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 is treated in a similar way to a process through steps B:1 to D:9, including the possibilities of having a procedure called internally and a procedure nested within the procedure.

When the normative behaviour is independent of some data in informative processes or procedures, the **any** construct can be used to create random values for this data or random decisions in informative processes 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 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

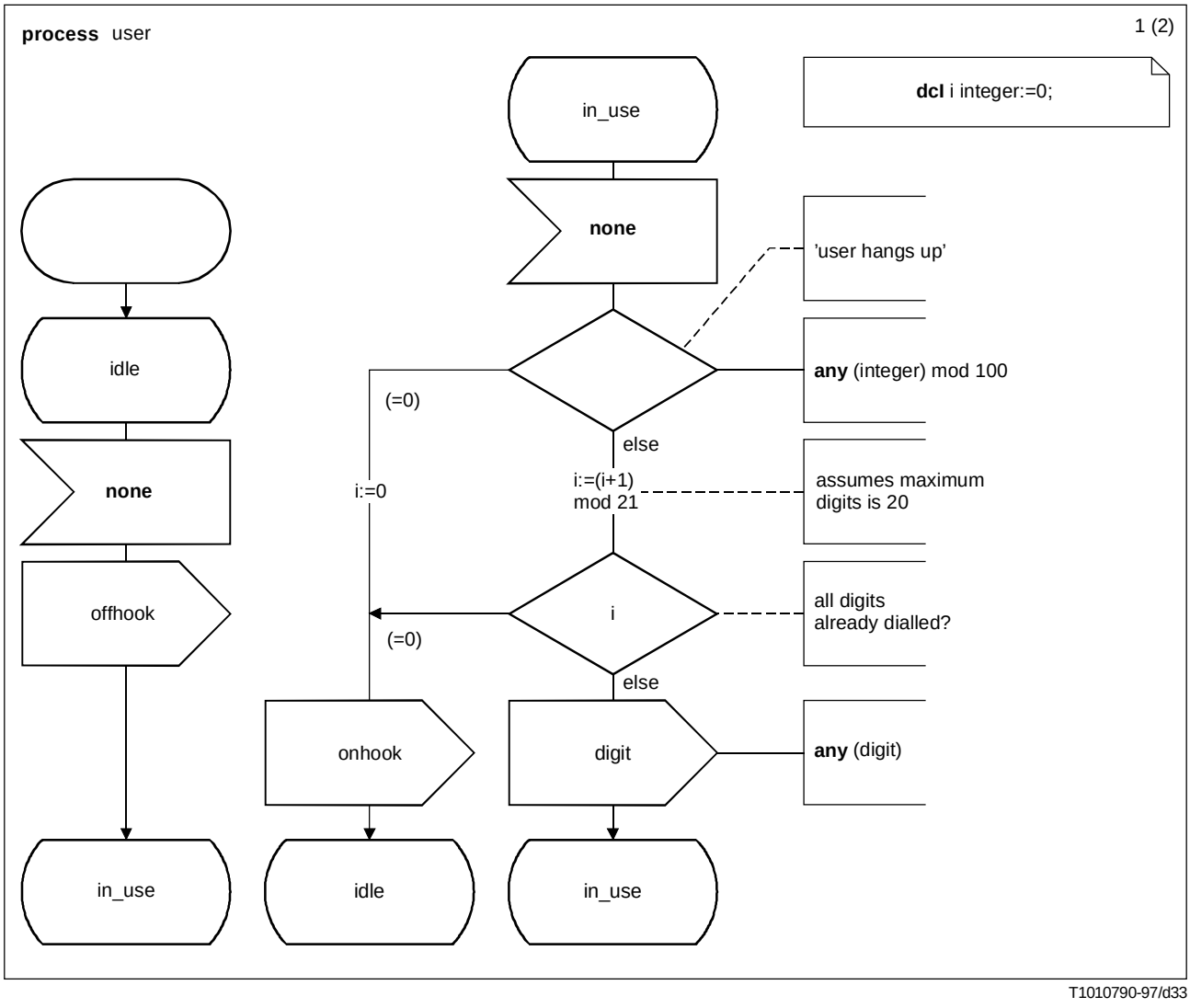


Figure 15-8/Suppl. 1 to Rec. Z.100 – Part of a complete but informal process for a user

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 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) which 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 create a newtype or syntype with a meaningful name or define an ASN.1 type.

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 newtypes 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 processes, 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 Z.100 [1] predefined data, ASN.1 "useful types" and Z.105 [2] predefined data. The objective is to avoid having to define the behaviour for new operators. If new operators are defined, they should be defined using an operator definition.

The introduction of new operators can be avoided by:

– using the ASN.1 definitions as defined in Recommendation Z.105 [2];

– using **struct** for records;

– using a predefined **generator** such as array or string;

– 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;

– using a user-defined **generator** that avoids defining axioms by using other generators.

The operators are usually listed in the newtype 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 newtype 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 using SDL axioms.

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 description, if new operators are introduced, the functionality may not be what was intended, because the user defined sorts of data for these operators may have "too many" values. Step D:7 corrects this deficiency, but makes interpretation depend on informal text. Steps D:8 and D:9 make 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 axioms in the form of informal text to the newtype definitions.

Guidelines

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

Although the result is correct SDL, only the static properties can be completely analyzed. The reason is that the dynamic properties depend on the interpretation of the axioms, which can only be done informally. However, in an actual support environment, some additional features or assumptions can make this level of description sufficient.

Result: The informal axioms provide a record of what is intended by the newtype definition, without formally defining it.

15.3.8 Step D:8 – Formal data description

Instructions

1) For each operator signature, add an operator definition in the form of an operator diagram if possible.

   NOTE – If all the operators can be defined in this way, the formalization is complete.

2) If some of the operators cannot be defined by operator diagrams, formalize the axioms by replacing informal axioms with formal ones that state the essential properties of the operators.

3) Use the text of informal axioms as comments to the formal axioms.

Guidelines

Operators can be defined algorithmically using an operator diagram, or axiomatically. The operator diagram (algorithmic) form is recommended because it is easier to understand. The axiomatic method is described below.

Although the axiomatic method has a better mathematical basis and is used within Recommendations Z.100 [1] and Z.105 [2], it is more difficult to write correct axioms than to construct an operator diagram that does what is wanted. Many engineers seem to find the definition of data types by axioms difficult and error prone. Axioms are difficult for tools to handle, so that good tool support is more difficult to find. Axioms should therefore not be used without either experience or guidance from an experienced user. To apply an operator in SDL it is sufficient to have the signature defined.

To specify operator properties with axioms, specify the properties of constructors first. This gives the possible values of the sort. Then specify equations for the remaining operators and literals. It is fairly easy to state the essential properties, but usually difficult to make the axioms complete and to make sure they do not conflict (see step D:9).

These axioms can be regarded as informative. Unless it is expected that the axioms are used as the basis of some automated implementation, the formalization can be treated as completed by placing a comment in the axioms that they are informative, have not been proven and may be incomplete. In this case, step D:9 shall be omitted.

Rule 42 – SDL axioms should not be used to define operator properties.

Result: The result is that some of the properties of the data types have been defined, but probably not all of them, so that, formally, each sort has many (usually infinitely many) more values than intended.

15.3.9 Step D:9 – Complete data formalization

NOTE – This step should be used only in exceptional circumstances.

Instructions

Add axioms (or operator definitions) to the data newtype definitions, until they are complete (that is, until all expressions containing non-constructor operators and literals can be rewritten into expressions containing only constructor operators and literals).

Guidelines

Detailed guidelines for this step are given in I.5/Z.100 [1].

Avoid defining constructors (that is, operators that create values of a sort of data). Instead, ensure that there is a literal for each (and every) value of a sort. Assuming **noequality** is not used, every literal defined has a value distinct from the value of any other literal.

In the case that new operators are required which entirely make use of sorts defined in enclosing scope units, define the operator in a dummy newtype.

Never (intentionally or unintentionally!) change the number of values of a sort by using axioms in the newtype of a different sort.

Result: If all the above steps have been followed completely, at this stage there will be a completely formal specification in SDL, 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, it is recommended that in larger systems they are applied in parallel with the preceding steps. 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 and steps T:3 to T:5 may be considered optional. A good knowledge of SDL-92 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 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 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 blocks with the same behaviour. The definition must be repeated for both blocks unless a block type is used, in which case the definition of the block type can be used for both blocks.

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 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 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

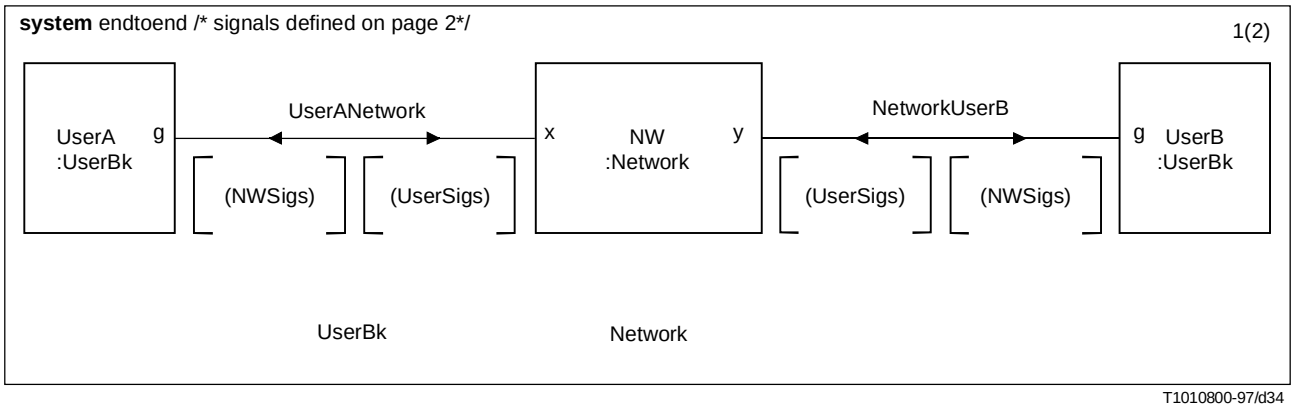


Figure 15-9/Suppl. 1 to Rec. Z.100 – Asystem definition using types

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 which **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 super-type of the subtype). Specialization can be done in two ways: adding properties to the super-type or changing some of the properties of the super-type in the subtype. This step considers adding properties to the super-type.

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 and blocks can be added to a system type or a block type enclosing blocks. Signal routes, process (type) definitions, procedures and signal definitions can be added to a block type enclosing processes. New transitions for new signals can be added to a process type and the transitions can lead to new states.

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

Example

Two processes(suerma and suerm_supertype) are identical except that one 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 for the second process can be a subtype for the first process (see Figure 15-10).

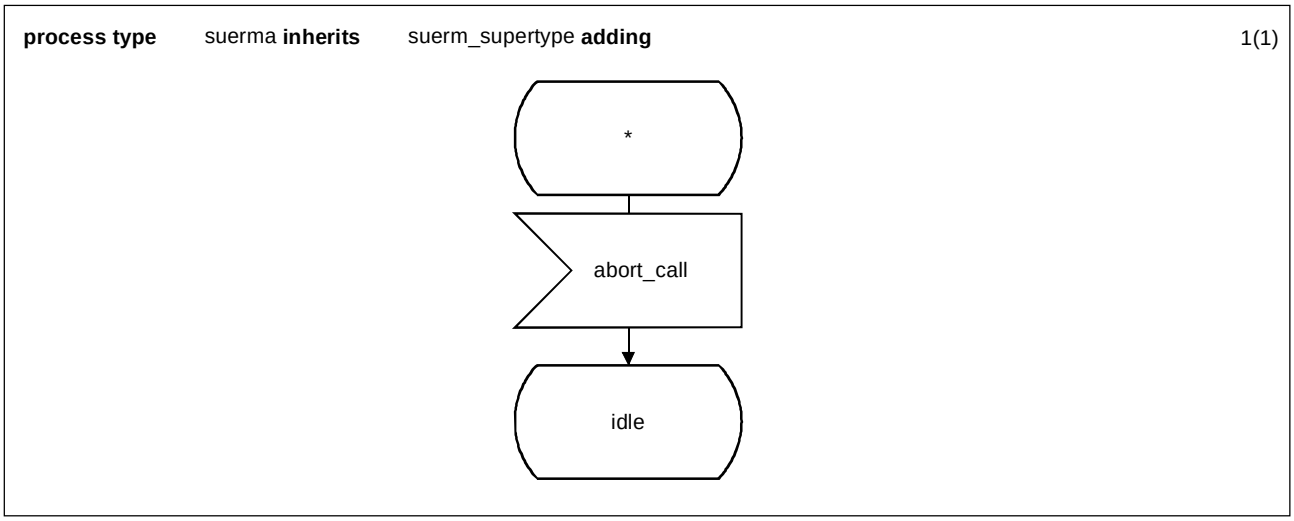


Figure 15-10/Suppl. 1 to Rec. Z.100 – A subtype created by adding a property

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

Instructions

1) If there are no types which have almost the same internal structure and behaviour, but handle slightly different situations (they are "similar"), then the step is complete.

2) 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 super-type to be used for the other types that become subtypes when step T:5 is applied.

4) If the general super-type is a block type, 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 super-type.

5) If the general super-type is a process type or procedure, 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 super-type, consider whether they should be virtual.

6) For each virtual type in the general super-type, 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 super-type defined by this step is not used within the SDL system until other steps are applied.

Guidelines

A general super-type is a super-type 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 super-type as applied to each case. The general super-type 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 super-type 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 that are identical except for the transition for one signal in one state. The super-type is then the process definition in which either this transition is virtual or the transition contains a call of one or more virtual procedures.

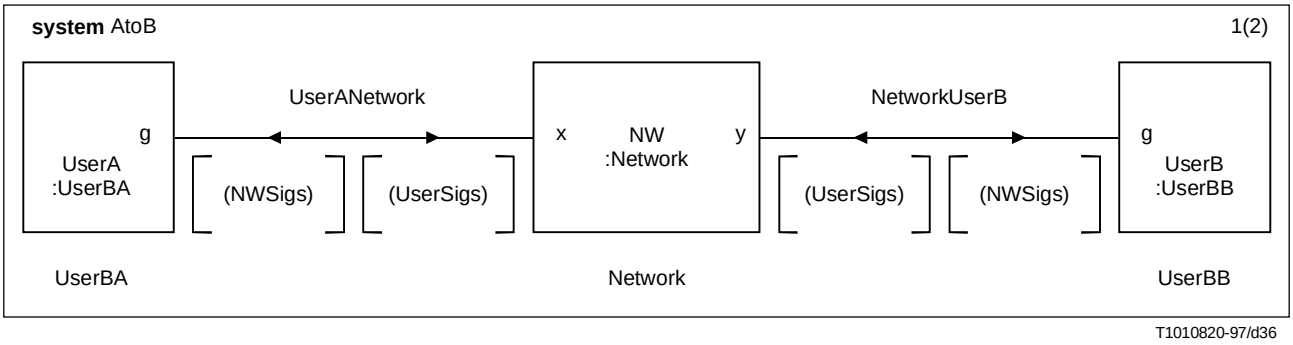


Figure 15-11/ Suppl. 1 to Rec. Z.100 – The AtoB system

As another example, assume that an "AtoB" system represents communication from A to B through a network. A and B are almost identical blocks containing one process; each called User. The User process is slightly different in each case. The general super-type can be a block type UserBk.

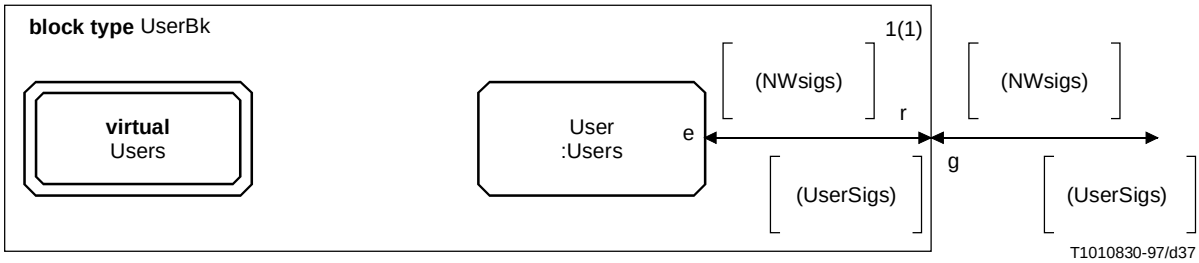


Figure 15-12/Suppl. 1 to Rec. Z.100 – The general super block type for the "users"

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 Users. The block types UserBA and UserBB can be defined as inheriting the general super-type UserBk and redefining the virtual process Users.

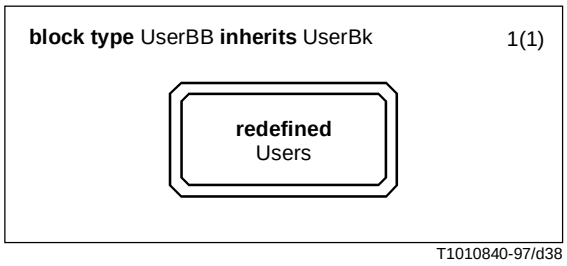


Figure 15-13/Suppl. 1 to Rec. Z.100 – The definition of UserBB using general super-type UserBk

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 super-type.

4) Define a **redefined** type (based on the virtual constraint type) for each case where the virtual type will be used.

Guidelines

A general super-type encloses virtual types that can be different each time the general super-type is used to define a subtype. There needs to be a definition of the virtual type enclosed in the general super-type, 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 super-type 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 must inherit the constraint type.

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 of:

– *the parameters of a virtual procedure so that all calls have the same number of parameters of the same type;*

– *parameter, gates and common behaviour for a virtual process;*

– *gates and internal blocks or processes connected to the gate for a virtual block.*

Result: A type definition (the "constraint type") for each virtual type in the general super-type, and a subtype (of the constraint type) for each actual behaviour.

Example

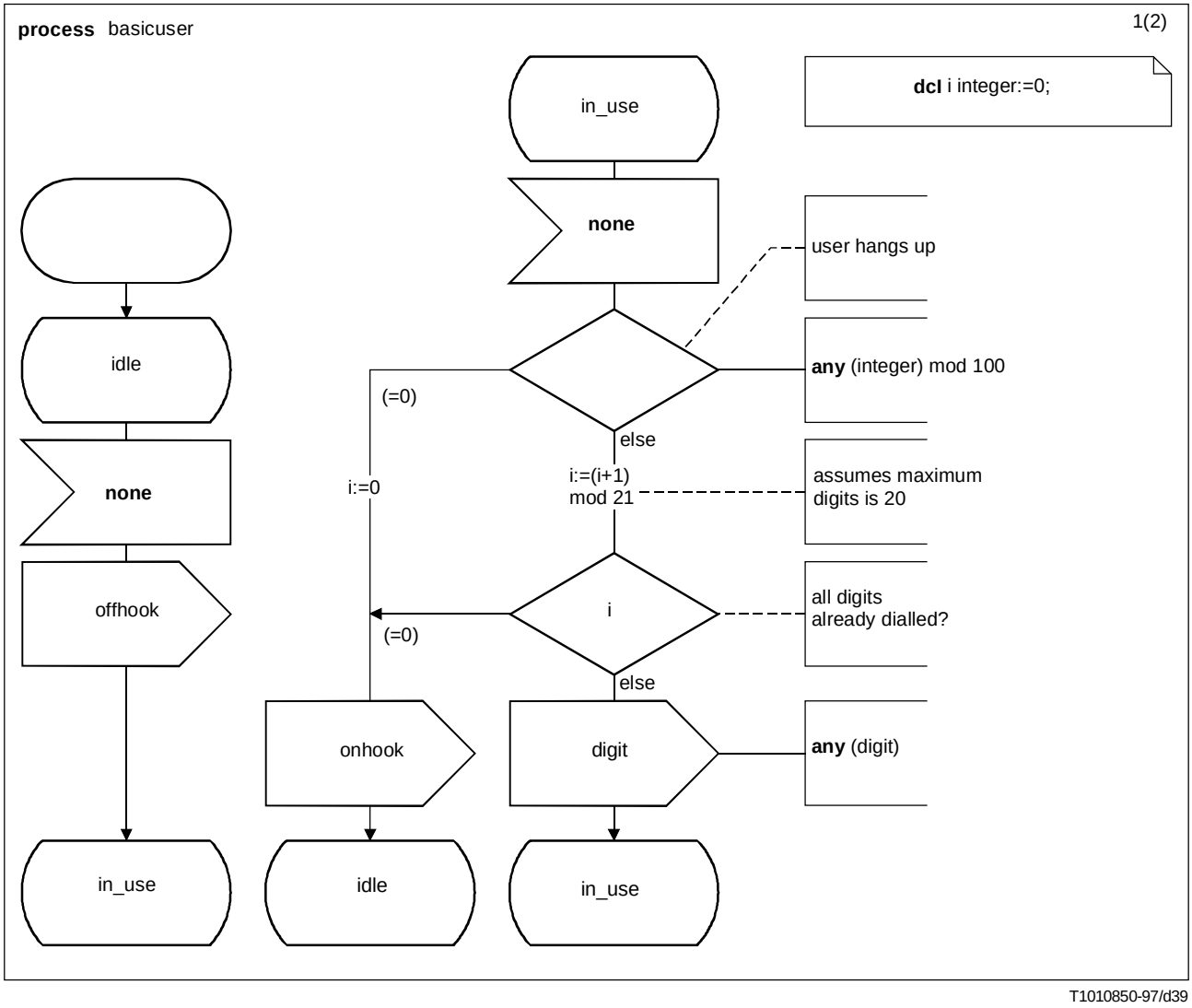


Figure 15-14/Suppl. 1 to Rec. Z.100 – The basicuser process type

It is assumed that every time UserBk is used, the Users process type has at least the properties of a basicuser. The definition of the virtual process type for Users in the general super block type UserBk is then defined to be **atleast** basicuser.

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 super-type, and in this subtype.

2) Redefine virtual transitions for the context.

3) 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.

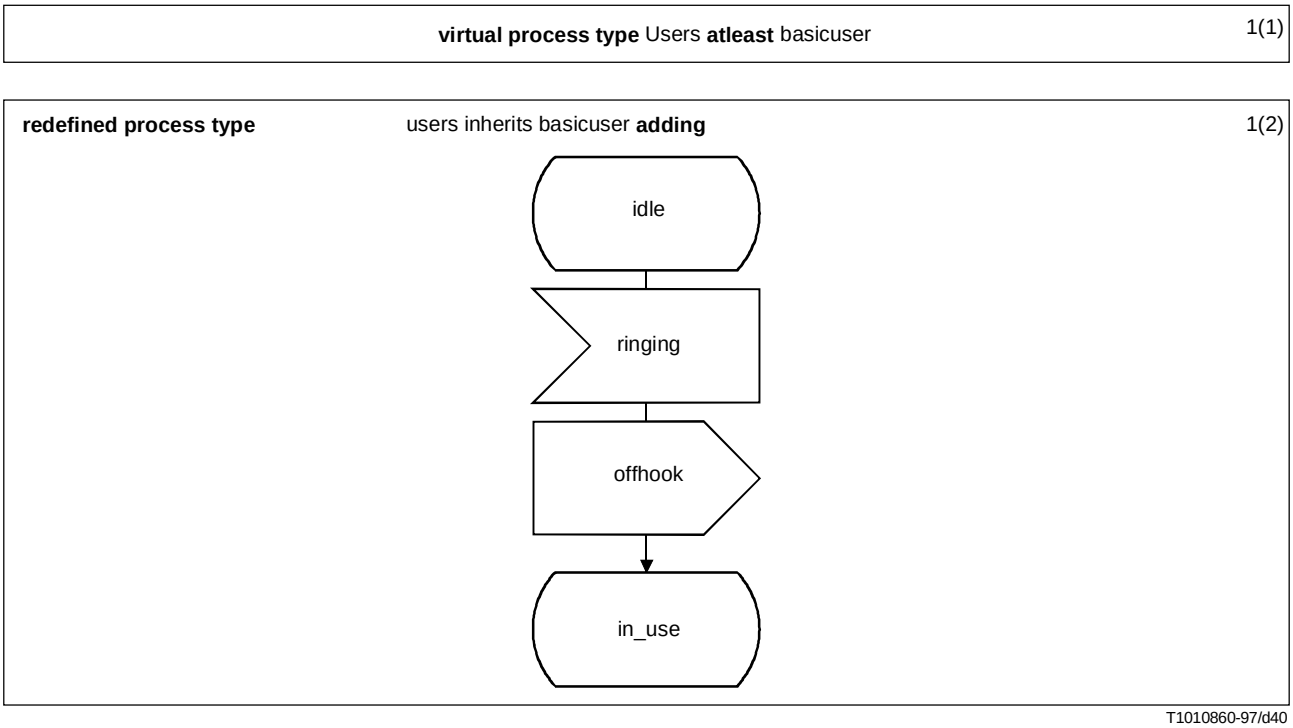
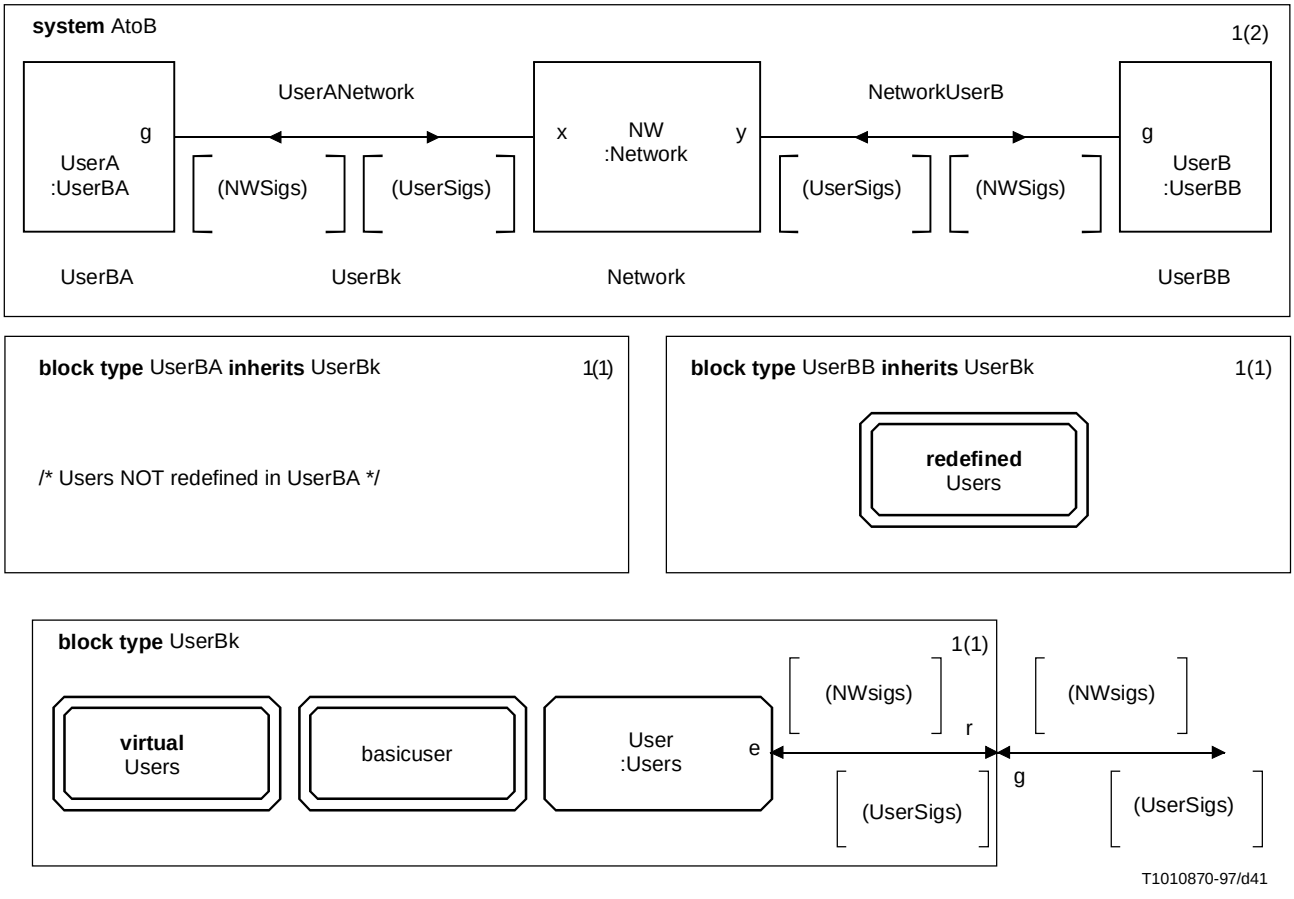


Figure 15-15/Suppl. 1 to Rec. Z.100 – The virtual process type Users and a redefinition of Users

Result: A system based on type definitions.

Examples



NOTE – As UserBA is the same as UserBk, UserBA could be eliminated. Similarly, as basicuser is used only in UserBk, the (virtual) definition of Users could also define the constraint type.

Figure 15-16/Suppl. 1 to Rec. Z.100 – The AtoB system (with a general super block type UserBk

15.5 Localization steps (L-steps)

The purpose of localization steps is to ensure that types are defined at the best places to prevent 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 type 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, but SDL-92 only allows package references to be attached to the system diagram. The use of a package 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 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 (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 signal, timer, variable, synonym and sort context parameters, there is no choice: SDL 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 a process it is preferable to use a process context parameter of a block, if the process types for the actual parameters can be usefully defined external to the blocks. System and block cannot be context parameters.

Table 4/Suppl. 1 to Rec. Z.100 – Use of context parameters

	Can be context parameter	**Can have context parameter**	**Can be virtual type**
system, block	no	yes	yes
process	yes	yes	yes
procedure	yes	yes	yes
signal, timer	yes	yes	no
sort	yes	yes	no
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 understanding of the system is increased by 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 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 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 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 system.

16 References

[1] ITU-T Recommendation Z.100 (1993), *CCITT Specification and Description Language (SDL)*.

[2] ITU-T Recommendation Z.105 (1995), *SDL combined with ASN.1 (SDL/ASN.1).*

[3] ITU-T Recommendation Z.120 (1996), *Message Sequence Chart (MSC).*

[4] CCITT Recommendation X.208 (1988) equivalent to ISO/IEC 8824:1990, *Specification of Abstract Syntax Notation One (ASN.1).*

[5] ITU-T Recommendation X.680 (1994)/Amd. 1 (1995) (equivalent to ISO/IEC 8824-1:1996), *Information technology – Abstract Syntax Notation One* (*ASN.1): Specification of basic notation – Amendment 1: Rules of extensibility.*

[6] BRÆK (R.), HAUGEN (Ø.): Engineering Real Time Systems, Prentice-Hall, 1993.

[7] OLSEN(A.) *et al:* Systems Engineering Using SDL-92, North Holland, 1994.

[8] ITU-T Recommendations X.900 – X.905 (to be published), *Information technology – Open Distributed Processing* (ISO/IEC 10746).

[9] ITU-T Recommendation Z.500 (1997), *Framework on formal methods in conformance testing*.

[10] ITU-T Recommendations X.290 – X.292, *OSI conformance testing methodology and framework for protocol Recommendations for IUT-T Applications* (ISO/IEC 9646).

[11] RUMBAUGH (J.) *et. al*: Object-Oriented Modeling and Design, Prentice-Hall, 1991.

[12] JACOBSEN (I.) *et al*: Object-Oriented Software Engineering, Addison-Wesley, 1992.

[13] ITU-T Recommendation Q.1200 (1993), *Q-series intelligent network Recommendation structure.*

[14] ITU-T Recommendation Z.400 (1993), *Structure and format of quality manuals for telecommunications software*.

[15] HOGREFE (D.): Validation of SDL Systems, Computer Networks and ISDN Systems, North Holland, 1996.

[16] CCITT Recommendation I.130 (1988), *Method for the characterization of telecommunication services supported by an ISDN and network capabilities of an ISDN*.

[17] ITU-T Recommendation Q.65 (1997), *The unified functional methodology for the characterization of services and network capabilities.*

[18] BELINA (F.), HOGREFE (D.), TRIGILA (S.): Modelling OSI in SDL (in Turner: Formal Description Techniques), North-Holland, 1988.

[19] BELINA (F.), HOGREFE (D.), SARMA (A.): SDL with Applications from Protocol Specification, Prentice Hall, 1991.

[20] ITU-T Recommendation X.200 (1994), *Information Technology – Open Systems Interconnection – Basic Reference Model: The Basic Model.*

[21] ITU-T Recommendation X.210 (1993), *Information Technology – Open Systems Interconnection – Basic Reference model: conventions for the definition of OSI services.*

[22] CCITT Recommendation X.219 (1988), *Remote Operations: Model, notation and service definition*.

[23] CCITT Recommendation X.722 (1992), *Information Technology – Open Systems Interconnection – Structure of Management Information: Guidelines for the definition of managed objects.*

[24] REED (R.) (editor): Specification and Programming Environment for Communication Software, North Holland, 1993.

[25] OLSEN (A.) *et al*: Systems Engineering Using SDL-92, North Holland, 1994.

[26] WITASZEK (D.) *et al*: A Development Method for SDL-92 Specifications based on OMT, in SDL '95 with MSC in CASE, *Proceedings of the Seventh SDL Forum*, North Holland, 1995.

[27] GUO (F.), MACKENZIE (T.W.): Translation of OMT to SDL-92, in SDL '95 with MSC in CASE, *Proceedings of the Seventh SDL Forum*, North Holland, 1995.

ITU-T RECOMMENDATIONS SERIES

Series A	Organization of the work of the ITU-T
Series B	Means of expression: definitions, symbols, classification
Series C	General telecommunication statistics
Series D	General tariff principles
Series E	Overall network operation, telephone service, service operation and human factors
Series F	Non-telephone telecommunication services
Series G	Transmission systems and media, digital systems and networks
Series H	Audiovisual and multimedia systems
Series I	Integrated services digital network
Series J	Transmission of television, sound programme and other multimedia signals
Series K	Protection against interference
Series L	Construction, installation and protection of cables and other elements of outside plant
Series M	TMN and network maintenance: international transmission systems, telephone circuits, telegraphy, facsimile and leased circuits
Series N	Maintenance: international sound programme and television transmission circuits
Series O	Specifications of measuring equipment
Series P	Telephone transmission quality, telephone installations, local line networks
Series Q	Switching and signalling
Series R	Telegraph transmission
Series S	Telegraph services terminal equipment
Series T	Terminals for telematic services
Series U	Telegraph switching
Series V	Data communication over the telephone network
Series X	Data networks and open system communication
Series Z	**Programming languages**