EUROPEAN TELECOMMUNICATION STANDARD

ETS 300 414

December 1995

Source: ETSI TC-MTS

Reference: DE/MTS-00011

ICS: 33.020

Key words: SDL, MSC, ASN.1, specification, testability, formal validation

Methods for Testing and Specification (MTS); Use of SDL in European Telecommunication Standards; Rules for testability and facilitating validation

ETSI

European Telecommunications Standards Institute

ETSI Secretariat

Postal address: F-06921 Sophia Antipolis CEDEX - FRANCE
Office address: 650 Route des Lucioles - Sophia Antipolis - Valbonne - FRANCE
X.400: c=fr, a=atlas, p=etsi, s=secretariat - **Internet:** secretariat@etsi.fr

Tel.: +33 92 94 42 00 - Fax: +33 93 65 47 16

New presentation - see History box

Whilst every care has been taken in the preparation and publication of this document, errors in content, typographical or otherwise, may occur. If you have comments concerning its accuracy, please write to "ETSI Editing and Committee Support Dept." at the address shown on the title page.

Contents

Blank page

Foreword

This European Telecommunication Standard (ETS) has been produced by the Methods for Testing and Specification (MTS) Technical Committee of the European Telecommunications Standards Institute (ETSI).

Transposition dates	
Date of adoption of this ETS:	27 October 1995
Date of latest announcement of this ETS (doa):	31 March 1996
Date of latest publication of new National Standard or endorsement of this ETS (dop/e):	30 September 1996
Date of withdrawal of any conflicting National Standard (dow):	30 September 1996

Introduction

Specification, validation and testing

The growing complexity of telecommunication systems, requires advanced methods for design, implementation, and testing. Errors or omissions in telecommunication standards are often costly to correct. Moreover, if errors are not detected, and permeate in telecommunication products, they may lead to loss of property or revenue. Therefore, validation of telecommunication standards and testing of products that claim to implement a standard, is of extreme importance.

International standardisation organisations have developed formal description techniques, such as Specification and Description Language (SDL) (ITU-T Recommendation Z.100 [1]). The essence of these techniques is that they have a mathematical basis. This facilitates precision and has enabled development of computer-based tools that support their use. SDL tools are commercially available. These tools typically include facilities to edit SDL diagrams, to check consistency and completeness of the specifications, to run a simulation of the specified system, to validate the specifications and to generate tests from the specifications. Application of formal description techniques in specifications is a first and essential step towards improvement of validation of standards and easier testing of telecommunication products.

This ETS specifies how to use the ITU-T Recommendation Z.100 [1] SDL in ETSs. The use of SDL diagrams in an ETS and SDL tools in the development process of standards, will add to the clarity, non-ambiguity, and consistency of telecommunication standards:

clarity: use of standard diagrams known to telecommunication experts, instead of own inventions;

non-ambiguity: the meaning of SDL diagrams is determined completely by the formal semantics described in ITU-T Recommendation Z.100 [1], and implemented in simulation tools;

consistency: the use of tools allows thorough cross checking of system diagrams, block diagrams, process diagrams and message sequence charts.

Applications of SDL in telecommunication standards include (see ITU-T Recommendation Z.100 [1], §§ 1.1.2):

- call processing;
- maintenance and fault treatment;
- system control;
- operation & network management;
- data communication protocols;
- telecommunication services.

Formal techniques should be used in ETSs in combination with established and proven specification methods. SDL diagrams need to be used to specify those parts of a telecommunication system, that need to be defined precisely and unambiguously. Message Sequence Charts (MSCs) (ITU-T Recommendation Z.120 [3]) need to be used to illustrate the functions of a system. Normal text, tables and informal figures can be used in combination with SDL diagrams. Furthermore, Abstract Syntax Notation One (ASN.1) (CCITT Recommendation X.208 [4]) is recommended for use in combination with SDL and MSC diagrams. This recommendation is based on practical experience in application of SDL by telecommunication manufacturers and telecommunication operators.

If both SDL diagrams and MSCs are used in an ETS, a certain amount of redundancy in the information provided cannot be avoided. This is not necessarily harmful; redundancy is beneficial for human understanding. Duplication of information should, however, be avoided. It should be assumed that a telecommunication expert is able to "read" SDL diagrams, MSCs and ASN.1 type definitions. Therefore, there is no need to explain in normal text what can be read in a diagram. There is an analogy here with the situation in construction companies. Every professional in such a company can read the drawings of an architect. As a rule, explanatory text is not provided for floor plans or drawings of construction details.

The rules presented in this ETS make the specification more suitable as a basis for formal validation and testing, without paying attention to how every requirement in the specification should be validated or tested.

Furthermore, if the rules are followed, the specifications will be more suitable for application of automated test generation techniques. This may lead to important cost reduction in the development and maintenance of conformance tests.

About this ETS

This ETS is structured as follows:

Clauses 1 to 4 contain the scope of this ETS, the normative references and definitions, and symbols and abbreviations that are used in this ETS.

Clause 5 is the conformance clause. In this clause, the requirements, which ETSs that claim to conform to this ETS should meet are defined.

Clause 6 contains informative text on validation and testing, which provides a rationale for the selection of concepts and restrictions of the use of concepts by rules.

Clause 7 describes on a global level, how SDL diagrams are to be used in ETSs.

Clauses 8, 9, and 10 list, in more detail, which concepts of SDL, MSC and ASN.1 are selected for use in ETSs. Moreover, rules are provided that restrict the use of these concepts. Clauses 8, 9 and 10 are normative, the rules are to be considered as requirements on ETSs.

Annex A is normative, all other annexes are informative.

- Annex A presents an overview of the selected SDL, MSC and ASN.1 concepts;
- Annex B gives a detailed motivation for the exclusion of certain SDL, MSC, ASN.1 concepts;
- Annex C gives a collection of examples of good use of SDL;
- Annex D contains a bibliography of books and reports on SDL, MSC, ASN.1, conformance testing, and formal validation;
- Annex E contains an index to this ETS, including a graphical index of SDL symbols.

Intended audience

This ETS is written for telecommunication experts that are familiar with SDL, MSC and ASN.1. For tutorial introductions to these formalisms a number of text books are available. A selection of these books is listed in annex D.

This ETS is intended to be used in the production of ETSs. However, its use is not necessarily restricted to ETSs.

1 Scope

This European Telecommunication Standard (ETS) specifies rules for the use of the Specification and Description Language (SDL), defined in ITU-T Recommendation Z.100 [1] and Message Sequence Charts (MSC), defined in ITU-T Recommendation Z.120 [3], in ETSs. Object oriented extensions in SDL are not considered. It is intended that SDL and MSC diagrams will be used in ETSs, in combination with text, informal figures and tables. SDL diagrams are to be used to formalise those parts of an ETS that need to be defined precisely and unambiguously.

Furthermore, it is recommended to use the Abstract Syntax Notation One (ASN.1), formalism defined in ITU-T Recommendation X.680-X683 [7], that is related to SDL. Therefore, this ETS also specifies rules for the use of ASN.1, when used in combination with SDL.

All rules are aimed at improving the possibilities to validate a standard in an early phase of development, and to improve the possibilities to test products that claim to implement a standard.

This ETS does not contain any changes to the (formal) semantics of SDL, MSC or ASN.1. The use of SDL and related formalisms, enforced by the rules in this ETS, fully conforms to the language definitions of SDL, with exception of the use of ASN.1 within SDL diagrams.

This ETS does not provide a methodology for the production of ETSs which contain SDL.

This ETS does not restrict the use of ASN.1 in ETSs, if ASN.1 is used otherwise than in combination with SDL diagrams.

This ETS is not aimed at the use of SDL for product design and development. The subset of SDL that is selected in this ETS is tailored to standardisation, where clarity is of the utmost importance.

This ETS is not applicable to the specification of conformance test standards.

2 Normative references

This ETS incorporates by dated or undated reference, provisions from other publications. These normative references are cited at the appropriate places in the text and the publications are listed hereafter. For dated references, subsequent amendments to or revisions of any of these publications apply to this ETS only when incorporated in it by amendment or revision. For undated references the latest edition of the publication referred to applies.

[1] ITU-T Recommendation Z.100 (1993): "Specification and description language (SDL)".

[2] ITU-T Recommendation Z.105 (1994): "SDL combined with ASN.1 (SDL/ASN.1)".

[3] ITU-T Recommendation Z.120 (1993): "Messages sequence charts".

[4] CCITT Recommendation X.208 (1988): "Specification of Abstract Syntax Notation One (ASN.1)".

[5] CCITT Recommendation X.209 (1988): "Specification of basic encoding rules for Abstract Syntax Notation One (ASN.1)".

[6] CCITT Recommendation X.229 (1988): "Remote operations: Protocol specification".

[7] ITU-T Recommendations X.680-X.683 (1994): "Abstract Syntax Notation One (ASN.1)".

[8] CEN/CENELEC IR (1989): "Part 3: Rules for the drafting and presentation of European Standards (PNE - Rules)".

[9] ISO Standard 9646-1 (1992): "Information technology - Open Systems Interconnection-Conformance testing methodology and framework - General concepts”.

[10] ISO Standard 9646-3 (1992): "Information technology - Open Systems Interconnection-Conformance testing methodology and framework - The Tree and Tabular Combined Notation (TTCN)”.

[11] ETS 300 175 (1992): "Radio Equipment and Systems (RES); Digital European Cordless Telecommunications (DECT) Common interface".

3 Definitions

For the purposes of this ETS, the following definitions apply:

conformance testing: Testing the extent to which an implementation under test satisfies both static and dynamic conformance requirements, consistent with the capabilities stated in the implementation conformance statement (ISO 9646-1 [9], subsections 3.4.10, and 3.5.6).

conformance requirement: Statement in a standard that shall hold in a product that claims to conform to the specification.

black box testable requirement: A conformance requirement that can be tested by applying tests at the normative interfaces of an implementation.

external validation: Validation of the consistency of a standard with other related standards.

implementation conformance statement: A document supplied by the manufacturer of a product that defines which standards are claimed to be implemented and which implementation options in the standards are supported.

implementation option: A statement in a standard that may or may not be supported in an implementation. If it is supported, the statement shall hold in the implementation.

internal validation: Validation of the internal correctness of a standard.

invalid event: The reception or sending of a signal that is not listed in the signallist of a channel or signal route.

normative interface: A physical or software interface of a product on which requirements are imposed by a standard.

normative channel: A channel in an SDL specification that models a normative interface.

testability: A specification is testable if it unambiguously defines the allowed implementation options, the conformance requirements, and the normative interfaces, and if all conformance requirements can be checked, up to a certain degree of confidence, by applying tests at the normative interfaces of a product.

validation: The process, with associated methods, procedures and tools, by which evidence is given that a standard actually can be implemented and is able to provide, when correctly implemented, the intended level of functionality and performance at minimal cost.

formal validation: Automatic analysis of specifications to determine whether or not they possess certain desirable properties.

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

state space: The collection of all states of a system that can be reached from the initial state.

4 Symbols and abbreviations

For the purposes of this ETS, the following symbols and abbreviations apply:

APDU	Application Protocol Data Unit
ASN.1	Abstract Syntax Notation No. one
DECT	Digital European Cordless Telecommunications
ICS	Implementation Conformance Statement
MSC	Message Sequence Chart
OSI	Open systems Interconnection
PCO	Point of Control and Observation
PDU	Protocol Data Unit
PICS	Protocol Implementation Conformance Statement
PId	Process Instance Identifier
SDL	Specification and Description Language
TTCN	Tree and Tabular Combined Notation
TPDU	Transport Protocol Data Unit

5 Conformance to this ETS

An ETS conforms to this standard if all rules stated in clauses 8 and 9 are adhered to.

If an ETS uses ASN.1 in combination with SDL, also the rules in clause 10 can apply. Alternatively, an ETS can use the SDL concepts for definition of data types and values, in which case the rules in clause 10 are not applicable.

6 Testing and validation

6.1 Validation of specifications

Validation can be defined as follows:

> "Validation of a standard is the process, with associated methods, procedures and tools, by which evidence is given that a standard actually can be implemented and is able to provide, when correctly implemented, the intended level of functionality and performance at minimal cost".

This is a very broad definition, which covers validation approaches, ranging from automatic deadlock detection in a formal model of the standard, up to field trials of telecommunications products. A formal approach to analyse a formal model puts the burden on the standardization process itself; the implementation and field trials on the manufacturers, if they are prepared to implement a standard and subsequently perform interoperability tests. The latter type of validation may, however, take considerable time before the results are available and this may not be timely for standards development.

Furthermore, a standard should not only be correct in itself but it should also be consistent with other related standards (for example the standard containing a protocol specification needs to be consistent with the standard containing the test suite for that protocol). The terms internal validation and external validation differentiate between these two aspects of validation.

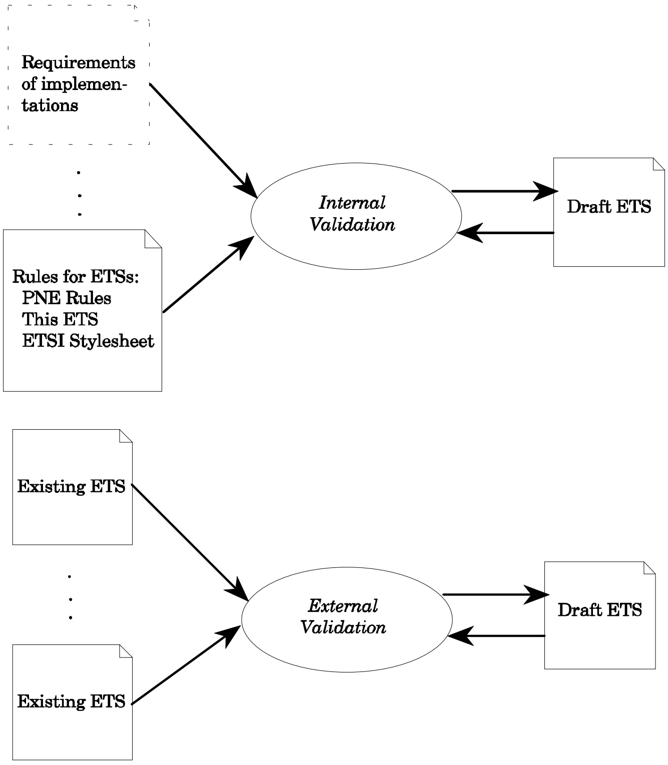


Figure 1: Internal and external validation of a draft ETS

Currently, the most frequently used technique for early validation is to organise reviews performed by experts. Subclause 6.1.1 focuses on formal validation, i.e. approaches to simulate or analyse the formal description of the system by means of computer-based tools. Formal validation is an emerging technology, that will gain importance in the near future.

6.1.1 Formal validation

Formal validation of a standard includes both checking the syntactic and semantic correctness of the specification, and checking that the known requirements on the specified system are expressed by the specification.

The main techniques implemented in tools for formal validation are:

a) interactive simulation;

b) random state space exploration;

c) "exhaustive" state space exploration.

All techniques are based on execution of the specification by computer. Therefore the techniques can only be applied if it is feasible to execute the SDL diagrams used in the specification, for example by generating executable code. In most cases the SDL diagrams as they are found in an ETS may not be complete and/or detailed enough to be processed by the tools directly. In that case additional information needs to be provided. This information can comprise, for example:

- choices between implementation options;

- specification of parts of the system that are out of the scope of the standard but still needed in order to have a complete system specification for execution;

- programming language code for SDL concepts that are too complicated to automatically generate code from (typically, abstract data types and operations).

These activities will be referred to as the building of a validation model. One or more validation models can be built and be processed by validation tools.

With a simulation tool, the specifier can interactively simulate the execution of the specified system and investigate the specified behaviour, variable values, etc. In this way the specifier can verify that the intended behaviour is indeed part of the specified behaviour. The major drawback with this technique is that it focuses on the expected behaviour of the specified system, since the person performing the simulation has a clear understanding of how the system is supposed to function. The behaviour of the system related to unintentional usage will not be revealed during simulation.

The second type of automated validation techniques is based on state space exploration or reachability analysis. A state space exploration algorithm generates and analyses, by approximation, all system states that can be reached from an initial system state. In other words, it will automatically construct all possible execution sequences of the system starting from the initial system state. The states, or sequences of states, can then be analysed with respect to certain correctness criteria. The correctness criteria can be, for example, absence of deadlock, no violations of dynamic semantics (e.g. division by zero, or sending of signals to non-existent processes), or temporal claims about the behaviour of the system. An example of a temporal claim is "signal A cannot be sent before signal B is received". In general, algorithms for state space exploration are based on operational definitions of the semantics of the specification language (see ETR 060 for an operational semantics of SDL).

However, it is often not possible to generate all possible system states of a realistic system due to the state space explosion, i.e. the number of states is so large that it exceeds the capabilities of the computer system used for the analysis. Several techniques have been developed to deal with the state space explosion problem. For a detailed description of techniques that are applicable to SDL specifications, see references 7), 12) and 14) of annex D.

Since all techniques based on state space exploration suffer from this state space explosion problem, it is important to decrease the size of the state space (complexity) of the validation model. This can be done in the following way:

a) restrict the number of instances of a process;

b) reduce the size of the state space of processes;

c) specify (parts of) the behaviour of the environment;

d) make assumptions on maximum number of signals in channels, signal routes, and input queues of process instances.

Unfortunately, these adaptations will increase the distance between the validation model(s) and the standard SDL diagrams. The validation model(s) may contain details that are not present in the standard (because they do not need to be standardised) or make assumptions on queue sizes that are in conflict with the semantics of SDL.

It is important to be aware of this distance between the standard and its validation model(s). If errors are discovered during the validation process, it should be investigated if the errors can be traced back to the standard. If so, the standard should be corrected and new validation model(s) can be built. If not, an error was made in the building of the validation model, which is not significant for the standard. Since this is an ongoing process, a specification team should continuously experiment with validation model(s) during the development of a standard.

6.1.2 Important aspects of a specification from a validation perspective

One of the major concerns of this ETS is how to restrict the features and styles of using SDL to enable adequate validation of a specification.

Since reviews by human experts is the most frequently used validation technique, it is important that a standard can be understood without unnecessary difficulties. The use of SDL in standards is helpful for understanding. SDL concepts have a clear and unambiguous interpretation. SDL diagrams allow to present the standard in a well structured way.

It can be expected that the application of computer-based tools for formal validation will play a more and more important role. In order to apply formal validation techniques it is essential that the SDL diagrams are correct with respect to the language definition. Furthermore, the use of SDL concepts that prohibit execution of the specification should be avoided. In order to minimise the distance between the standard specification and its validation model(s), the state space of the specification should not be unnecessarily big. This can be realised, for example, by limiting the number of processes, limiting size of data structures, or limiting the number of states per process.

6.2 Testing of telecommunication products

6.2.1 Conformance testing

Standardisation of telecommunication systems, services, protocols, and interfaces is aimed at enabling interoperability between products made by different manufacturers. Testing the conformance of a product to a standard, is considered to be essential to ensure that a product is able to interoperate with other products that implement the same standard. Conformance tests are usually carried out by an independent test laboratory.

A conformance test consists of two parts (ISO 9646-1 [9]):

a) the static conformance review, i.e. checking whether the choices between the implementation options that the manufacturer claims to have implemented is a combination allowed by the standard. The static conformance review consists of a check of the Implementation Conformance Statement (ICS), part of the documentation of the product, against the standard.

b) the dynamic conformance test, i.e. execution of test cases against the product, to check if the product really has implemented the standard, given the options that are claimed to be implemented in the ICS.

A conformance test suite is aimed at checking whether a product conforms to the standard. The purpose of an individual test case is related to one conformance requirement of the standard. A conformance test case is a black box test: it only controls and observes the product via the PCOs. Usually abstract test cases are described in the standard test notation TTCN (ISO 9646-3 [10]).

6.2.2 Important aspects of a specification from a testing perspective

In the process of test suite development for an ETS, problems to be dealt with are:

- identification of the conformance requirements in the ETS;

- identification of the interfaces of the product type that can be accessed for test execution;

- determine how the requirements can be tested using the available interfaces, in a telecommunication product that claims to implement the standard.

The suitability of an ETS to perform these steps determines the testability of the ETS. One of the major concerns of this ETS is to restrict the features and styles of SDL in order to enable adequate testing of a product that implements the specification.

A specification can be defined to be testable [1)] if it unambiguously defines:

a) the allowed implementation options;

b) the conformance requirements;

c) the normative interfaces;

and if

d) all conformance requirements can be checked, up to a certain degree of confidence, by running tests against a product via normative interfaces.

[1)] Testability is currently only defined on the level of specification, not on the level of individual conformance requirements. This may be improved in future versions of this ETS.

This definition is based on experience in development of test suites for telecommunication systems. A major problem in designing a test suite is to identify the optional parts in a specification. In most cases it is necessary to first produce a ICS proforma that clearly indicates which parts of a specification are mandatory and which parts are optional. Another problem is the selection of an abstract test method, i.e. to decide which interfaces of a type of product will be accessible for testing.

Furthermore, the definition anticipates expected problems in test derivation from an SDL specification. Many current automated test derivation methods have in common that they produce enormous amounts of tests, too many to run in practice. Often, an individual test case is not clearly related to a conformance requirement, which makes it difficult to select a subset of the test suite. In a natural language specification, it is quite easy to find conformance requirements. The test designer goes through the text with a marker, and highlights every sentence containing a "shall". If SDL is used to define a rather complete model of the telecommunications system, this is more complicated.

An example of a class of requirements that usually is well addressed in a conformance test, but is often neglected in specifications using SDL, is the handling of inopportune or invalid events.

In the following, the main issues concerning testability are discussed in more detail. These issues are:

- normative interfaces;
- implementation options;
- conformance requirements;
- handling of invalid inputs.

6.2.2.1 Normative interfaces

A normative interface in a telecommunication product is defined as the physical or software interface of a product on which requirements are imposed by a telecommunication standard. Definition of normative interfaces allows telecommunication products built by different manufacturers to be interconnected, to jointly provide a telecommunication service. These requirements typically concern the information flow between the systems, encoding of information, and the way products can be connected. If the products are physical devices, the latter may comprise the dimension of plugs or frequencies used to transmit radio signals. In case of software products, the connection may be defined by language bindings for programming interfaces.

An example of a normative interface in ISDN is the user-network interface (the S reference point).

A telecommunication standard, or set of related standards that together define a product type, shall clearly indicate which interfaces in the product are considered to be normative. The definition of normative interfaces should be kept to the minimum that is necessary in order to guarantee interoperability. Restriction of implementation freedom by defining more normative interfaces than necessary should be avoided.

Normative interfaces shall be marked in an SDL diagram with a comment "normative" attached to the channels that model the interfaces. Informative channels may optionally be marked with a comment "informative". Figure 2 shows an example.

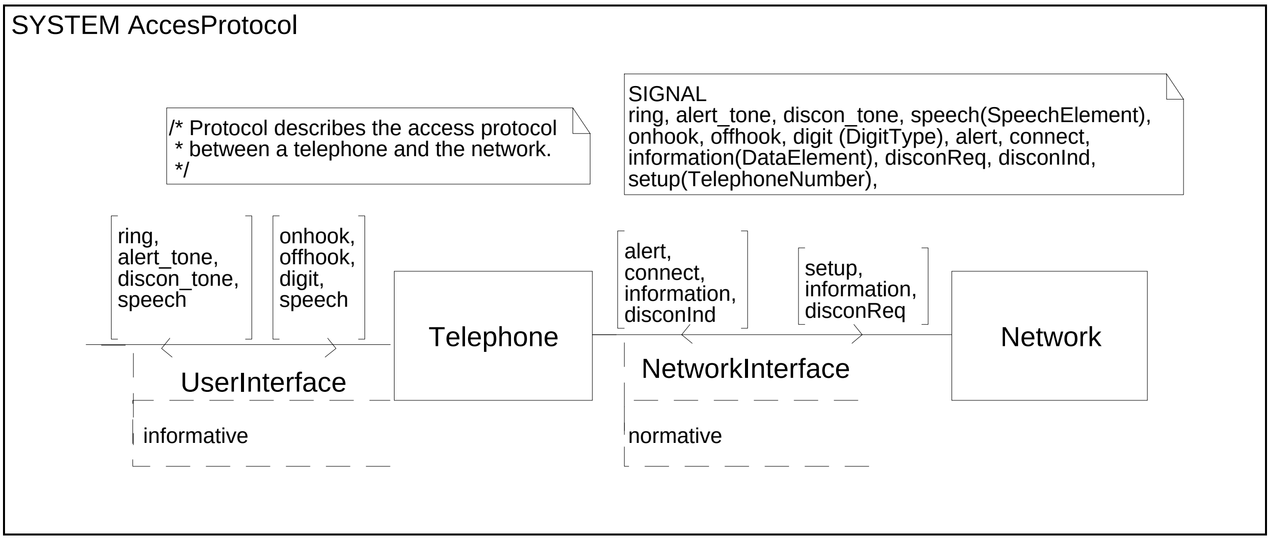

SYSTEM AccesProtocol
/* Protocol describes the access protocol
* between a telephone and the network.
*/
SIGNAL
ring, alert_tone, discon_tone, speech(SpeechElement),
onhook, offhook, digit (DigitType), alert, connect,
information(DataElement), disconReq, disconInd,
setup(TelephoneNumber),
ring, alert_tone, discon_tone, speech
onhook, offhook, digit, speech
Telephone
alert, connect, information, disconInd
setup, information, disconReq
Network
UserInterface
NetworkInterface
informative
normative

Figure 2: A system diagram indicating the normative interfaces of a system

However, in many ETSs it may not be possible to indicate the normative interface, because the ETS describes only part of a product type. A specification of a supplementary service, for example, uses a parameter of standard call control messages to exchange information. Hence, the normative interface can not be indicated in a supplementary service standard. It should, however, be possible to find the definition of the normative interface in one of the normative references that are listed in the standard.

A normative interface should not be confused with a Point of Control and Observation (PCO) (see ISO 9646-1 [9]) a concept in conformance testing. Normative interfaces can be used as PCOs, but a PCO is not necessarily a normative interface. For example, the remote PCO in the distributed test method (ISO 9646-1 [9]) is located at an internal interface of the test execution system and not a part of the product under test.

6.2.2.2 Implementation options

Most ETSs do not describe a single system, but rather a type of system, for example mobile telephones that function within the Digital European Cordless Telecommunications (DECT) infrastructure. Often these standards contain implementation options, allowing products conforming to the standard to have different features. In the example of DECT telephones a common basic function is the possibility to initiate telephone calls. An optional feature is the capability to receive incoming calls.

For a tester it is important to know which parts of the standard are claimed to be implemented in a product. Therefore, the manufacturer of a product is requested to provide an ICS.

As a consequence, it is important that an ETS clearly documents the implementation options in the standard. Figure 3 shows how SDL can be applied for this purpose.

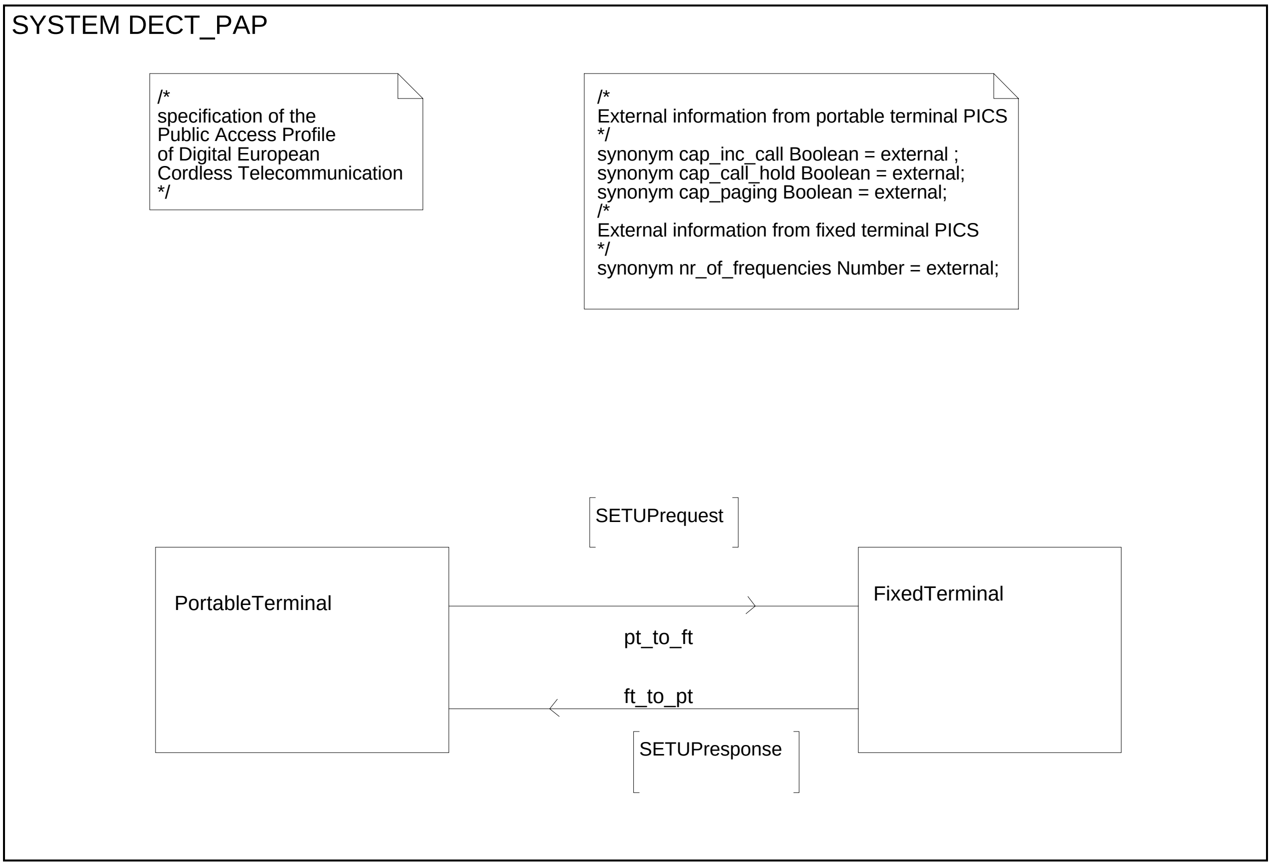

SYSTEM DECT_PAP
/*
specification of the
Public Access Profile
of Digital European
Cordless Telecommunication
*/
/*
External information from portable terminal PICS
*/
synonym cap_inc_call Boolean = external ;
synonym cap_call_hold Boolean = external;
synonym cap_paging Boolean = external;
/*
External information from fixed terminal PICS
*/
synonym nr_of_frequencies Number = external;
SETUPrequest
PortableTerminal
FixedTerminal
pt_to_ft
ft_to_pt
SETUPresponse

Figure 3: System diagram with declaration of external synonyms

The system diagram uses external synonyms to model basic implementation options. The external synonym cap_inc_call has an unknown value, which has to be provided externally. However, the synonym can be used within the specification. In annex C a complete example is given that shows how implementation options are related to the behaviour of the system.

6.2.2.3 Conformance requirements

The normative parts of SDL diagrams express conformance requirements. This subclause gives an overview of which requirements are expressed by the different diagrams.

In a system or block diagram, channels can be marked as normative. The requirements imposed by this are, that the signals that are listed at the normative channels, including the parameters they carry, can be transmitted via a normative interface of a product.

Data type definitions (in ASN.1 or SDL data types) of signals or signal parameters that are to be exchanged via normative interfaces, impose requirements on the structure and information contents of signals.

Process and procedure diagrams contain requirements concerning the precise temporal ordering of signals, values of their parameters, and timing.

The conformance requirement expressed by an MSC is that the sequence of events given in the MSC can be performed by the system. Such a requirement is already expressed in the related SDL diagrams. The use of MSCs contributes to the testability of a standard, because they may be used to guide the selection of test purposes.

In order to distinguish between parts of SDL diagrams that are normative and parts that are given just for information, the normative parts should be marked as such. A normative part of a specification (normative channel, normative data type definition) should not be confused with a normative interface in a product. Normative parts of a specification put requirements on message exchange via normative interfaces, but do not need to define the normative interface completely.

For example, suppose that in the Open systems Interconnection (OSI) transport protocol specification the channel that conveys Transport Protocol Data Units (TPDUs) is marked as normative. This implies that in an implementation of the transport protocol, there is an interface where these T-PDUs can be observed: the T-PDUs will be contained in the user data field of lower-layer PDUs (or may even be segmented over multiple lower layer PDUs). In the lowest layer the T-PDUs are embedded in bit streams. The OSI transport specification does not contain the precise definition of what happens on a cable that connects two systems (the OSI normative interface). To determine this, the standards of all other layers need to be consulted (there are even multiple possibilities, depending on whether a Local Area Network (LAN), Connectionless-mode Network Service (CLNS) or Connection Oriented-mode Network Service (CONS) is used to provide the lower layer services). The OSI transport specification does, however, impose strict requirements on the encoding of those parts of the bit stream that represent the T-PDUs.

It is hard to derive conformance requirements from a standard that specifies a system at more than one level of abstraction. Suppose, for example, that one part of the standard specifies that a complete file is sent from A to B, and in another part it is stated that A should send a big file in blocks of a specific size, of which the receipt should be acknowledged. This makes it complicated to decide which conformance requirements apply. Therefore the normative parts of a standard shall specify the system at one level of abstraction only.

Levels of abstraction should not be confused with structure. It is possible to specify a system at one level of abstraction, and still use structure, i.e. describe the parts of a system and how the system is composed of these parts.

6.2.2.4 Handling of invalid inputs

A conformance test of a system usually includes tests for response of the system under test to invalid inputs. Therefore, it is important that a specification defines how the system should respond to invalid inputs.

A concept of "invalid" inputs does not exist in SDL. As soon as a signal is declared it is considered to be valid. The formal semantics of ITU-T Recommendation Z.100 [1], annex F, assumes that the environment of the system sends only valid signals to the system.

However, to specify the response of the system to invalid inputs, a signal named "invalid_input" can be introduced. This signal is assumed to informally represent all signals that are in reality considered to be invalid. Formally, however, it is just another SDL signal, for which it is possible to define how it should be handled. Figure 4 shows three ways in which a process may handle the "invalid_input" signal:

a) ignore invalid input;

b) give an error message;

c) future behaviour undefined.

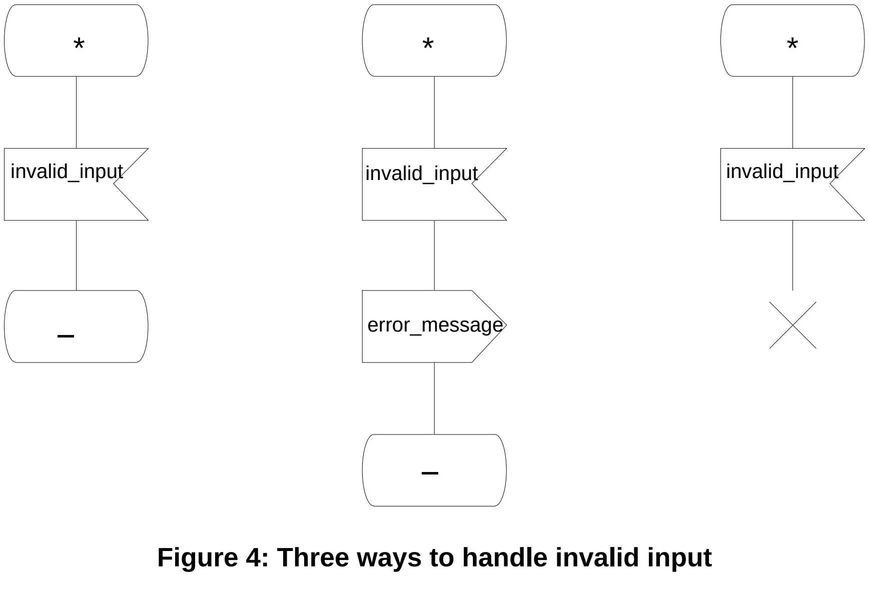


Figure 4: Three ways to handle invalid input

In practice it is often necessary to distinguish between different types of invalid inputs, for example "unexpected signal", "contents error", and "missing signal parameter". In that case it is possible to introduce a signal for each of these types, and specify how they should be processed.

6.3 Specification principles to enable validation and testing

Based on the previous considerations concerning conformance testing and validation, some principles can be formulated. These principles form the basis for the selection of the CCITT SDL concepts for use in ETSs, and rules for the use of selected concepts.

consistency An ETS shall be internally consistent, i.e. there shall be no contradictions between text paragraphs, or between a diagram and the text, or between different diagrams.

clarity An ETS shall clearly and unambiguously define requirements on the telecommunication product it specifies. The specification shall be well-structured, in order to ease reviews by human experts.

correct use of formalisms The diagrams presented in an ETS shall be syntactically and semantically correct with respect to the formal language definitions. The use of concepts that are not supported by tools should be avoided. This principle is essential to enable the use of computer-based tools for consistency checking, validation, and test derivation.

avoid state space explosion An important class of formal validation methods is based on state space exploration. By specification of infinite numbers of process instances, or infinite data types, the state space of the system explodes. This prohibits the use of validation tools that are currently available.

avoid implicit non-determinism If non-determinism is not explicitly specified, it is not easy to discover, and may easily be overlooked by the designer of tests. Furthermore, it is difficult to determine whether the non-determinism is intentional, or is a mistake in the standard.

indicate implementation options A standard should clearly describe the implementation options. These options will be reflected in the ICS, that a manufacturer provides to indicate which options are supported in a product. This is important for selection of relevant tests for a product.

indicate normative parts of standard A standard should clearly indicate which parts are normative. This determines which conformance requirements are implied by the standard.

one level of abstraction A standard should not specify a system on multiple levels of abstraction. This is essential in order avoid misinterpretation about which conformance requirements are implied by the standard.

7 SDL in European Telecommunication Standards

7.1 Introduction

The SDL diagrams are to be presented in the main body of an ETS, together with normal textual information, tables and informal figures. SDL diagrams, or parts of them, shall be marked as normative or informative. The SDL diagrams should also contain comments to ease understanding and indicate relations with text clauses.

Duplication of information should be avoided. Requirements that can be read in an SDL diagram, should not be repeated in the text, except when the text brings together requirements that are distributed over several SDL diagrams.

For example, textual information such as:

> "on receipt of a SUSPEND message, the network shall respond by sending a SUSPEND REJECT message with cause # 84 "call identity in use" if the information contained in the SUSPEND message is not sufficient to avoid ambiguities in subsequent call re-establishment".

is to be considered as duplication of the information found in the SDL process diagram in figure 5.

Whenever a conflict exists between the natural language description and a SDL or MSC diagram, or between SDL and MSC diagrams, this shall be considered as an error in the specification. If such an error is detected, the conflict should be resolved by changing either the text, the SDL diagram, or the MSC diagram. There shall be no order of precedence that automatically resolves conflicts by giving priority to either the textual or formal descriptions.

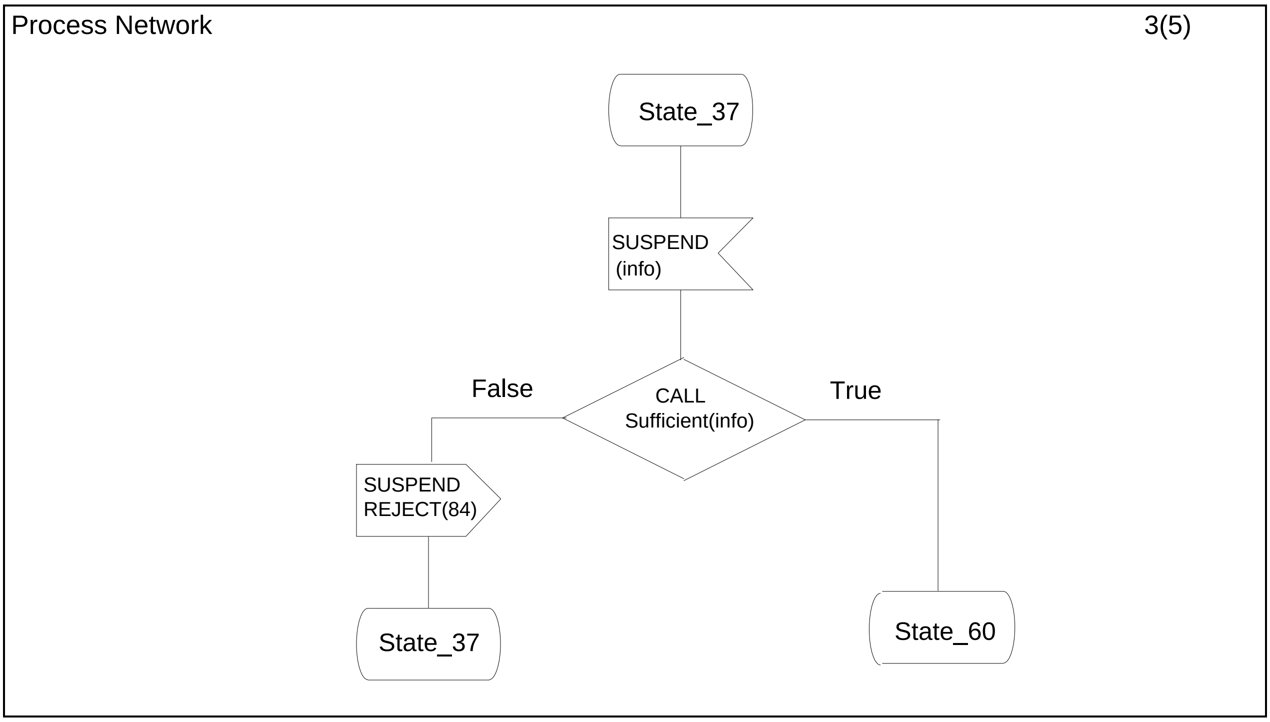


Figure 5: Part of network call control process

7.2 Structure and contents of an ETS containing SDL

It is recommended that an ETS containing SDL diagrams has the following structure:

Title page According to PNE-Rules [8], subclauses 2.2.1, 2.3.1

Table of Contents According to PNE-Rules [8], subclause 2.2.2, with the exception that its presence is mandatory

Foreword According to PNE-Rules [8], subclause 2.2.3

Introduction Describes the environment of the standardised telecommunication system. If the standard only describes part of a telecommunication product then this clause also describes other relevant parts of the telecommunications product.

1 Scope According to PNE-Rules [8], subclause 2.3.2

2 Normative references Where applicable references to:

- ITU-T Recommendation Z.100 (1993): "Specification and Description Language (SDL)" [1].

- ITU-T Recommendation Z.105: "SDL combined with ASN.1 (SDL/ASN.1)".[2].

- ITU-T Recommendation Z.120 (1993): "Message Sequence Charts (MSC)".[3].

- ITU-T Recommendations X.680-X.683: "Abstract Syntax Notation One (ASN.1)". [7].

- ETS 300 414: "Use of SDL in European Telecommunication Standards; Rules for testability and facilitating validation ".

3 Definitions According to PNE-Rules [8], subclause 2.4.1, if any definitions are given.

4 Symbols and abbreviations According to PNE-Rules [8], subclause 2.4.2, if any symbols or abbreviations are given.

5 Conformance to this standard Describes which parts of the standard contain requirements on implementations and which parts are included only for information.

6 System overview A description of the system containing:

a) an overview in natural language of the functions provided by the system;

b) an identification of the environment of the system;

c) a decomposition of the system into functional entities;

d) a description of the normative interfaces of the system, if applicable;

Items c) and d) shall be described using a SDL system diagram including signal declarations and data type definitions for signal parameters associated with the normative parts of the ETS.

e) a list of global functional requirements, i.e. requirements only implicitly given by the SDL specification as a whole;

Item e) should be described in precise, natural language.

f) typical examples of the communication between the system and its environment in normal situations;

g) typical examples of the communication between the system and its environment in exceptional situations.

items f) and g) shall be described using MSCs (at least one MSC for each global function).

Functional specification Describes the behaviour of the system by using:

a) SDL block diagrams giving the structure of the specification up to the process level;

b) SDL process and procedure diagrams describing the behaviour for each process in the system;

c) data type definitions.

Performance and reliability requirements Since these requirements can not be expressed in the SDL formalism, they have to be stated in normal text.

Encoding of messages that cross normative interfaces Encoding shall be defined unambiguously. For example with ASN.1 definitions and a reference to standardised encoding rules, or definition of original encoding rules. Alternatively, bit tables can be used, that show the exact lay-out in bits of the messages (see example in figure 6) and their meaning.

The normative requirements are contained in the sections on system overview, functional specification and performance and reliability requirements. These sections can also, in addition to the SDL and MSC diagrams, contain state overview diagrams, block tree diagrams or other kinds of diagrams. Such diagrams are considered as illustrations only, and shall not contain any normative requirements.

		Bits			
5	**4**	**3**	**2**	**1**	
0	0	0	0	0	speech
0	1	0	0	0	unrestricted digital information
0	1	0	0	1	restricted digital information
1	0	0	0	0	3,1 kHz audio
1	0	0	0	1	7 kHz audio
1	1	0	0	0	video

All other values are reserved

Figure 6: Bit table defining the encoding of ISDN information transfer capability

A drawback of using bit tables is that they are not related to the SDL diagrams in the specification. Furthermore, they can be complicated to use and understand for structured messages.

8 Specification and description language concepts

8.1 Introduction

This clause describes the subset of SDL 1992 that is allowed for use in ETSs. The description is given with respect to two categories of SDL concepts: diagrams and symbols. Not all symbols of ITU-T SDL are included: those that are harmful for testability or validation are left out. In annex B, motivations for the exclusion of concepts can be found.

A short explanation is given for every included diagram or symbol. There are references to the relevant sections in ITU-T Recommendation Z.100 [1], where the exact meaning of each diagram or symbol can be found. Occasionally examples of how the concept can be used in ETSs are given. For some concepts, rules are given when unrestricted use of that concept decreases the testability and validability of the specification. In such cases, guidelines are given about how to overcome the restrictions introduced by the rules.

Rule 1: In the printed version of an ETS only the graphical representation of SDL shall be used.

NOTE: The graphical representation includes the common textual grammar between SDL/PR (textual, Phrase Representation) but not the textual grammar only used by SDL/PR.

This rule is motivated by reasons of clarity. The graphical representation of SDL is more easy to understand for human experts than the textual representation.

Table 1: Selection of SDL concepts

	Unrestricted use allowed	Restricted use allowed	Not allowed
System or block diagrams	block, channel, comment, package, process, process create line, select, signal list, signal route, text, text extension	block substructure, data type definition, macro call, signal declaration	channel partitioning, signal refinement
Process or Procedure diagram	comment, input, join, label, optional transition, priority input, process creation, process start, process stop, procedure call, procedure return, procedure reference, procedure start, save, state, synonym, text, text extension, variable	continuous signal, decision, macro call, output, task, timer	enabling condition, import and export, internal input and output, service, view and reveal
Data type diagram	predefined data	abstract data types, ASN.1 type definition	name class literal

Table 1 gives an overview of the selection of SDL concepts. There are no provisions in this standard either allowing or prohibiting use of concepts of Z.100 [1] not explicitly listed in table 1. This is because very little experience in using them exists:

- the concepts in the column "Full use allowed" are the concepts that can be used without any restrictions, i.e. they are not considered harmful for testability or validation, or they were considered indispensable by a panel of experienced SDL users;

- the concepts in the column "Restricted use allowed" are concepts that, when used in a specific way, influences testability or validation. For each such concept a rule is given restricting the allowed use;

- the concepts in the column "Not allowed" are concepts that are harmful for testability or validation, or were considered superfluous.

Rule 2: The following SDL symbols shall not be used in ETSs: channel partitioning, signal refinement, enabling condition, internal input and output, view and reveal, import and export, service, and name class literals.

The motivation for this rule can be found in annex B.

NOTE: If a concept is not selected for use in ETSs, this does not necessarily mean that the concept is useless in general. For example, having multiple levels of abstraction is undesirable in standardisation, but may be extremely useful in product development.

The use of abstract data types with axioms and the use of macros is discouraged.

8.2 SDL diagrams

This subclause describes the different kinds of diagrams of an SDL specification. A diagram can consist of multiple pages. The top right hand corner should show the page number of the diagram, and between parentheses, the total number of pages of the diagram. For example if the top right hand corner of a page shows "2(5)", then that page is the second page of a diagram with a total of five pages.

8.2.1 System diagram

A system diagram (ITU-T Recommendation Z.100 [1], §§ 2.4.2) gives an overview of the specified system. It describes the static structure of the system. It is further refined using block diagrams (subclause 8.2.2). The normative interfaces of the system (see subclause 6.2.2.1) shall be indicated in the specification if applicable. This may impose restrictions on the way how the system is structured.

The symbols which can be used in a system diagram are shown in table 2.

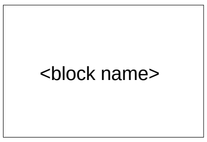
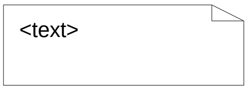
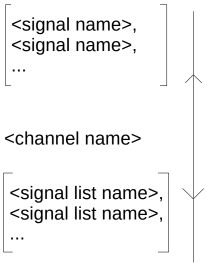
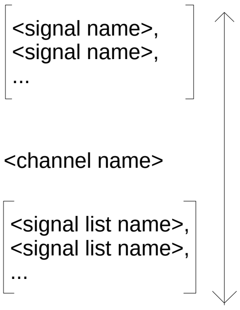
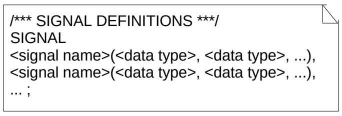
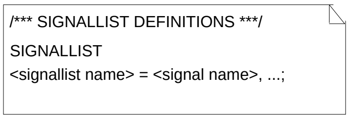
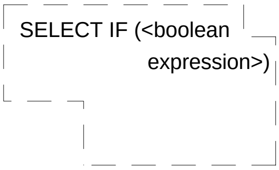
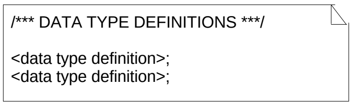
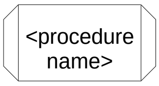
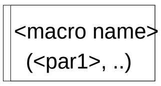
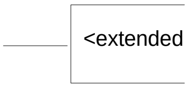
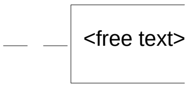
Rule 3: A system diagram shall be used to describe how the system is composed of functional units (modelled with blocks).

The motivation for this rule is given by the principle of clarity. The system diagram provides a uniform way to indicate implementation options and normative parts.

Rule 4: In a system diagram, the blocks, and channels shall appear on page 1. Signal definitions shall come before the data definitions.

This rule is motivated by the principle of clarity.

Table 2: Symbols allowed in a system diagram

	**Block symbol**		**Text symbol**
	**Channel with delay symbol**		**Channel without delay symbol**
 /*** SIGNAL DEFINITIONS ***/ SIGNAL <signal name>(<data type>, <data type>, ...), <signal name>(<data type>, <data type>, ...), ... ;	**Signal definitions in text symbol**	 /*** SIGNALLIST DEFINITIONS ***/ SIGNALLIST <signallist name> = <signal name>, ...;	**Signallist definitions in text symbol**
	**Select symbol**	 /*** DATA TYPE DEFINITIONS ***/ <data type definition>; <data type definition>;	**Data type definitions in text symbol**
	**Procedure reference symbol**		**Macro call symbol**
	**Text extension symbol**		**Comment symbol**

8.2.2 Block diagram

A block diagram (ITU-T Recommendation Z.100 [1], §§ 2.4.2) is used to describe how a block is composed of functional units and how these units are interconnected. The composition of a block can either be described by processes and how the processes are interconnected, or by blocks and how the blocks are interconnected.

If the composition of a block is described by blocks, the same rules as for a system diagram shall apply, and the same symbols as in a system diagram can be used, only their scope is restricted to that block.

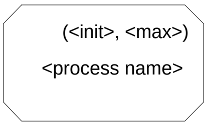
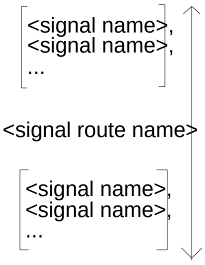
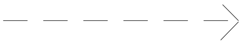
The following symbols can be used in both a block diagram composed of processes and in a system diagram:

- signal definition;
- signallist definition;

- text symbol;
- text extension;
- comment;
- data type definition;
- macro call;
- procedure symbol;
- select symbol.

The description of these symbols can be found in subclause 8.3. The symbols that can be used in addition in a block diagram composed of processes are shown in table 3.

Table 3: Additional symbols that can be used in a block diagram

 **Process symbol**	 **Signal route symbol**
 **Process create line symbol**	

By using block diagrams that are composed of blocks, the SDL specification can be structured in a hierarchical way of any complexity: the system is made up of blocks, blocks can consist of blocks, etc. Blocks on the lowest level of the hierarchy are composed of processes. Figure 7 gives an example of a block TransportLayer, which is composed of blocks. The diagram of one of these blocks, NetworkLayer is shown in figure 8. This block is composed of one process and, therefore, is not further structured.

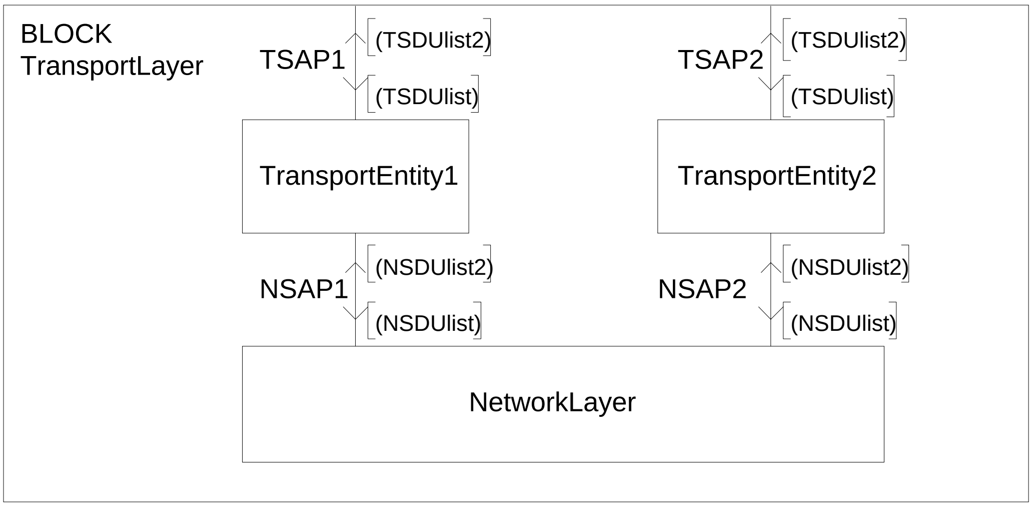


Figure 7: A block composed of blocks

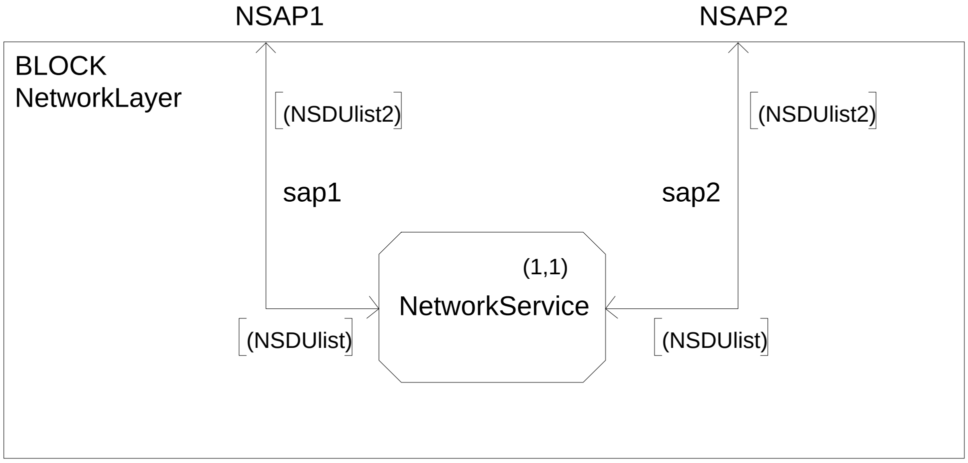


Figure 8: A block composed of processes

Rule 5: Block diagrams shall be used to describe how the functional units are composed of processes or blocks.

This rule is motivated by reasons of clarity. A well structured specification simplifies review by human experts.

Rule 6: A block diagram in the normative part of a specification shall not have alternative sub-block definitions, i.e. the feature of SDL (ITU-T Recommendation Z.100 [1], §§ 3.2.2) to describe the decomposition of a block in blocks as well as in processes is not allowed.

This rule is motivated by the principle of having only one level of abstraction.

8.2.3 Process diagram

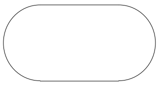
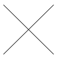
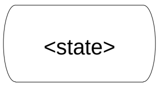
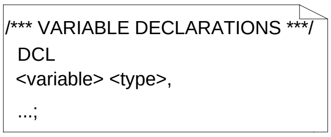
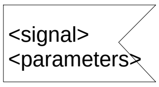
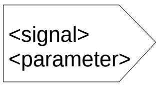
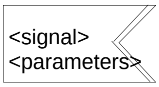
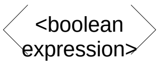
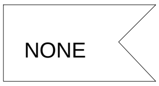
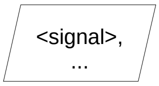
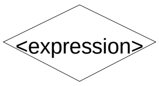
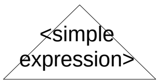
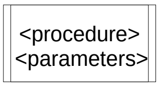
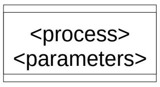
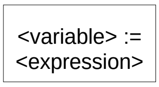
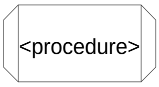
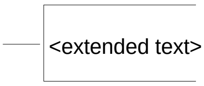
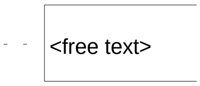
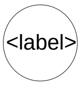
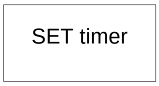
A process diagram (ITU-T Recommendation Z.100 [1], §§ 2.6) is used to describe the dynamic behaviour of a process. A process is described as a graph with states and transitions between states.

The symbols that can be used in a process diagram are listed in table 4.

Symbols that can be used in process diagrams, block diagrams, and system diagrams are:

- data type definitions;
- text symbol;
- text extension symbol;
- comment;
- macro call.

Table 4: Allowed symbols in process diagrams

	**Process start symbol**		**Process stop symbol**
	**State symbol**	 /*** VARIABLE DECLARATIONS ***/ DCL <variable> <type>, ...;	**Variable declarations in text symbol**
	**Input symbol**		**Output symbol**
	**Priority input symbol**		**Continuous signal symbol**
	**Spontaneous transition symbol**		**Save symbol**
	**Decision symbol**		**Option symbol**
	**Procedure call symbol**		**Process creation symbol**
	**Task symbol**		**Procedure reference symbol**
	**Text extension symbol**		**Comment symbol**
	**Join symbol**		**Set timer symbol**

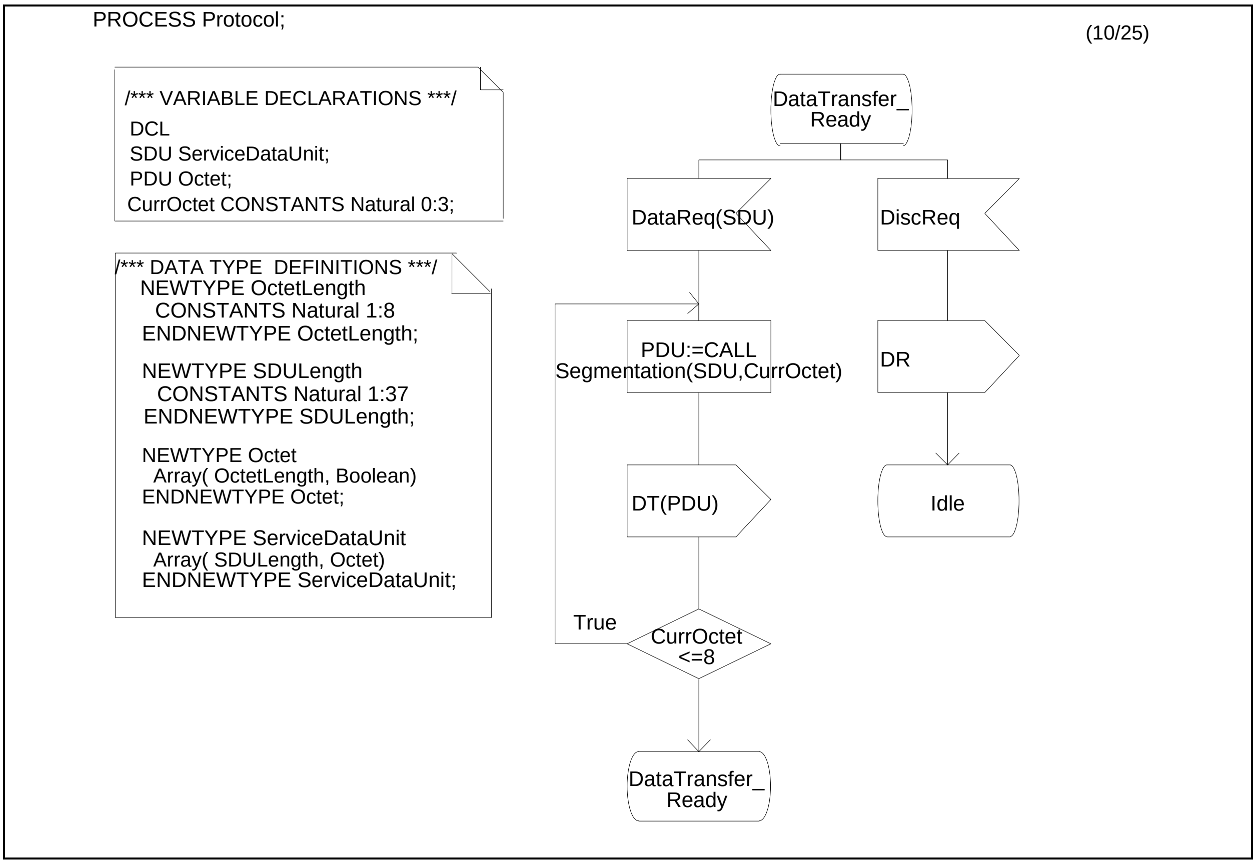
A process can be parameterized. The parameters are passed during the creation of the process by the creating process. The parameters of a process are declared in the process heading. A declaration of a parameter is similar to a normal variable declaration (subclause 8.5.2), with the exception that it is preceded by the keyword **fpar**.

If the process is further composed of procedures, these have to be referenced using a procedure symbol. A procedure symbol is used to indicate the procedures that are declared in the process diagram.

There are a number of transitions associated with a state. For each transition there is a trigger describing the condition for the transition to fire, for example the reception of a signal. During a transition, several actions can be executed. The transition ends in either a process stop or a nextstate symbol. An action can be any of the following:

- set/reset;
- task;
- decision;
- process creation;
- procedure call;
- output;
- macro call.

In figure 9 an example is shown of a page of a process diagram. The page shows how service data unit is split into a number of protocol data units.



```
/*** VARIABLE DECLARATIONS ***/
 DCL
 SDU ServiceDataUnit;
 PDU Octet;
CurrOctet CONSTANTS Natural 0:3;
```

```
/*** DATA TYPE  DEFINITIONS ***/
   NEWTYPE OctetLength
      CONSTANTS Natural 1:8
    ENDNEWTYPE OctetLength;

    NEWTYPE SDULength
      CONSTANTS Natural 1:37
    ENDNEWTYPE SDULength;

    NEWTYPE Octet
      Array( OctetLength, Boolean)
    ENDNEWTYPE Octet;

    NEWTYPE ServiceDataUnit
      Array( SDULength, Octet)
    ENDNEWTYPE ServiceDataUnit;
```

Figure 9: Example of a process diagram

8.2.4 Procedure diagram

A procedure is used to structure processes. It is described as a graph with states and transitions between states (ITU-T Recommendation Z.100 [1], §§ 2.4.6), containing the same symbols as a process graph except that the symbols for the start and end of the procedure are different. A procedure can return a result by attaching a result expression to the procedure end symbol.

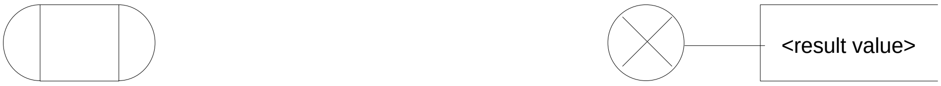


Figure 10: Start (left) and end (right) symbols used in procedure descriptions

The parameters of a procedure are listed in the upper left corner of the procedure diagram. There are two kinds of parameters:

In parameters These parameters are used to pass a value to the procedure. Changes to such a parameter in the procedure body have no effect on the calling procedure or process. These parameters can be preceded by the keyword **in**, although this is not necessary.

In/out parameters If such a parameter is changed in the procedure body, then the corresponding actual parameter in the calling process/procedure is also changed. These parameters are preceded by the **in/out** keyword.

In figure 11 an example is shown of a procedure that realises synchonization, by means of a handshake mechanism.

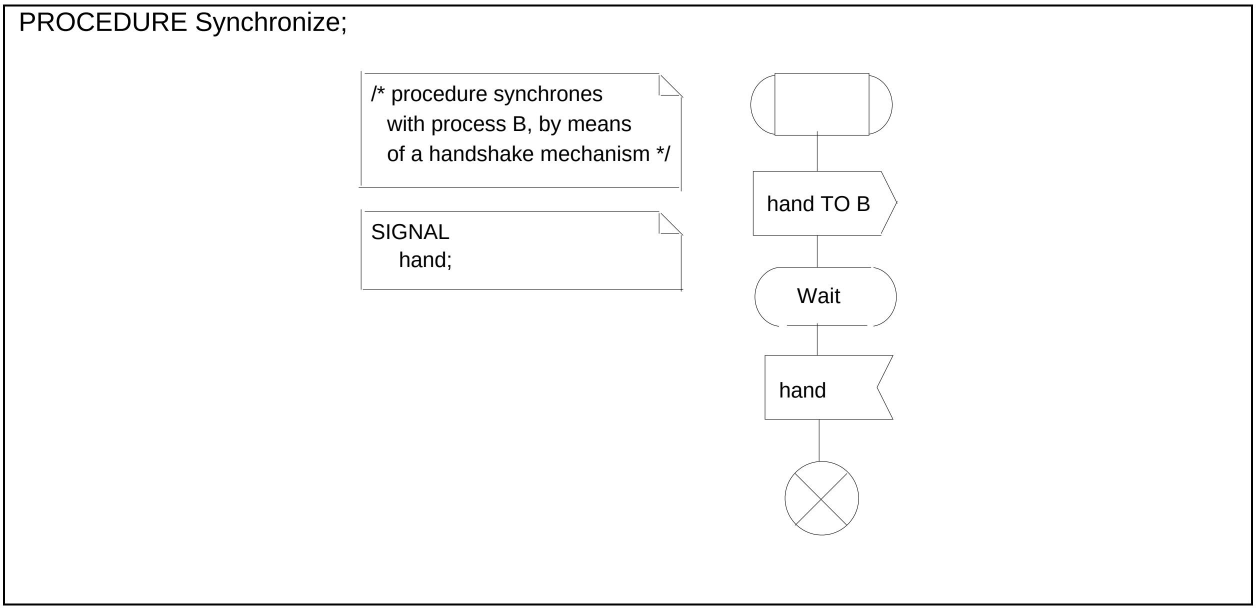


Figure 11: An example of a procedure defining a handshake mechanism

In figure 12 an example is shown of a value returning procedure named Segmentation. The procedure can be used to split a service data unit in a series of octets.

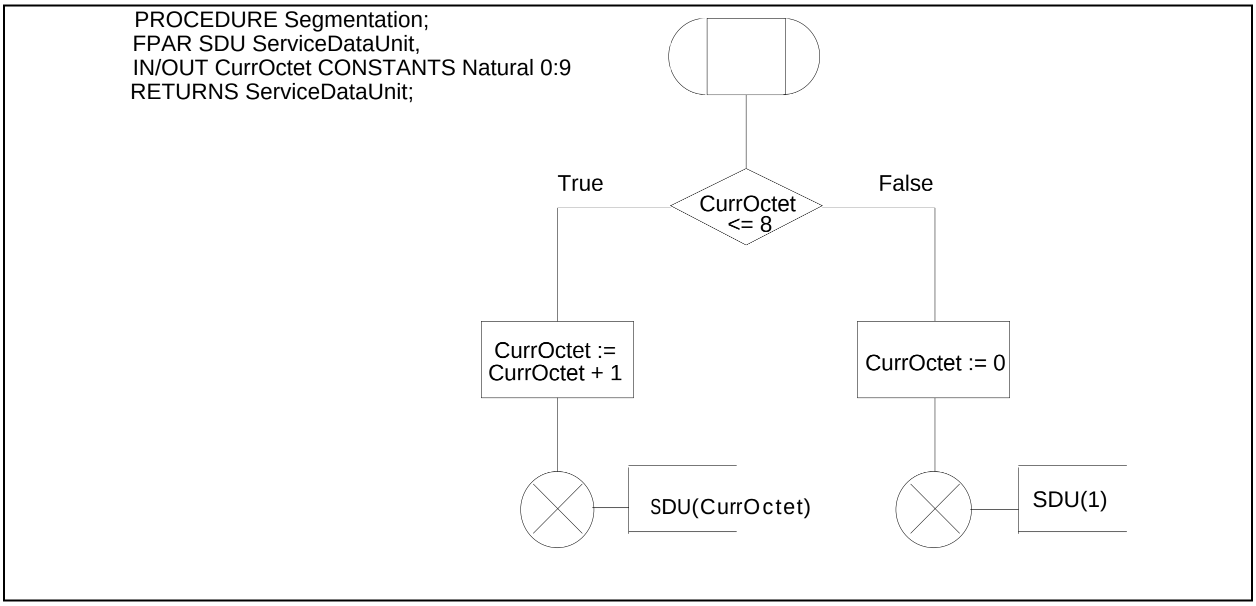


Figure 12: A value returning procedure defining segmentation

8.2.5 Macro diagram

A macro diagram (ITU-T Recommendation Z.100 [1], §§ 4.2) is used to describe a part of a system, block, process, or procedure diagram. Macros can be parameterized. A macro call in one of these diagrams expands to the full macro diagram with actual parameters. The connection points of the macro are named. When a macro is expanded in a diagram, these names are used to establish the right connections.

The use of macros is discouraged. The motivation is that macros are very powerful, and therefore can be used instead of many other SDL concepts, for example procedure or block substructure. Use of macros decreases the clarity of a specification. Furthermore the checking of macros is limited, which makes it impossible to use tools to check correctness (principle of correct use of formalisms).

8.3 Symbols used in system diagrams

8.3.1 Block

A block symbol (ITU-T Recommendation Z.100 [1], §§ 2.4.6) refers to a block diagram that defines a functional unit. The block symbol contains the name of the block.

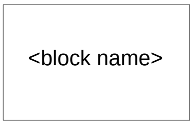


Figure 13: The block symbol

8.3.2 Channel

A **channel** (ITU-T Recommendation Z.100 [1], §§ 2.5.1) is used to describe an information flow between blocks. A channel can convey signals either in one direction or bidirectionally, which is indicated by arrows. For each direction it is described which signals can be conveyed by the channel. There exist two types of channels in SDL:

Channel without delay the time to transmit a signal is zero

Channel with delay the time to transmit a signal is unknown

The symbols for both channels are distinguished by the placement of the arrows: in channels without delay, the arrows are placed at the endpoint. In channels with delay they are placed anywhere on the channel line, except at the endpoint.

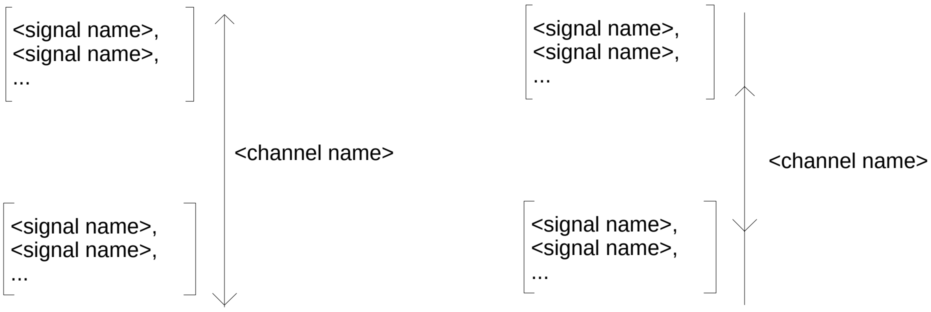


Figure 14: The channel symbols, channel without delay (left) and with delay (right)

Use of channels with delay may cause implicit non-determinism, which is harmful for testability. This is for example the case where the behaviour of a system depends on the arrival order of some signals conveyed by channels with delay.

An example of such a case is given in figure 15. This is an example of bad use of SDL. The system contains two blocks Block_A and Block_B, connected by two channels Channel_B and Channel_C. In Block_A there is one process, named A, connected to Channel_A, Channel_B and Channel_C. In the

Block_B there is one process, named B, connected to Channel_B, Channel_C and Channel_D. The process graphs of processes A and B are given in figure 16.

If signal a is sent to the system via Channel_A, it is not possible to predict whether signal d or signal e will be sent first from the system via Channel_D. The reason is that it is impossible to know whether signal b or signal c arrives first at process B, due to the unknown and possibly different delays of the channels Channel_B and Channel_C.

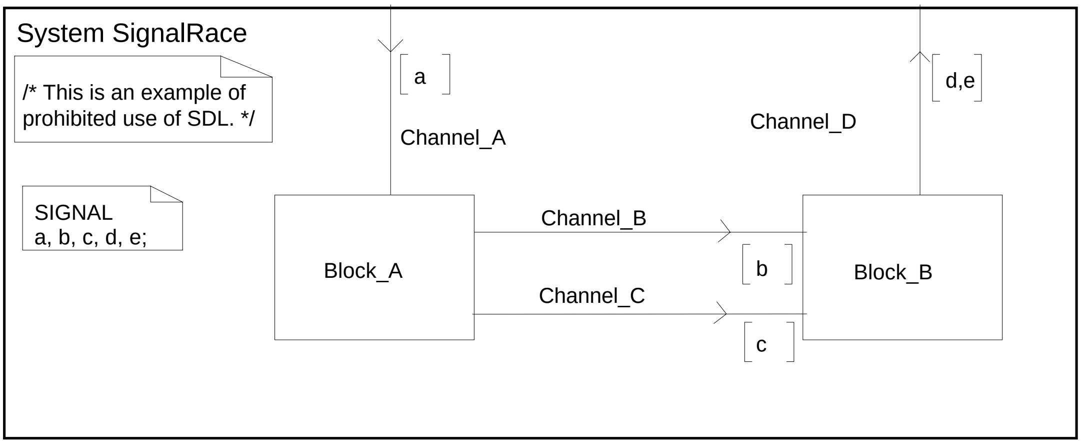


Figure 15: Example of a system where two channels with delay cause implicit non-determinism

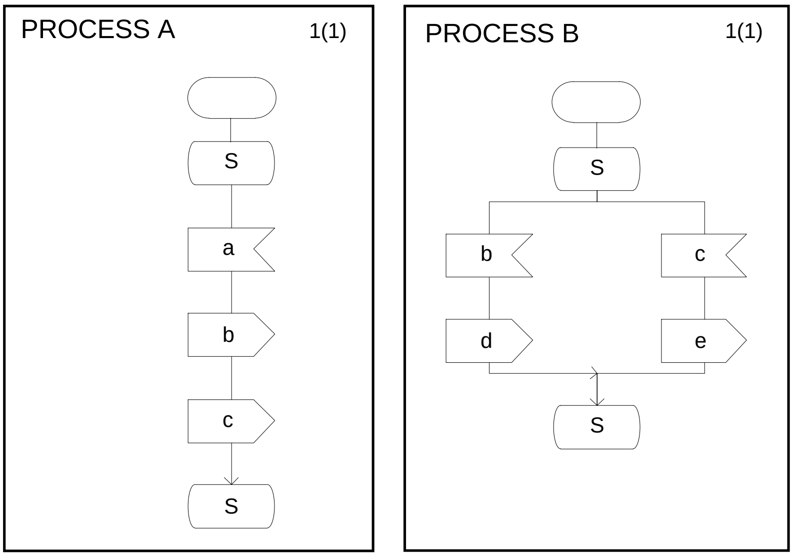


Figure 16: Process diagrams of process A and process B

Rule 7: Implicit non-determinism arising as an effect of using channels with delay shall be avoided.

This rule is motivated by the principles of clarity and avoiding implicit non-determinism.

Implicit non-determinism may also occur when two signals from different sources arrive almost simultaneously at the input queue of a process instance. This should be avoided as much as possible.

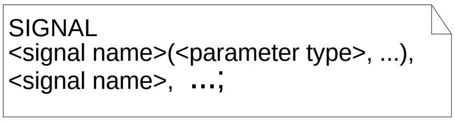
Rule 8: To every normative channel of the standardised system a comment "normative" shall be attached.

This rule is motivated by the principle that normative parts of an ETS should be clearly indicated.

Informative channels may be marked as such when this increases readability.

8.3.3 Signal definition

A signal definition (CCITT Recommendation Z100 [1], §§ 2.5.4) is used to define the parameters of signals. Signals are units of communication and may, for example, be used to specify PDUs. In order to use signals, they have to be defined in a system or block diagram. Signals are defined by putting the **signal** keyword in a text symbol, followed by the names of the signals and the data type of the parameters, if present.



```
SIGNAL
<signal name>(<parameter type>, ...),
<signal name>,  ...;
```

Figure 17: Text symbol with signal definition

Rule 9: The data types of all parameters of all signals relevant to the system and conveyed over normative channels shall be specified.

This rule is motivated by the principle that normative parts shall be clearly and unambiguously indicated.

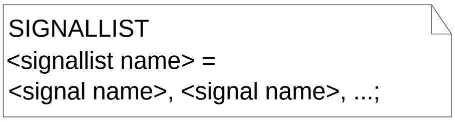
Rule 10: The data types of parameters of signals that are conveyed over normative channels shall have a finite size.

This rule is motivated by the principle of avoiding state space explosion. A parameter that can have an infinite size or an infinite number of values may make the state space of the system infinite.

If it is not possible to give a maximum value at specification time, an externally defined synonym can be used as a maximum value.

8.3.4 Signallist

If the same list of signals appears at several channels and signal routes, it is convenient to declare these signals as a list using the signallist construct (ITU-T Recommendation Z.100 [1], §§ 2.5.5) in a text symbol. At any signal route or channel where this list occurs, the name of the signallist is put between parentheses, instead of typing the names of all the individual signals.



```
SIGNALLIST
<signallist name> =
<signal name>, <signal name>, ...;
```

Figure 18: Signallist definition

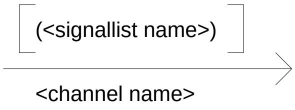


Figure 19: Use of signallist

8.3.5 Select symbol

The select symbol (ITU-T Recommendation Z.100 [1], §§ 4.3.3) is used to indicate that the presence of the symbols contained in the dashed area is dependent on one or more implementation options. The select symbol is useful for major optional implementation capabilities. If the selection expression evaluates to TRUE, the symbols in the dashed area are selected. If the selection expression evaluates to FALSE, the symbols in the dashed area and all channels and signal routes that cross the boundary are left out.

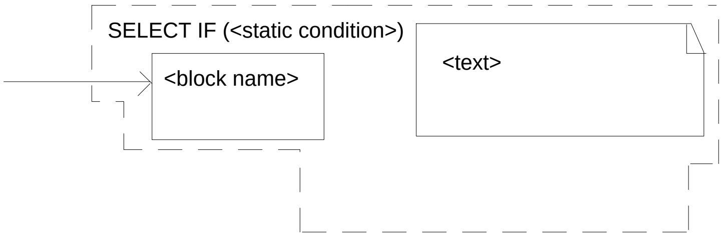


Figure 20: The select symbol

Rule 11: The selection expression in a select symbol shall depend on implementation options.

This rule is motivated by the principle to clearly indicate implementation options in an ETS.

8.3.6 Text symbol

The text symbol (ITU-T Recommendation Z.100 [1], §§ 2.2.8) can be used in any diagram. It is used for many purposes: it can contain comment, definitions of signals, signallists, timers, data types and variables.

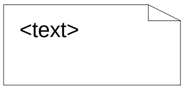


Figure 21: The text symbol

8.3.7 Text extension

When a symbol does not have enough space to contain the text, it is possible to attach a text extension symbol (ITU-T Recommendation Z.100 [1], §§ 2.2.7) to that symbol. The text that is in the text extension symbol is considered to be a continuation of the text in the other symbol.

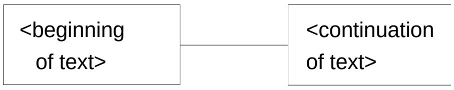


Figure 22: The text extension symbol attached to another symbol

8.3.8 Comment

A comment (ITU-T Recommendation Z.100 [1], §§ 2.2.6) can either be placed in a comment symbol or as text between the /* and */ in other symbols. The comment symbol can be placed in any diagram. It can be attached to any other symbol.

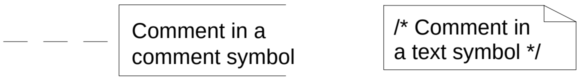


Figure 23: The comment symbol and comment used in another symbol

8.3.9 Procedure symbol

The procedure symbol (ITU-T Recommendation Z.100 [1], §§ 2.4.6) is used in a process diagram to indicate that a procedure is called by that process. If the same procedure is used in several processes in the same block or system, it is possible to place the procedure symbol in the block or system diagram instead.

If a procedure is declared as a remote procedure (ITU-T Recommendation Z.100 [1], §§ 4.14), it can be called from other processes. In that case the procedure symbol contains the keyword **exported**. An example of the use of remote procedures is given in annex C.

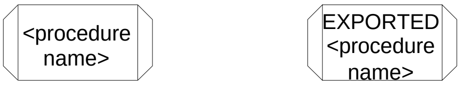


Figure 24: The procedure symbol

8.3.10 Macro call

The macro call symbol (ITU-T Recommendation Z.100 [1], §§ 4.2.3) is a shorthand notation. The macro call symbol is replaced by the symbols in the corresponding macro diagram. A macro call can be used in any diagram. It can also contain parameters. In that case, the formal parameters in the macro diagram are substituted by the actual parameters supplied in the macro call.

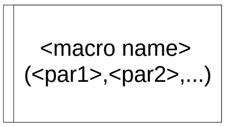


Figure 25: The macro call symbol

In some cases it may be necessary to explicitly indicate how the macro is connected to other symbols. This is done by using the names of the connection points of the corresponding macro definition.

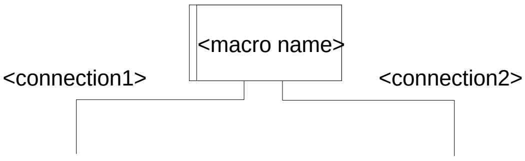


Figure 26: A macro call using the names of the connection points

The use of macros is discouraged. Procedures can usually be used instead.

8.3.11 Synonyms

A synonym (ITU-T Recommendation Z.100 [1], §§ 5.3.1.13) is used to give a name to a value. A synonym is always local to the diagram in which it is defined. The value of a synonym is constant, it cannot be changed. A synonym has to be declared with the **synonym** keyword in a text symbol, as depicted in figure 27.

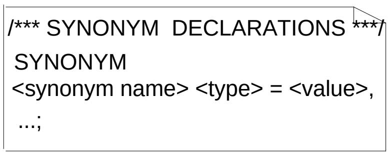

/*** SYNONYM DECLARATIONS ***/
SYNONYM
<synonym name> <type> = <value>,
...;

Figure 27: Declaration of synonyms

If ASN.1 is used in combination with SDL, the type of the synonym can be an ASN.1 type.

8.4 Symbols used in block diagrams

The following symbols can be used in both block diagrams and system diagrams:

- block;
- channel;
- signal definition;
- signallist definition;
- comment;
- data type definition;
- macro call;
- procedure symbol;

- select symbol;
- synonym declaration.

The description of these symbols can be found in subclause 8.3.

The additional symbols that can be used in block diagrams are described in subclauses 8.4.1 to 8.4.4.

8.4.1 Process symbol

The processes within a block shall be referenced in the block diagram. This is done by using the process symbol (ITU-T Recommendation Z.100 [1], §§ 2.4.6).

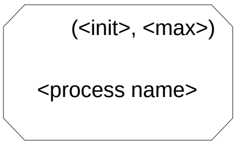


Figure 28: The process symbol

The process symbol contains:

- the name of the process;
- a specification of the number of initial instances of the process at system start up, and the maximum number of allowed instances of the process.

The specification of the behaviour of the process is given in the corresponding process diagram. The interface of the process is described by the signal routes connected to the process symbol.

Rule 12: If the process diagram contains a create symbol, this shall be shown in the block diagram by using the create line symbol.

This rule is motivated by reasons of clarity. A specification that clearly indicates the relations between processes simplifies review by human experts.

Rule 13: In the normative part of a specification, the maximum number of instances of a process shall be limited.

This rule is motivated by the principle of avoiding state space explosion. Allowing an unlimited number of instances of a process may make the state space of the system infinite.

If the maximum number of instances is not known at specification time, the maximum number can, for example, be given as an external synonym related to an entry in the ICS.

8.4.2 Signal route

A signal route (ITU-T Recommendation Z.100 [1], §§ 2.5.2) is used to describe an information flow between two processes. A signal route can convey signals either in one direction or bidirectionally, which is indicated by arrows. For each direction there is a signal list describing which signals can be conveyed by the signal route. The transmission time to convey a signal over a signal route is zero.

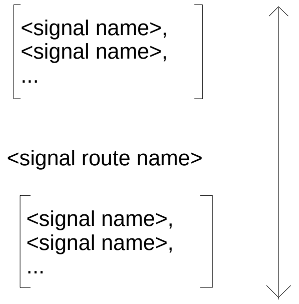


Figure 29: The signal route symbol

8.4.3 Create line

The process create line (ITU-T Recommendation Z.100 [1], §§ 2.4.3) shows that process instances at the beginning of the create line can create new instances of the process at the end of the create line.

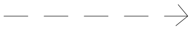


Figure 30: The create line symbol

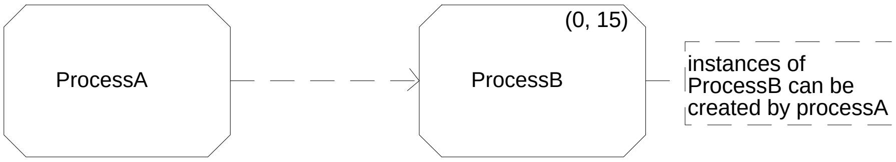


Figure 31: An example of the use of the create line

8.4.4 Connection between channels and signal routes

A signal route that leads to the boundary of a block diagram is connected to a channel of the surrounding block or system diagram in the following way: the name of the channel is put just outside the block boundary near the place where the signal route ends (ITU-T Recommendation Z.100 [1], §§ 2.5.3). If more than one signal route is connected to the same channel, it is possible to give the name of the channel only once, as is shown in figure 32.

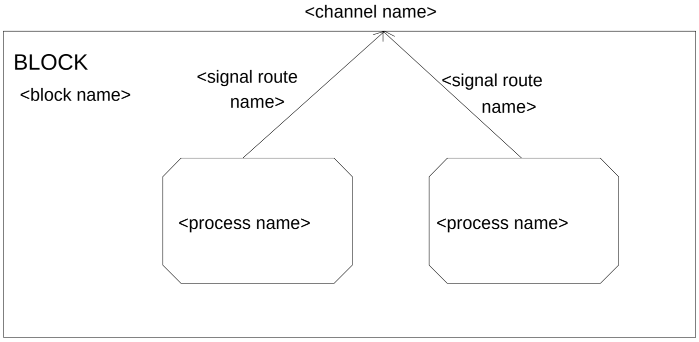
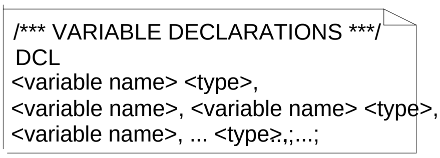


Figure 32: Connection between channel and signal routes

8.5 Symbols used in process diagrams

8.5.1 Variable

A variable (ITU-T Recommendation Z.100 [1], §§ 2.6.1.1) is used to store a value. A variable is always local to the process instance in which it is defined. Global variables (for example global to a block or to a system) do not exist. A variable has to be declared with the **dcl** keyword in a text symbol, as depicted in figure 33.



```
/*** VARIABLE DECLARATIONS ***/
 DCL
<variable name> <type>,
<variable name>, <variable name> <type>,
<variable name>, ... <type>..,;...;
```

Figure 33: Declaration of variables

For validation purposes it is important that a variable cannot get an infinite number of different values. Therefore, finite-sized data structures should be used where possible.

If ASN.1 is used in combination with SDL, the type of the variable can be an ASN.1 type. For example, it is allowed to use "INTEGER (0..31)" in an SDL variable declaration to define a variable that can have integer values 0 to 31.

Every process instance contains a number of predefined variables that are updated implicitly during process execution. These variables are:

Parent The Process instance Identifier (PId) of the process instance that created this instance. If the process was created at system initialisation, the value of Parent equals NULL.

Offspring The PId of the process instance that was most recently created by this instance.

Sender The PId of the process instance that sent the last received signal.

Self The PId of this process instance.

Now A variable of type TIME containing the current system time.

8.5.2 Process start

The process start (ITU-T Recommendation Z.100 [1], §§ 2.6.2) indicates where the execution of a process starts. From the start node a transition is started. The transition can be empty, in which case it directly leads to a state symbol.

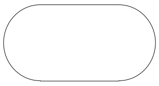


Figure 34: Process start symbol

8.5.3 State

When a state is entered (ITU-T Recommendation Z.100 [1], §§ 2.6.3), the process waits for a transition to be triggered.

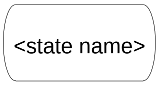


Figure 35: The state symbol

A transition can be triggered in the following cases [2]:

- a signal is received and the signal is covered by an input symbol attached to the state;
- a timer has expired and the timer is listed in an input symbol attached to the state;
- a continuous signal is attached to the state, and the value of its expression evaluates to TRUE;
- a spontaneous transition is attached to the state.

The following events are also handled by the process, but do not trigger a transition:

- a signal is received that is listed in a save symbol attached to the state. The signal is left in the input queue for later use, and the process remains in the same state;
- a signal is received that is not listed in an input or save symbol attached to that state. The signal is discarded, and the process remains in the same state.

The occurrence of automatic discarding of signals during execution of a specification may be an indication that the specification is incomplete. Usually, SDL validation tools are capable of generating warnings if signals are discarded automatically.

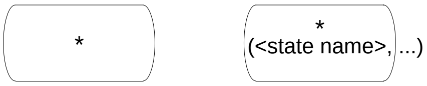


Figure 36: The *-state construct

It is possible to put more than one state name in one state symbol. In this case the transitions which start from that symbol are valid for all the states with the names in the symbol. A shorthand notation is to put a "*" in the state symbol, which denotes all states of the process. This is useful if the process reacts in all states in the same way on certain inputs, e.g. such as is often the case for invalid inputs. By putting a *, followed by a list of signals between parentheses, it can be indicated that all signals except the ones listed in the parentheses have to be saved.

8.5.4 Input

The input symbol (ITU-T Recommendation Z.100 [1], §§ 2.6.4) is used to indicate that the reception of the specified signal triggers the associated transition. The input symbol is shown in figure 37.

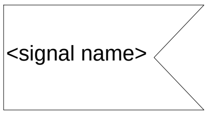


Figure 37: The input symbol

The predefined variable Sender is updated implicitly when a transition is triggered by an input. The new value of Sender is the PId of the process instance that sent the signal.

If the signal contains parameters, there shall be a variable in the input symbol for every signal parameter. These variables receive the values of the actual parameters that are conveyed by the received signal. The input symbol for a signal with parameters is shown in figure 38.

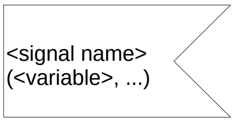


Figure 38: The input symbol for a signal with parameters

[2] This is a slight simplification of the semantics given in Annex F of ITU-T Recommendation Z.100 [1].

It is possible to list more than one signal in the input symbol. If one of the signals is received, the transition starts. It is also possible to put a * in the input symbol, which denotes all signals except those already listed in input or save symbols attached to the state.

8.5.5 Priority input

The priority input symbol (ITU-T Recommendation Z.100 [1], §§ 4.10) is used to indicate that a signal has priority over other signals contained in ordinary input symbols (see subclause 8.5.5). Normally in SDL it is the arrival order of the signals that decides in which order they are processed. The signal which arrives first is the first one to be processed. By using the priority input symbol, it is the signal in the priority input symbol that is processed first, independent of its location in the input queue. The priority input symbol is shown in figure 39.

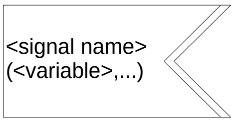


Figure 39: The priority input symbol

In all other respects, there is no difference between the priority input and the normal input.

8.5.6 Save symbol

The save symbol (ITU-T Recommendation Z.100 [1], §§ 2.6.5) is used to bypass the normal removal of incoming messages that are not referred to in input symbols. By putting a * in the save symbol, all signals will be saved, except those listed in inputs attached to that state. By putting a *, followed by a list of signals between parentheses, it can be indicated that all signals except the ones listed in the parentheses have to be saved.

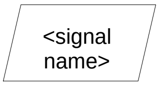


Figure 40: The save symbol

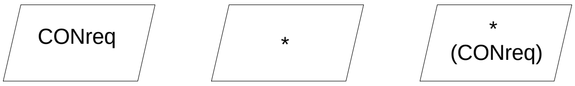


Figure 41: Examples of save symbol

8.5.7 Spontaneous transition

A spontaneous transition (ITU-T Recommendation Z.100 [1], §§ 2.6.6) is used in a process definition to specify that a particular transition can take place without any corresponding input. It appears as an input symbol with the keyword **none**, instead of a signal name. A spontaneous transition can always take place, even if there is a signal in the process queue that can trigger another transition.

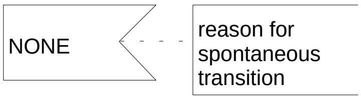


Figure 42: A spontaneous transition

A spontaneous transitions is used when the condition for the associated transition to occur is implementation dependent, or difficult to formalise using SDL.

Rule 14: A comment shall be attached to a spontaneous transition that explains the condition for the transition to fire.

This rule is motivated by reasons of testability. For the production of a test specification it is important to have more information about the reason for a transition to fire.

8.5.8 Continuous signal

The continuous signal symbol (ITU-T Recommendation Z.100 [1], §§ 4.11) is used to specify that the associated transition is triggered if the Boolean expression in the symbol evaluates to TRUE, and the input queue is empty.

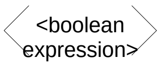


Figure 43: The continuous signal symbol

If there is more than one continuous signal in the same state, then a priority can be provided for each continuous signal. The priority is given as an integer; the lower the number the higher the priority. If more than one continuous signal with the same priority evaluates to TRUE, a non-deterministic choice is made among these. A continuous signal has always a lower priority than an input.

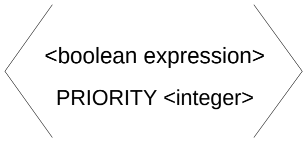


Figure 44: The continuous signal symbol with priority

Rule 15: The Boolean expression in a continuous signal shall not contain NOW, ANY, or remote procedure calls.

The motivation for this rule is given by the principle to avoid state space explosion.

NOTE: The only circumstances in which continuous signals can be effectively used is to test if a certain condition is true, and continue if no other signals can be received.

It is discouraged to use many continuous signals in the same state, since this leads to state space explosion.

8.5.9 Timer

A timer in SDL (ITU-T Recommendation Z.100 [1], §§ 2.8) is used to model a time-out. A timer is set to expire at a certain moment in time. When the timer expires, a signal with the name of the timer is put in the process queue of the process that set the timer. The duration of the time unit can not be specified in SDL: it needs to be supplied using informal text.

A timer needs to be defined in a text symbol in the process diagram.

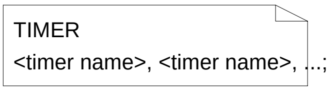


Figure 45: A text symbol with timer definition

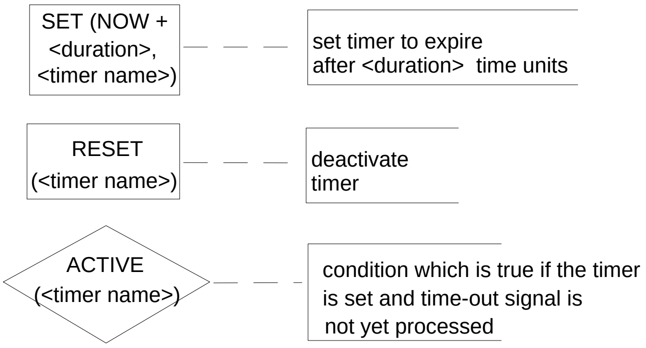


Figure 46: Operations set, reset and timer active

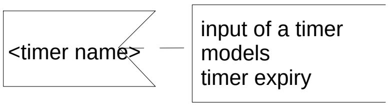


Figure 47: Timer expiry

It is possible to associate a default value with a timer. In that case it is not necessary to provide the duration as a parameter in the set statement. A default timer is set to **now** + the default value.

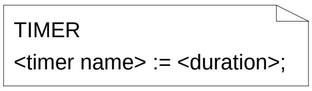


Figure 48: Definition of a timer with a default value

Rule 16: The basic real time associated with a unit of a timer shall be one second.

Rule 17: A timer shall be started as set (now + <Duration>, <TimerName>) or, if a default timer value is specified: set (<TimerName>).

These rules are motivated by reasons of testability. For writing tests related to timers it is necessary to have information about the real time requirement imposed by a timer.

A timer should never be started as: “set(<Time>, <TimerName>)" where <Time> is an absolute value.

8.5.10 Optional transition

The optional transition symbol (ITU-T Recommendation Z.100 [1], §§ 4.3.4) is used to indicate optional behaviour in a branch of a process or procedure description. The optional transition symbol contains a condition that is evaluated statically, i.e. variables are not allowed in the condition.

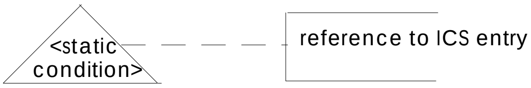


Figure 49: The option symbol

Rule 18: The condition of the option symbol shall depend on implementation options.

This rule is motivated by the principle to clearly indicate implementation options in an ETS.

8.5.11 Task

A task symbol (ITU-T Recommendation Z.100 [1], §§ 2.7.1) contains either assignment statements or informal text. An assignment statement is used to assign values to variables.

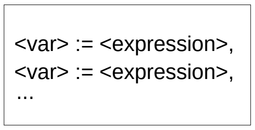


Figure 50: Task symbol containing assignment statements

Rule 19: Task symbols with informal text shall not occur in the normative part of a standard.

This rule is motivated by the principles of correct use of formalisms and clarity. A task box with informal text may be interpreted in several different ways, thus decreasing the clarity of the specification. If task boxes contain informal text, the application of computer-based tools for validation and test derivation is more difficult.

Informal text can be used in draft versions of a standard. But in the final version it shall be replaced by formal text. How informal text can be formalised differs from case to case. Below, three possibilities are listed:

a) replace by an assignment (for example "increment subscriber counter by the call charge" becomes SubscrCounter := SubscrCounter + CallCharge, in which SubscrCounter and CallCharge are variables;

b) replace by a procedure call: "calculate charge" becomes a procedure call to CallCharge, with some relevant parameters added;

c) if the task is too difficult to formalise: introduce non-determinism. For example if it is too difficult to specify how the call charge is computed in "increment subscriber counter by the call charge", this can be formalised by SubscrCounter := SubscrCounter + ANY Real, with the comment /* increment with call charge */ attached. This approach is more difficult to test and validate than possibilities a) and b).

8.5.12 Decision

The decision symbol (ITU-T Recommendation Z.100 [1], §§ 2.7.5) is used to split a transition into two or more branches depending on a condition. A condition can be described either with:

- an expression; or
- the keyword **any**, which means that the decision is non-deterministic.

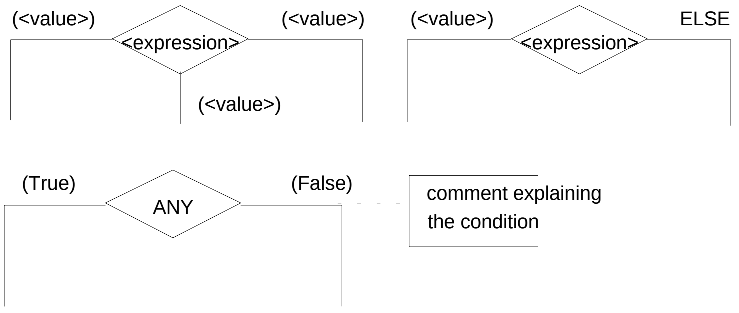


Figure 51: Decision symbol

The non-deterministic decision (ITU-T Recommendation Z.100 [1], §§ 2.7.5) is used to abstract from details of the associated condition. This is useful when it is unfeasible to formally specify the condition, or when the condition depends on implementation specific details. It is also possible to use informal text in the decision symbol.

Rule 20: Decisions with informal text shall not occur in the normative part of the final version of a standard.

This rule is motivated by the principle of clarity. A decision with formal text can only be given one interpretation. Informal text may be interpreted in several different ways, therefore decreasing the clarity of the specification. Furthermore, if all decisions contain formal text, the application of computer-based tools for validation and test derivation is simplified.

There are a number of ways to formalise informal text. Some examples are given in figure 52 and in subclause 8.5.12 on the task symbol.

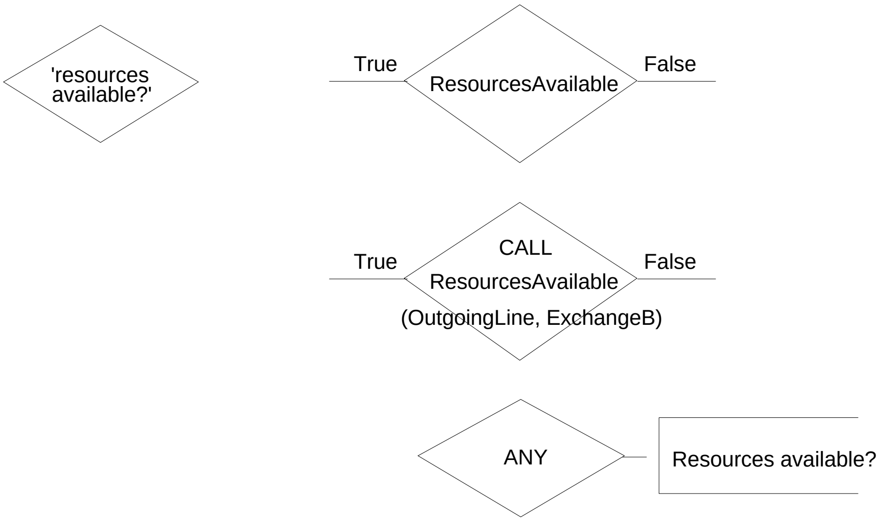


Figure 52: Decision symbol with informal text and possible (formal) replacements

Rule 21: Whenever **any**, is used a comment shall be given explaining how the value is determined.

This rule is motivated by the clarity principle. For the production of a test specification it is important to have more information about the condition for the selection of a branch in the decision.

8.5.13 Process creation

A process creation symbol (ITU-T Recommendation Z.100 [1], §§ 2.7.2) is used to specify dynamic creation of processes. Parameters can be given to the new process. Predefined variable Offspring of the creating process instance is implicitly updated with the PId of the created process instance. If an attempt is made to create more than the maximum number of process instances, no process instance is created and the predefined variable Offspring becomes NULL.

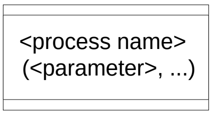


Figure 53: The process creation symbol

8.5.14 Procedure call

The procedure call symbol (ITU-T Recommendation Z.100 [1], §§ 2.7.3) is used to call a procedure.

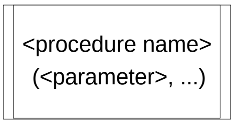


Figure 54: The procedure call symbol

When the procedure call is completed, i.e. the execution has reached the procedure return symbol in the called procedures diagram, the execution continues with the next symbol following the procedure call.

The procedure may return a value as a result. In which case, the procedure call occurs in a decision symbol or a task symbol preceded by the keyword **call**.

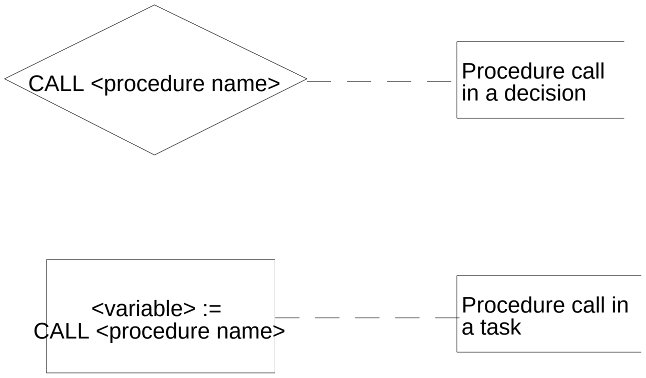


Figure 55: Call of a value returning procedure

8.5.15 Remote procedure call

It is possible to call a procedure that is defined in another process if the procedure is defined as remote (ITU-T Recommendation Z.100 [1], §§ 4.14). When a remote procedure is called, implicitly a signal is sent to the process that exports the procedure. This process executes the remote procedure, and the result is returned to the calling process by sending a signal with result parameters.

The process in which the remote procedure is defined contains a procedure symbol with the keyword **exported**. An incoming remote procedure call is handled as an incoming signal: they are processed on a first come-first served basis. The processing of an incoming remote procedure call can be postponed in a certain state by using a save symbol with keyword **procedure** followed by the name of the procedure.

In order to call a remote procedure, it has to be declared as remote in a text box. The actual call of a remote procedure is the same as a normal procedure call.

8.5.16 Output

The output symbol (ITU-T Recommendation Z.100 [1], §§ 2.7.4) is used to send a signal to another process instance. To which instance it is sent depends on the channels and signal routes between the blocks/processes. However, in some situations this instance may be ambiguous. Different ways to resolve this ambiguity exist, by explicitly addressing the receiver of the signal that is sent:

Output via used to send signals over an explicitly specified channel or signal route. This resolves ambiguity if there is more than one channel that can bear the same signal, and the PId of the receiver is not known. Output via leads to non-determinism if there exist more than one instance of the receiving process;

Output to there are two possible ways to use output to:

1) the name of the receiving process is provided: **output** <sig> **to** <process name>. This is leads to implicit non-determinism if there is more than one instance of that process;

2) the process identifier is provided: **output** <sig> **to** <PId>. This delivers the signal at the right receiver, even if there are more instances of that process;

Output to via a combination of **output to** and **output via.**

It is possible to send a signal to a number of receivers by using the "output via all" construct. With this construct the same signal is sent via every signal route or channel mentioned in the VIA §§. Note that this does not guarantee a full broadcast. If multiple processes are connected to the same signal route, only one process will receive the signal.

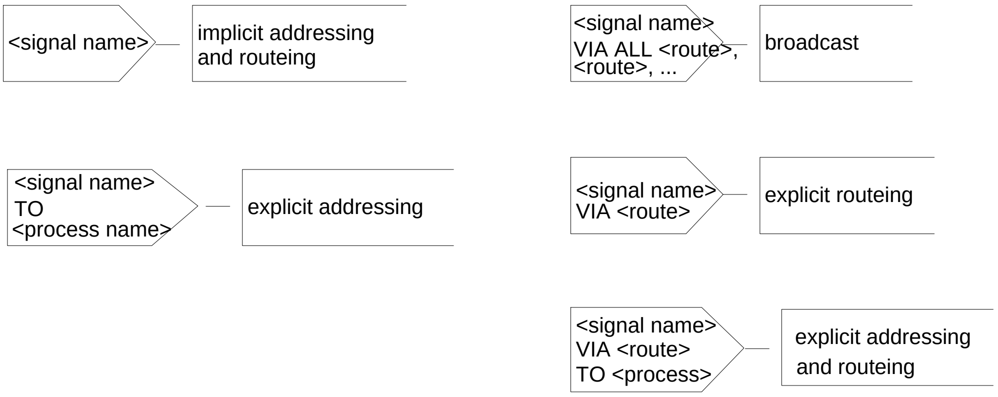

<signal name>
implicit addressing and routeing
<signal name> VIA ALL <route>, <route>, ...
broadcast
<signal name> TO <process name>
explicit addressing
<signal name> VIA <route>
explicit routeing
<signal name> VIA <route> TO <process>
explicit addressing and routeing

Figure 56: Ways to address the receiver of an output

Rule 22: The receiving process instance shall always be uniquely identified in order to avoid non-deterministic behaviour.

This rule is motivated by the principle to avoid implicit non-determinism.

8.5.17 Nextstate

The nextstate symbol (ITU-T Recommendation Z.100 [1], §§ 2.6.8.2.1) is used to terminate a transition and enter a new state of the process. The name of the next state is given in the nextstate symbol.

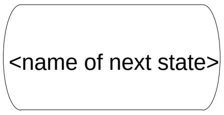


Figure 57: The nextstate symbol

Instead of giving a specific state name in the nextstate symbol, a "-" can be given instead. This means that the next state is the same as the state from which the transition originated, i.e. "return to the same state".

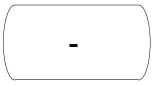


Figure 58: The nextstate symbol with a "-" as the next state

8.5.18 Process stop

The process stop symbol (ITU-T Recommendation Z.120 [3], §§ 2.6.8.2.3) is used to specify that a process instance ceases to exist.

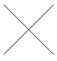


Figure 59: The process stop symbol

8.5.19 Join

The join symbol (ITU-T Recommendation Z.100 [1], §§ 2.6.8.2.2) is a page-layout mechanism that is used to redirect the flow of control. If the execution reaches a join symbol with an incoming arrow, the execution continues at the join symbol with an outgoing arrow that has the same label.

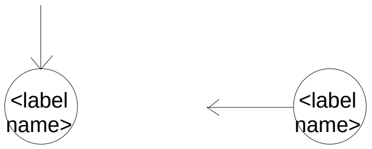


Figure 60: The join symbol with incoming arrow and outgoing arrow

The join construct makes it more difficult to understand the flow of control. Therefore, it should only be used if the normal way of connecting symbols (using solid lines) is not possible.

8.6 Symbols used in procedure diagrams

The following symbols can be used in both process diagrams and procedure diagrams:

- text symbol;
- state;
- input;
- priority input;
- save symbol;
- spontaneous transition;
- timer;
- continuous signal;
- label;
- task;
- process creation;
- procedure call;
- macro call;
- output;
- decision;
- nextstate;
- join;
- optional transition;
- comment;
- variable.

The description of these symbols can be found in subclause 8.5.

In the remainder of this subclause the additional symbols that can be used in procedure diagrams are described.

8.6.1 Procedure start

The procedure start (ITU-T Recommendation Z.100 [1], §§ 2.6.2) indicates where the execution of a procedure starts. From the start node a transition is started. The transition can be empty, in which case it directly leads to a state symbol.

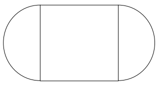


Figure 61: The procedure start symbol

8.6.2 Procedure return

The procedure return symbol (ITU-T Recommendation Z.120 [3], §§ 2.6.8.2.3) is used to specify that the execution of a procedure is stopped. The control is returned to the calling process or procedure, and all variables local to the procedure cease to exist. A procedure can also return a result by attaching a result expression to the procedure return symbol.

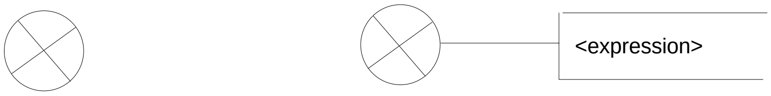


Figure 62: The procedure return symbol, normal (left) and with a value returned as a result (right)

8.7 Symbols used in macro diagrams

All symbols can be used in a macro diagram. There are some special symbols to indicate the connections of the macro.

The use of macros is discouraged.

8.7.1 Macro connections

Macro connections (ITU-T Recommendation Z.100 [1], §§ 4.2.2) are used to specify how the branches connected to the macro call symbol are connected to the branches of the macro diagram. There are two ways to specify macro connections:

a) with a line from a symbol in the macro diagram to the border of the diagram, labelled with a label name;

b) with the macro inlet and outlet symbols. The inlet or outlet symbols indicate the connections of a macro diagram.

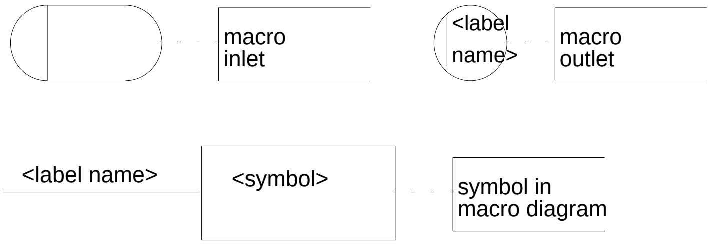


Figure 63: Macro connections

8.8 Data types

SDL contains predefined data types and facilities to let the user define additional data types if the predefined data types are not sufficient. Definition of additional data types can be done in two ways:

- with construction mechanisms; or

- with abstract data type definitions.

This ETS also provides an alternative to the SDL construction mechanisms, which enables specifiers to use ASN.1 to define data structures.

8.8.1 Predefined data

SDL contains the following predefined data types (ITU-T Recommendation Z.100 [1], §§ 5.1.1): Boolean; Character; Integer; Natural; Real; PId; Duration; and Time. The following list shows the predefined data types and related operations. Because they are predefined, they can be used directly in declarations of variables or formal parameters.

Boolean values: True, False

operations:

```
not ... Boolean -> Boolean                          /* logical negation */
... and...  Boolean, Boolean -> Boolean             /* logical and */
... or ...  Boolean, Boolean -> Boolean             /* logical or */
... xor ... Boolean, Boolean -> Boolean             /* exclusive or */
...=>...Boolean, Boolean -> Boolean                 /* logical implication */
... = ...Boolean, Boolean -> Boolean                /* equality */
... /= ...  Boolean, Boolean -> Boolean             /* inequality */
any(Boolean) -> Boolean                             /* gives a random value */
```

Character values: All international alphabet No. 5 (ASCII) characters, e.g. "a", "B", "("

operations:

```
num(...)    Character -> Integer /* gives ASCII code of character */
chr(...)    Integer -> Character /* gives character with given ASCII code */
... = ...Character, Character -> Boolean            /* equality */
... /= ...  Character, Character -> Boolean         /* inequality */
... < ...Character, Character -> Boolean            /* less than */
... <= ...  Character, Character -> Boolean         /* less than or equal */
... > ...Character, Character -> Boolean            /* greater than */
... >= ...  Character, Character -> Boolean         /* greater than or equal */
/* all comparison operators are based on numerical order of ASCII code */
any(Character) -> Character                         /* gives a random value */
```

Charstring values: strings of characters between apostrophes ('), for example 'a string', 'XYz_ 8&*', "

operations:

```
mkstring (...) Character -> Charstring
            /* makes a Charstring from a character */
first (...) Charstring -> Character
            /* gives the first character of a Charstring */
last (...) Charstring -> Character
            /* gives the last character of a Charstring */
length (...) Charstring -> Integer
            /* gives the length of a charstring */
... // ... Charstring, Charstring -> Charstring
            /* concatenates two Charstrings */
... (...) Charstring, Integer -> Character
            /* s(i) returns the i-th character of s (s(1) the first) */
... (...) := ... Charstring, Integer, Character -> CharacterString
            /* s(i) := c changes the i-th character of s to c */
substring (..., ..., ...) Charstring, Integer, Integer -> Charstring
            /* substring (s, i, n) gives substring s(i)..s(i+n-1) */
... = ..., ... /= ..., ... < ..., ... <= ..., ... > ..., ... >= ...
            /* comparison operators based on lexicographic ordering */
```

Integer values: all whole numbers in decimal notation, for example 37, -5, 185

operations:

- ...	Integer -> Integer	/* unary minus */
... + ...	Integer, Integer -> Integer	/* addition */
... - ...	Integer, Integer -> Integer	/* subtraction */
... * ...	Integer, Integer -> Integer	/* multiplication */
... / ...	Integer, Integer -> Integer	/* division */
... mod ...	Integer, Integer -> Integer	/* modulo, remainder of Integer division */
Float(...)	Integer -> Real	/* Integer to Real conversion */
... = ..., ... /= ..., ... < ..., ... <= ..., ... > ..., ... >= ...		/* comparison operators */
any(**Integer**) -> Integer		/* gives a random integer */

Natural values: all positive integers including zero, for example 1958, 11, 6

operations (in every operation on Natural also Integer operands are allowed):

- ...	Integer -> Natural	/* unary minus */
... + ...	Natural, Natural -> Natural	/* addition */
... - ...	Natural, Natural -> Natural	/* subtraction */
... * ...	Natural, Natural -> Natural	/* multiplication */
... / ...	Natural, Natural -> Natural	/* division */
... mod ...	Natural, Natural -> Natural	/* modulo, remainder of Natural division */
Float(...)	Natural -> Real	/* Natural to Real conversion */
... = ..., ... /= ..., ... < ..., ... <= ..., ... > ..., ... >= ...		/* comparison operators */
any(**Natural**) -> Natural		/* gives a random value */

Real values: all real numbers (infinite precision), for example 3.14159265, -37.73

operations:

- ...	Real -> Real	/* unary minus */
... + ...	Real, Real -> Real	/* addition */
... - ...	Real, Real -> Real	/* subtraction */
... * ...	Real, Real -> Real	/* multiplication */
... / ...	Real, Real -> Real	/* division */
Fix(...):	Real -> Integer	/* Real to Integer conversion */
... = ..., ... /= ..., ... < ..., ... <= ..., ... > ..., ... >= ...		/* comparison operators */
any(**Real**) -> Real		/* gives a random real */

PId values: Null, and other non-literal values.

operations:

... = ...	PId, PId -> Boolean	/* equality */
... /= ...	PId, PId -> Boolean	/* inequality */
any(**PId**) -> PId		/* gives a random PId */

Duration values: all values of time intervals (same value notation as Real), for example 5, 1.72358, 6.0828, 0

operations:

```
- ...          Duration -> Duration               /* unary minus */
... + ...Duration, Duration -> Duration           /* addition */
... - ... Duration, Duration -> Duration          /* subtraction */
... * ... Duration, Real -> Duration              /* multiplication */
... * ... Real, Duration -> Duration              /* multiplication */
... / ... Duration, Real -> Duration              /* division */
... = ..., ... /= ..., ... < ..., ... <= ..., ... > ..., ... >= ...
                                                  /* comparison operators */
any(Duration) -> Duration                         /* gives a random duration */
```

Time values: same values as Real, for example 1993.75, 373, 18.5

operations:

```
... + ...Time, Duration -> Time                   /* addition with duration */
... + ...Duration, Time -> Time                   /* addition with duration */
... - ... Time, Duration -> Time                  /* subtraction with duration */
... - ... Time, Time -> Duration                  /* subtraction of times */
... = ..., ... /= ..., ... < ..., ... <= ..., ... > ..., ... >= ...
                                                  /* comparison operators */
any(Time) -> Time                                 /* gives a random time */
```

The use of Integer and Natural is discouraged. Instead, a subrange sort with finite range can be used. If a maximum value is not known, it is possible to use an external synonym as a place holder for the maximum value.

The use of Charstring is discouraged, because the size of Charstring cannot be limited. It is recommended to use an array of characters whenever possible.

8.8.2 User defined data types

If the predefined data types are not adequate, it is possible to define additional data types. Additional data types are defined in text symbols. The scope of the data type definition is restricted to the diagram where the text symbol is placed. For example, if the text symbol is placed in the system diagram, the scope is the whole system, if the text symbol is placed in a process diagram, the scope is restricted to that process. It is also possible to define operations on user defined data types by using value returning procedures.

The remainder of this subclause contains a presentation on how to define data types and operations.

8.8.2.1 Subrange of a predefined data type

The **syntype** construct is used to define a subrange and / or default values of an already defined data type. The operations of the original type are automatically available for the new type. If the result of an operation is out of range of the syntype, then the future behaviour of the system is undefined.

EXAMPLE 1: **syntype** Digit = Natural

constants 0:9

endsyntype Digit;

EXAMPLE 2 : **synonym** MaxIndex Integer = **external**;

```
syntype IndexValues = Integer

    constants 0:MaxIndex

    default 0

endsyntype IndexValues;
```

8.8.2.2 Construction of data types

It is possible to construct data types from existing types. SDL contains a number of built-in construction mechanisms to create data types, e.g. records, arrays, strings and sets:

Struct a record consisting of fields of possibly different types, e.g.

```
    newtype MyStruct struct
        i  Integer;
        b  Boolean;
        r  Real;
    endnewtype;
values: values of all fields between (. and .), e.g.
    (. 5, True, -3.14 .)

operations:

    <variable name>!<field name>              /* access of a field */
    ... = ...Struct, Struct -> Boolean        /* equality */
    ... /= ...    Struct, Struct -> Boolean   /* inequality */
```

Array an array of items of the same type which are indexed by an index type, for example

```
    newtype MyArray array(IndexValues, Boolean) endnewtype;
      /* array of booleans */
values: value of element type between (. and .), for example (. False .) gives an
array with all elements set to False

operations on Array (Indextype, Itemtype):

    ... (...) Array, Indextype  -> Itemtype   /* returns i-th element */
    ... (...) := ...  Array, Indextype, Itemtype  /* changes i-th element */
    ... = ...Array, Array -> Boolean          /* equality */
    ... /= ...    Array, Array -> Boolean     /* inequality */
```

String a string of items of the same type indexed with integer values starting from one. The empty string gets a user defined name, for example,

```
    newtype MyString String(Integer, EmptyString) endnewtype;
    /* defines a string of integers. EmptyString denotes an empty string of
        integers */
values: the only literal value for strings is the emptystring
```

operations on String (Itemtype, emptystring):

```
<variable name>(<integer value>)                /* access of an item */
length(...)   String -> Integer                 /* length of a string */
first(...)    String -> Itemtype                /* first item of a string */
last(...)     String -> Itemtype                /* last item in string */
... // ...String, String -> String              /* concatenation of strings
      */
mkstring(...,...,...) Itemtype -> String        /* make string */
substring(...,...,...) String, Integer, Integer -> String /* get substring,
              substring(s,i,j) gives a string of length j starting at i */
... = ... Array, Array -> Boolean               /* equality */
... /= ...    Array, Array -> Boolean           /* inequality */
```

PowerSet a set of elements of the same type, for example

newtype BoolSet PowerSet(Boolean) **endnewtype**;

values:

operations on Powerset (Itemtype):

```
... in ...      Itemtype, PowerSet -> Boolean     /* is member of */
incl (..., ...) Itemtype, PowerSet -> PowerSet    /* add to set */
del (..., ...)  Itemtype, PowerSet -> PowerSet    /* delete from set */
... and ...     PowerSet, PowerSet -> PowerSet    /* intersection */
... or ...      PowerSet, PowerSet -> PowerSet    /* union */
... = ...PowerSet, PowerSet -> Boolean            /* equality */
... /= ...      PowerSet, PowerSet -> Boolean     /* inequality */
... < ...PowerSet, PowerSet -> Boolean            /* proper subset */
... <= ...      PowerSet, PowerSet -> Boolean     /* subset */
... > ...PowerSet, PowerSet -> Boolean            /* proper superset */
... >= ...     PowerSet, PowerSet -> Boolean     /* superset */
```

8.8.2.3 Abstract data types

In abstract data type definitions (ITU-T Recommendation Z.100 [1], §§ 5.2) new types and operations on these types are defined in combination. Operations are defined using axioms, that state when two expressions are equal.

The use of abstract data type definitions in ETSs is discouraged. It is in general impossible to verify that the type is defined completely (i.e. enough axioms have been defined). Abstract data type definitions tend to be cryptic, and thus hard to understand (against the principle of clarity). Value returning procedures can be used instead to define new operations.

8.8.2.4 User defined operations

Operations on data types can be defined using value returning procedures (see subclause 8.2.4).

9 Message sequence charts concepts

9.1 Introduction

Message sequence charts are used to describe sequences of events that can be performed by the standardised system. MSCs are useful to give an overview of the system functions. MSCs contribute to the testability of a standard, because they may be used to guide the selection of test purposes.

MSCs are closely related to SDL: SDL diagrams give the complete behaviour of the standardised system, while MSCs give typical cases. The sequence of events in an MSC shall be a part of the behaviour that is defined in the SDL diagrams. Consistency between SDL and MSCs can be checked automatically with tools.

This clause defines a subset of MSC for use in ETSs where also SDL is used. The presented rules do not apply to ETSs which use MSCs but do not contain SDL diagrams.

Table 5 gives an overview of the MSC concepts that are allowed in combination with SDL.

Rule 23: The following MSC symbols shall not be used in ETSs: co-region; sub-MSC.

The motivation for this rule can be found in annex B.

NOTE: Not selecting a concept does not mean that the concept is useless in general. For example, sub message sequence charts may be very useful when using MSCs in ETSs that do not contain SDL diagrams.

Table 5: Overview of selection of MSC constructs for use in combination with SDL

unrestricted use allowed	restricted use allowed	not allowed
action comment condition process stop text extension	process creation instance message timer	co-region sub MSC

Rule 24: Message sequence charts shall be used to give at least one example of message exchange for each required system function. The message sequence charts shall also give examples of message exchange in exceptional situations.

This rule is motivated by the principle of clarity. Extensive experience from practice has shown that MSCs are indispensable to give an overview of the system functions that are described in detail by the SDL diagrams.

Rule 25: Message sequences shown in MSCs shall be in accordance with the allowed behaviour of instances specified in the related SDL diagrams.

This rule is motivated by the principle of consistency.

Figure 64 gives an example of a MSC of the multi-party supplementary service in GSM. It is shown that the visitors location register is consulted before a service is provided, and that a service request is rejected if the subscriber is not authorised for use of the service.

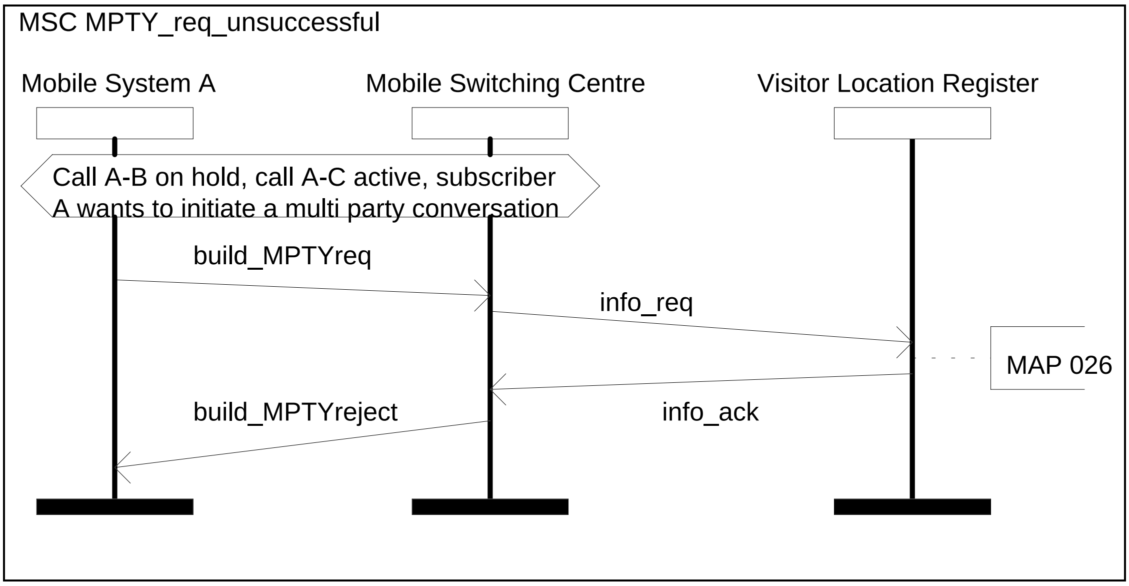
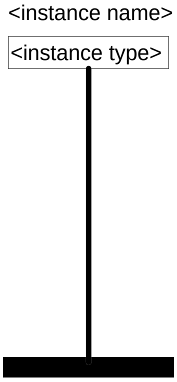
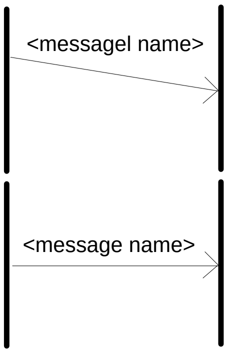
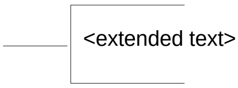
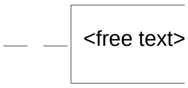
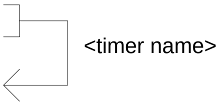
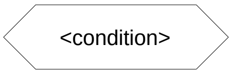
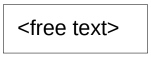
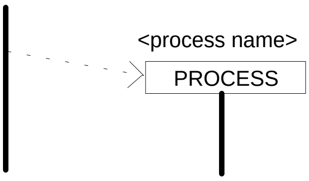
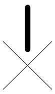

MSC MPTY_req_unsuccessful
Mobile System A
Mobile Switching Centre
Visitor Location Register
Call A-B on hold, call A-C active, subscriber A wants to initiate a multi party conversation
build_MPTYreq
info_req
MAP 026
info_ack
build_MPTYreject

Figure 64: Example of MSC showing an unsuccessful request for a GSM multi party call

9.2 Symbols used in message sequence charts

The symbols that can be used in MSCs are shown table 6.

Table 6: Allowed symbols in MSCs

 **Instance symbol**	 **Message exchange**
 **Text extension symbol**	 **Comment symbol**
 **Timer symbol**	 **Condition symbol**
 **Action symbol**	
 **Process creation line symbol**	 **Process end symbol**

9.2.1 Instance

An instance (ITU-T Recommendation Z.120 [3] §§ 4.2) is a party involved in the communication that is shown in a MSC. Instances can be either part of the environment, or a SDL system, block, or an instance of a process. The instance consists of a heading that gives a name to the instance, an instance axis, to which events are attached, and the termination of the instance.

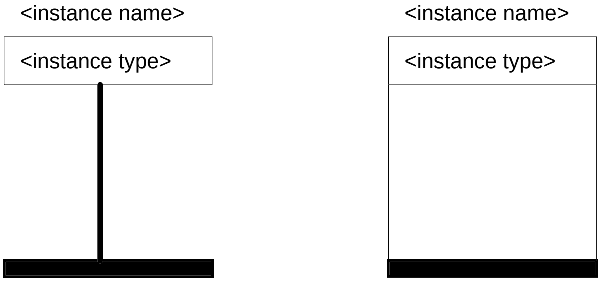


Figure 65: Two MSC instances with different instance axes

Provision of the instance type is optional. It refers to the corresponding SDL diagram where the type of instance is defined, or it shows that the instance is part of the environment. There are the following possibilities:

environment: the instance is part of the environment;

process: the instance refers to an instance of the SDL process with the given instance name;

block: the instance refers to an SDL block with the given instance name;

system: the instance represents the whole system.

The instance axis can appear in two different forms: a single vertical line; or a double vertical line. Both forms are shown in figure 65.

Rule 26: The instances shown in MSCs shall be related to system, blocks, processes or parts of the environment (related to channels connected with the environment) in the related SDL specification.

This rule is motivated by the principle of consistency.

9.2.2 Message

A message (ITU-T Recommendation Z.120 [3], §§ 4.3) represents information exchange between two instances or an instance and the environment. The message symbol is an arrow labelled with the name of the message.

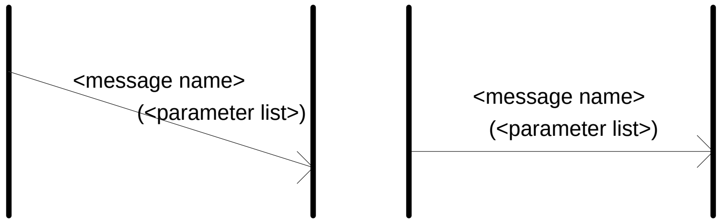


Figure 66: MSC message

Rule 27: Every message in an MSC shall be defined in a signal definition in the related SDL system or block diagram, and as an input and/or output in the related SDL process diagram.

This is because consistency between the SDL and MSC diagrams is required.

9.2.3 Comment

Comments (ITU-T Recommendation Z.120 [3], §§ 2.3) can be placed in the comment symbol attached to the symbol which is commented upon, or in any other symbol between delimiters **/*** and ***/**.

comment symbol
(attached to MSC
symbol)
/* comment in
other symbols */

Figure 67: Comment symbol and alternative way to give comments in other symbols

9.2.4 Timer

In an MSC, a timer (ITU-T Recommendation Z.120 [3], §§ 4.5) is used to show the setting of a timer, and its subsequent timer expiry or timer reset.

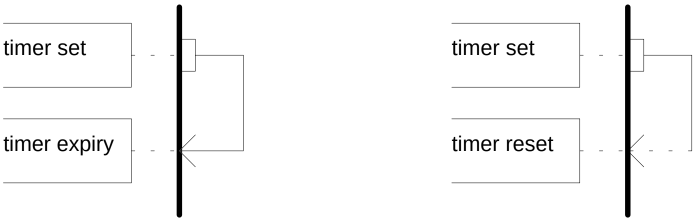


Figure 68: Timer set, timer expiry and timer reset

Rule 28: A timer in an MSC shall be defined as a timer in a process that occurs in the SDL diagram that is related to the MSC instance.

This rule is motivated by the principle of consistency.

9.2.5 Action

An action (ITU-T Recommendation Z.120 [3], §§ 4.6) describes internal processing by a MSC instance. The action symbol is a box containing an informal textual description of the action.

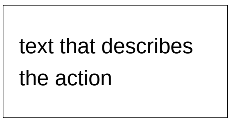


Figure 69: Action symbol

Actions should be considered as comments.

9.2.6 Process creation

Process creation in an MSC (ITU-T Recommendation Z.120 [3], §§ 4.7) is used to indicate that an instance creates a new process instance. The process creation symbol is a dashed arrow that points from the creating instance to the heading of the new instance.

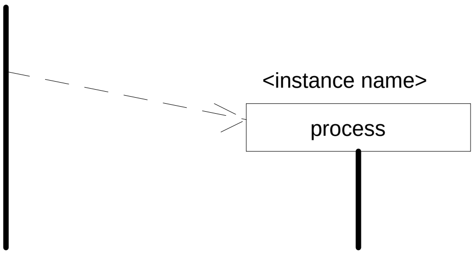


Figure 70: Process creation

Rule 29: The created MSC instance shall be of type "process".

This rule is motivated by the principle of consistency: in SDL only processes can be created.

9.2.7 Condition

A condition (ITU-T Recommendation Z.120 [3], §§ 4.4]) describes either a global system state, referring to all instances contained in the MSC, or a state referring to a subset of instances (non-global condition).

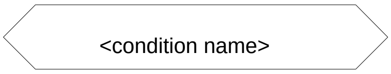


Figure 71: Condition

9.2.8 Process stop

Process stop (ITU-T Recommendation Z.120 [3], §§ 4.8) can be used to indicate that an instance (of a process) ceases to exist. The process stop symbol is identical to the process end symbol that is used in process diagrams.

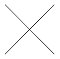


Figure 72: Process stop

10 ASN.1 concepts

ASN.1 is a description technique that is designed to specify data types and values for these data types. By choosing suitable encoding rules (e.g. the basic encoding rules as defined in CCITT Recommendation X.209 [5], or other encoding rules), it can also be specified how values are encoded to bit streams in order to transmit them to another device.

This ETS defines how ASN.1 concepts shall be used in combination with SDL diagrams. Use of ASN.1 in combination with SDL is defined in ITU-T Recommendation Z.105 [2].

The rules presented in this clause do not apply to ETSs which use ASN.1 but do not contain SDL diagrams. The use of SDL without using ASN.1 is permitted, and in this case SDL data types shall be used as described in subclause 8.8.

Table 7 gives an overview of ASN.1 concepts that can be used without restrictions in combination with SDL, concepts that can be used if the rules that restrict their use are met, and concepts that shall not be used at all.

Table 7: Overview of selection of ASN.1 concepts to be used in combination with SDL

unrestricted use allowed	restricted use allowed	not allowed
ASN.1 module definition IMPORTS, EXPORTS NULL BOOLEAN INTEGER REAL ENUMERATED OBJECT IDENTIFIER BIT STRING, OCTET STRING the different character strings ASN.1 comment CHOICE default and optional component subtyping	ASN.1 identifier names SEQUENCE SEQUENCE OF SET OF ASN.1 tags MIN, MAX PLUS-INFINITY MINUS-INFINITY ANY	SET ASN.1 macro (for example the operation macro) value notation of ASN.1 ANY type

Rule 30: The following ASN.1 concepts shall not be used in combination with SDL: ASN.1 SET, ASN.1 macro, value notation of ASN.1 ANY type.

The motivation for this rule can be found in annex B.

NOTE 1: The restrictions in this clause only apply when ASN.1 is used in combination with SDL. The ASN.1 concepts that are not selected for use in combination with SDL may be very useful if ASN.1 is used in ETSs which do not use SDL.

NOTE 2: Z.105 [2] is based on the ASN.1 version as defined in X.680 [7]. X.680 differs in some details from ASN.1 as defined in X.208 [4]. Most of these differences have been listed in this clause.

NOTE 3: Z.105 [2] imposes restrictions on the use of ASN.1 in combination with SDL. For clarity these restrictions have been repeated in this clause.

ASN.1 concepts can occur in the following ways in an ETS:

- in separate ASN.1 modules. The definitions in these modules can be imported in an SDL diagram;
- in SDL diagrams for the definition of data types and values, for example for variables and parameters of messages.

10.1 ASN.1 identifiers

ASN.1 identifiers, or names, can be used in SDL. In SDL, names are case insensitive. For example, "COUNTER", "Counter", and "counter" are considered to be identical in SDL. ASN.1 names are case sensitive. This difference motivates the following rule, which is imposed by Z.105 [2].

Rule 31: The definition of types in ASN.1, whose names only differ in the case of the letters that compose the name shall be avoided.

It is permitted to have a value notation that only differs in the case of the letters from a type notation, or an identifier that only differs in the case of the letters from a value or type notation, i.e. the following is allowed in combination with SDL:

```
X ::= SEQUENCE { x INTEGER, y BOOLEAN }
x X ::= { x 5, y FALSE }
```

SDL allows the construction of names from an alphabet which includes the underscore character ("_") and does not contain the dash character ("-"). For example, message-counter is not a valid name in SDL: it will be interpreted as an operation "subtract value of counter from value of message". ASN.1 allows the construction of names from an alphabet which includes the dash character ("-") and does not contain the underscore character ("_").

This difference motivates the following rules, that are imposed by Z.105 [2].

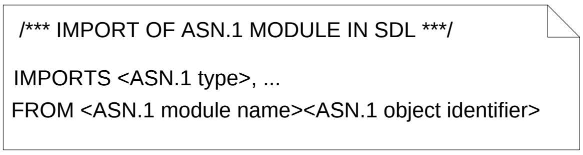
Rule 32: Names of ASN.1 identifiers that are used in combination with SDL shall not contain dash characters ("-").

Rule 33: Names of ASN.1 identifiers that are used in combination with SDL may contain underscore characters ("_").

It is recommended to use underscore characters instead of dash characters to separate parts of identifiers. For example: use `pdu_definitions` instead of `pdu-definitions.`

10.2 ASN.1 import

The ASN.1 IMPORTS concept is used to include modules that are defined elsewhere. The IMPORTS concept can be used in the SDL text symbol in any diagram. Only the type and value notations that are exported by the module (using EXPORTS) are imported in the SDL diagram.



```
/*** IMPORT OF ASN.1 MODULE IN SDL ***/

IMPORTS <ASN.1 type>, ...
FROM <ASN.1 module name><ASN.1 object identifier>
```

Figure 73: Importing an ASN.1 module in an SDL diagram

Rule 34: An ASN.1 module which is directly or indirectly imported into an SDL diagram shall meet the rules stated in this ETS, with the exceptions that names of types or values may contain dash characters.

If a name in an imported ASN.1 module contains dash characters, then in the SDL diagrams the dash characters shall be replaced by underscore characters.

The motivation for the exceptions is to allow the import of existing, pure ASN.1 modules into SDL.

10.3 ASN.1 simple types

All simple types of ASN.1 can be used in SDL diagrams. These types are null, boolean, integer, real, enumerated, bit string, octet string, the different character strings, and object identifier.

The list below shows the simple types, the value notations for these types, and the operations that are available for these types. For every type, the operations = (is equal to) and /= (is not equal to) are available. Both operations yield a boolean value. For ordered types the comparison operations < (is less than), > (is greater than), <= (is less than or equal to), and >= (is greater than or equal to) are also available.

NULL

```
values: NULL
operations:
        ... = ..., ... /= ...,
                /* comparison operators */
```

BOOLEAN

```
values: TRUE, FALSE
operations:
        ... = ..., ... /= ...
                * comparison operators */
        not ... :       BOOLEAN -> BOOLEAN
                /* logical negation */
        ... and ... :   BOOLEAN, BOOLEAN -> BOOLEAN
                /* logical and */
        ... or ... :    BOOLEAN, BOOLEAN -> BOOLEAN
                /* logical or */
        ... xor ... :   BOOLEAN, BOOLEAN -> BOOLEAN
                /* exclusive or */
        ... => ... :    BOOLEAN, BOOLEAN -> BOOLEAN
                /* logical implication */
```

INTEGER

```
values: all whole numbers in decimal notation, for example 4, -123
operations:
        ... = ..., .../=..., ...<..., ...>..., ...<=..., ...>=...
                /* comparison operators */
        - ... :         INTEGER -> INTEGER
                /* unary minus */
        ... + ... :     INTEGER, INTEGER -> INTEGER
                /* addition */
        ... - ... :     INTEGER, INTEGER -> INTEGER
                /* subtraction */
        ... * ... :     INTEGER, INTEGER -> INTEGER
                /* multiplication */
        ... / ... :     INTEGER, INTEGER -> INTEGER
                /* integer division */
        ... mod ... :   INTEGER, INTEGER -> INTEGER
                /* modulo */
        float(...):     INTEGER -> REAL
                /* integer to real conversion */
```

ENUMERATED

values: defined by the user, for example:

```
BasicService ::= ENUMERATED {
         data (1), voice (0), videotelephony(2) }
```

defines values `voice`, `data` and `videotelephony`

operations:

... = ..., .../=..., ...<..., ...>..., ...<=..., ...>=...

/* comparison operators, based on the integer numbers that are supplied with the definition of enumerated values, i.e. in the above example is voice < data < videotelephony */

REAL

values: rational numbers, denoted by 3 integers (the mantissa, the base and the exponent) between brackets, for example {1, 10, -1} (1x10^{-1} = 0.1). The base is either 2 or 10.

operations:

... = ..., .../=..., ...<..., ...>..., ...<=..., ...>=...

/* comparison operators, based on the rational numbers that are denoted by the operand,

i.e. {1, 10, -1} = {10, 10, -2} */

- ...: REAL -> REAL

/* unary minus */

... - ...: REAL, REAL -> REAL

/* subtraction */

... + ...: REAL, REAL -> REAL

/* addition */

... * ...: REAL, REAL -> REAL

/* multiplication */

... / ...:REAL, REAL -> REAL

/* division */

fix (...): REAL -> INTEGER

/* conversion to INTEGER, rounded to near-lowest integer, i.e. fix ({8, 10, -1}) = 0 */

BIT STRING

values: strings of bits, denoted as:

- bit notation, for example `'01011'B, '11'B`
- hexadecimal notation, for example '`58'H, '32C0F'H`
- notation where only '1-bits' are given, for example for

```
SupplServices ::= BIT STRING {
     callForwarding (0), callWaiting (1), threeParty (2)
     }
```

a value is `{callForwarding, threeParty}`, which is the same as `'101'B`

operations:

... = ..., ... /= ...

/* comparison operators */

... and ...: BIT STRING, BIT STRING -> BIT STRING

/* bitwise and operation */

... or ...: BIT STRING, BIT STRING -> BIT STRING

/* bitwise or operation */

length (...): BIT STRING -> INTEGER

/* length of the bit string, i.e. length ('100'B) = 3 */

... // ...: BIT STRING, BIT STRING -> BIT STRING

/* concatenation, i.e. '10'B // '100'B = '10100'B */

octetstring (...): BIT STRING -> OCTET STRING

/* converts bitstring to octet string (padding with 0 to the right, i.e. octetstring ('1'B) = '10000000'B) */

bool (...): BIT STRING -> BOOLEAN

/* converts a bitstring of length 1 to a BOOLEAN:

'1'B to TRUE, '0'B to FALSE */

...(...) BIT STRING, INTEGER -> BIT STRING (SIZE (1))

/* returns the indexed bit, i.e. '01'B(0) gives '0'B */

...(...) := ... BIT STRING, INTEGER, BIT STRING (SIZE (1))
/* changes the indexed bit in a bit string, i.e. if variable
v has value '000'B, than after v(1) := '1'B,
v has value '010'B */
mkstring (...), first (...), last (...), ... // ..., substring (..., ..., ...):
/* same operators as SEQUENCE OF (see section 10.4.3) */

OCTET STRING

values: bit notation and hexadecimal notation (see BIT STRING above)
operations:
... = ..., ... /= ...,
/* comparison operators */
bitstring (...) OCTET STRING -> BIT STRING
/* convert to bit string */
... (...) OCTET STRING, INTEGER -> OCTET STRING (SIZE (1))
/* returns the indexed octet, i.e.
'0AE1C'H(0) = '0A'H, '0AE1C'H(2) = 'C0'H */
... (...) := ... OCTET STRING, INTEGER, OCTET STRING (SIZE (1))
/* changes the indexed octet, i.e. if variable v has value
'0AE1C'H, then after v(2) := '1B'H, v has value
'0AE11B'H */
mkstring (...), length (...), first (...), last (...), ... // ..., substring (..., ..., ...):
/* same operators as SEQUENCE OF (see section 10.4.3)*/

NumericString, PrintableString, TeletexString, VideotexString, VisibleString, IA5String, GraphicString, GeneralString

values: a character string enclosed by double quotes,
e.g. `"A string", "@2%♣∨∨."`
operations:
... = ..., .../=...
/* comparison operators */
...<..., ...>..., ...<=..., ...>=... IA5String, IA5String -> BOOLEAN
/* comparison operators based on lexicographic ordering.
Only defined for IA5String */
... (...) String, INTEGER -> String (SIZE (1))
/* returns the indexed character */
... (...) := ... String, INTEGER, String (SIZE (1))
/* changes the indexed character of a variable */
mkstring (...), length (...), first (...), last (...), ... // ..., substring (..., ..., ...):
/* same operators as SEQUENCE OF (see section 10.4.3). */

OBJECT IDENTIFIER

values: sequences of numbers between brackets, optional starting with an object identifier (see CCITT Recommendation X.208 [4] for more detail), for example
`{ ccitt identified_organisation etsi (0) 45 213 }`
operations:
... = ..., ... /= ...
/* comparison operators */

ANY

values: no values for ANY are allowed (see rule 30)
operations:
... = ..., ... /= ...
/* comparison operators */

The use of MIN, MAX, MINUS_INFINITY or PLUS_INFINITY is discouraged, because their use gives rise to infinite data structures, which complicates formal validation and test generation.

10.4 ASN.1 structured types

The ASN.1 structured types are types which are defined in terms of other types, called component types. A component type can be a simple type, but it can also be a structured type. This makes it possible to construct types of any desired complexity. The ASN.1 structured types that are allowed in SDL diagrams are SEQUENCE, SEQUENCE OF, SET OF and CHOICE.

In ASN.1, it is possible to have recursion in structured data type definitions (e.g. `TypeA ::= SEQUENCE { a TypeA, b BOOLEAN }`). This feature shall not be used in combination with SDL.

Rule 35: Recursive ASN.1 data structures shall not be used in combination with SDL diagrams.

10.4.1 SEQUENCE

The ASN.1 SEQUENCE construct is used to specify records that consist of fields of possibly different types.

type notation:

<type> ::= SEQUENCE { <identifier> <type>; ... }, for example:

```
Subscr_data ::= SEQUENCE {
                    name IA5String,
                    number ISDN_nr }
```

value notation:

{ <identifier> <value>, ...} or { <value>, ... }, for example:

```
{ name "Smith", number "+46-46355240" }
```

operations:

... = ..., ... /= ...
/* comparison operators */

...!<identifier>: SequenceType -> ComponentType
/* component selection. The selection of an optional component that is absent results in an error */

...!<identifier> := ...: SequenceType, ComponentType
/* modification of the specified component, for example
if v is a variable of the above defined Subscr_data that
has the value `{ "Smith",  "+46-46355240" }`:
v!name gives value `"Smith"`, v!number :=
`"+33-92944200"` changes component number */

<identifier> Present (...) SequenceType -> BOOLEAN
/* This operation is only available for components that are
OPTIONAL. The operation gives TRUE if the field is present.
I.e. for
`T ::= SEQUENCE { i INTEGER, b IA5String OPTIONAL }`
bPresent ({5, "text"}) = TRUE, bPresent ({37}) = FALSE */

The following rules are imposed by Z.105 [2], and motivated by the fact that Z.105 [2] is based on X.680 [7], and not on X.208 [4].

Rule 36: In a type definition, a name shall be provided for every component type of an ASN.1 SEQUENCE type.

Rule 37: In a value definition, a name shall be provided for every component type of an ASN.1 SEQUENCE type.

i.e. in the above defined type Subscr_data, value `{ "Smith", "+46-46355240" }` is not permitted to be used.

10.4.2 Default and optional components in SEQUENCE

Default values for components of an ASN.1 SEQUENCE type are allowed. In value notations the value of a default component does not need to be specified, in which case, that component will have the default value.

In ASN.1, components of SEQUENCE can also be made optional. In value notations the value of an optional component does not need to be specified, in which case, no assumptions can be made on the value of that component.

An example of a SEQUENCE with an optional component and a default component is given below.

```
PDU ::= SEQUENCE {
     dataField      DataField,
     sequenceNr     INTEGER (0..65535),
     checkSum       CheckSum OPTIONAL,
     expedited      BOOLEAN DEFAULT FALSE }
```

In order to access an optional component in SDL, it is first necessary to check whether the component is present, using the operation present. Figure 74 shows a fragment of an SDL process which checks whether the checksum is correct for the variable pdu of the above defined type PDU.

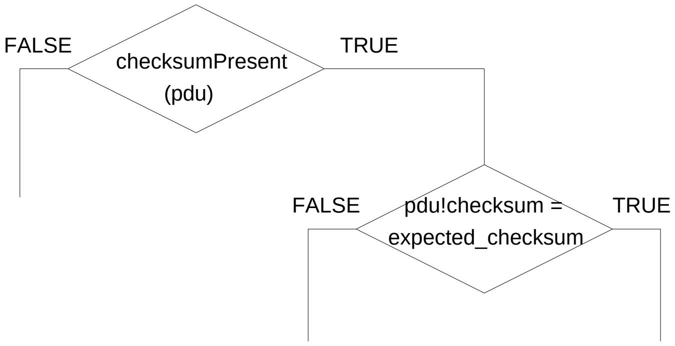


Figure 74: A fragment of a process that accesses an optional component

10.4.3 SEQUENCE OF

The ASN.1 SEQUENCE OF construct is used to specify strings of items of the same type. By supplying a fixed SIZE constraint, arrays can be specified.

type notation:
<type> ::= SEQUENCE OF <component type>, for example
`Int_array ::= SEQUENCE SIZE (4) OF INTEGER`

value notation:
{ <value>, ... }, for example
`{ -3, 5, 0, 234 }` is a value for `Int_array` that is defined above

operations:
... = ..., ... /= ...,
/* comparison operators */
...(...) SEQUENCE OF <type>, INTEGER -> <type>
/* Returns the indexed value (index of first element is 0) */
...(...) := ... SEQUENCE OF <type>, INTEGER, <type>
/* modification of a component, for example (suppose v is a variable of the above defined `Int_array`, and has value `{-3, 5, 0, 234}`): v (0) (gives value -3), `v(3) := -100` (v gets value `{-3, 5, 0, -100}`) */
mkstring (...): <type> -> SEQUENCE OF <type> (SIZE (1))
/* makes a string from one item */
length (...): SEQUENCE OF <type> -> INTEGER
/* gives the length of a sequence */

```
first (...): SEQUENCE OF <type> -> <type>
            /* gives first element of a sequence */
last (...): SEQUENCE OF <type> -> <type>
            /* gives last element of a sequence  */
... // ...: SEQUENCE OF <type>, SEQUENCE OF <type> ->
                        SEQUENCE OF <type>
            /* concatenates two strings  */
substring(..., ..., ...): SEQUENCE OF <type>, INTEGER, INTEGER ->
                        SEQUENCE OF <type>
            /* substring (s, i, n) gives a sequence of length n starting from the i-th
            element of s */
```

It is recommended that types based on SEQUENCE OF have a SIZE constraint. Infinite data types cause state space explosion, which complicates validation and test generation.

10.4.4 SET OF

The ASN.1 SET OF construct is used to specify sets of items of the same type.

type notation**:**

```
<type> ::= SET OF <component type>, for example
      Int_set ::= SET SIZE (0..6) OF INTEGER
      defines a set that can contain maximally 6 elements
```

value notation:

```
{<value>, ... }, for example
      { 2, 0, 0, -5, -134 } is a value for Int_set that is defined above
```

operations**:**

```
... = ..., ... /= ...
            /* comparison operators */
... < ..., ... <= ...
            /* <: is proper subset of, i.e. {0, 1} < {0, 0, 1},
            <=: is subset of */
... > ..., ... >= ...
            /* >: is proper superset of, >=: is superset of */
... in ...: <type>, SET OF <type> -> BOOLEAN
            /* is element of */
incl (.., ..):    <type>, SET OF <type> -> SET OF <type>
            /* add element to set, i.e. incl (0, {0, 1}) = {0, 0, 1} */
del (..., ...):   <type>, SET OF <type> -> SET OF <type>
            /* delete element from set, i.e. del (0, {0, 0}) = {0} */
```

NOTE: There is a difference between the ASN.1 set (SET OF) and the SDL set (PowerSet, see ITU-T Recommendation Z.100 [1], annex D]). In ASN.1, the number of occurrences of the same element in a set plays a role whereas, in SDL this is not the case. For example, for a SET OF INTEGER, the values {0, 1} and {0, 0, 1} are different ({0,1} /= {0, 0, 1}), while for a PowerSet (INTEGER) (the SDL set) these values are equal.

It is recommended that types based on SET OF have a SIZE constraint. Infinite data types cause state space explosion, which complicates validation and test generation.

10.4.5 CHOICE

The ASN.1 CHOICE construct is used to specify that a type can have values of one of a set of alternative types.

type notation:

```
<type> ::= CHOICE { <identifier> <type>, ... },
for example
PartyNumber ::= CHOICE {
          dataPartyNumber    [0] DataPartyNumber,
                        telexPartyNumber   [1] TelexPartyNumber,
                        privatePartyNumber [2] PrivatePartyNumber }
```

value notation:

<identifier> : <component value>, for example
`dataPartyNumber : 94932665`

operations:

... = ..., ... /= ...
/* comparison operators */

...!<identifier>: ChoiceType -> ComponentType
/* extraction of value
for example: suppose v is a variable of the above defined PartyNumber,
and has value `dataPartyNumber : 94932665`
`v!dataPartyNumber` gives value 94932665.
`v!telexPartyNumber` gives an error. */

..!present: ChoiceType -> <identifier>
/* indicates which identifier was chosen by giving its identifier,
i.e. for the above variable v, `v!present = dataPartyNumber` */

<identifier>Present (...): ChoiceType -> BOOLEAN
/* gives TRUE if the component was selected, i.e. for the above variable v,
`dataPartyNumberPresent(v)` = TRUE,
`telexPartyNumberPresent(v)` = FALSE */

In order to access a value of a choice type, it is necessary to first determine, via the !present operator, which alternative component has been given a value. The SDL fragment in figure 75 shows how values of a choice type are accessed.

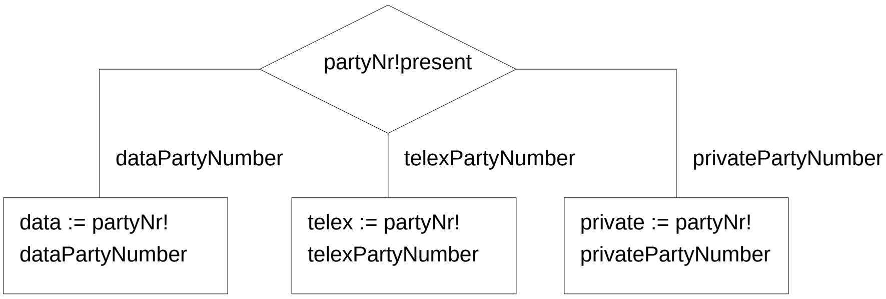


Figure 75: A fragment of a process that accesses the components of a choice type

10.5 ASN.1 subtypes

The ASN.1 subtype mechanisms are used to limit the values of an existing type. There are several subtype mechanisms available for simple types, and for structured types. All operators of the original type are available for the subtype. If the result of an operation on a subtype is out of range, the future behaviour of the system is undefined.

10.5.1 Subtyping of simple types

The following ways to restrict the possible values of a simple type are available:

Single value subtyping the subtype consists of the mentioned values alone, e.g.

```
SignType ::= INTEGER (-1 | 0 | 1)
```

Contained subtyping the subtype consists of all values of another (sub)type, together with possibly some other values, e.g.

```
ExtendedSignType ::= INTEGER (INCLUDES SignType | -2 | 2)
```

The values of ExtendedSignType are -2, -1, 0, 1 and 2.

Value range subtyping to define a sub range of another type, e.g.

```
ShortInt ::= INTEGER (-32768..32767)
```

Size range subtyping the type is restricted to values with a limited size, e.g.

```
Signal_Unit ::= BIT STRING (SIZE (1..28))
```

Values of the above defined Signal_Unit shall be between 1 and 28 bits long.

Alphabet limitation subtyping the values of a character string are limited to a defined subset of characters, e.g.

```
FileName ::= PrintableString (SIZE (12)) (FROM
     ("A"|"B"|"C"|"D"|"E"|"F"|"G"|"H"|"I"|"J"|"K"|"L"|"M"|
      "N"|"O"|"P"|"Q"|"R"|"S"|"T"|"U"|"V"|"W"|"X"|"Y"|"Z"|
      "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"|"."))
```

This example shows a combination of a size range and an alphabet limitation.

It is recommended to use subtyping as much as possible to restrict the size of data types. For example, the use of INTEGER, other than in combination with a finite value range subtype is discouraged. BIT STRING, OCTET STRING, and all ASN.1 character string types are recommended to always have a size constraint.

The use of infinite size data types gives rise to state space explosion which complicates validation and test generation. Furthermore, if infinite data types are used for variables and finite data types are used for signal parameters, the use of conversion operators is required in assignments of variable values to signal parameters. This may have a negative effect on readability of the standard.

If the maximum value or size of a data type is not known, an external synonym can be specified as a place holder for this value. The ICS proforma should contain an entry that asks for the maximum value supported by a product.

10.5.2 Subtyping of structured types

Subtyping of structured types, called inner subtyping (CCITT Recommendation X.208 [4]), can be used to subtype some or all of the component types of a SEQUENCE. Also constraints can be imposed on the presence or absence of optional components. Inner subtyping is useful in the definition of messages where the contents or presence of some fields is dependent on whether some option in the specification has been chosen or not.

The presence constraint ABSENT means that the corresponding component is never present in values of the inner subtype. The presence constraint PRESENT means that the corresponding component shall always be present in values of the inner subtype. Presence constraint OPTIONAL means that the component may or may not be present in values of the inner subtype.

Both the full specification style and partial specification style for inner subtyping are allowed (see CCITT Recommendation X.208 [4]).

An example of inner subtyping using the full specification style is given below.

```
PDU ::= SEQUENCE {
     dataField     [0] DataField,
     sequenceNr    [1] INTEGER (0..65535),
     checkSum      [2] CheckSum OPTIONAL }

/*
*  Class0_PDU is a PDU with a smaller sequence number,
*  and in which there is no checksum present
*/
Class0_PDU ::= PDU (
          WITH COMPONENTS {
          dataField,
          sequenceNr    (0..255),
          checkSum      ABSENT }
          }
```

10.6 ASN.1 tags

Tags are used in ASN.1 in order to distinguish types. When used in SDL, they only have a meaning in choice types in order to distinguish which alternative component has been assigned a value. In other ASN.1 constructs their use is allowed, but they have no meaning and are ignored. Tags are not necessary because the encoding is outside of scope of the SDL formalism.

10.7 ASN.1 useful types

ASN.1 provides a small set of so-called "useful types". Examples are types for denoting time, and a type that can have externally defined values. These types are defined in terms of other ASN.1 types. They can also be used in combination with SDL. The values and operations on these types can be directly derived from the way they are defined.

Annex A (normative): Summary of use of ITU SDL 1992 in European Telecommunication Standards

A.1 Selection of SDL concepts

Table A.1: Use of SDL concepts

Concept		Full use allowed	Restricted use allowed	Use prohibited
Abstract data types		allowed, but use of axioms is discouraged		
Asterisk input		full use allowed		
Asterisk save		full use allowed		
Asterisk state		full use allowed		
Block diagram			use restricted by rules 5, 12 and 13	
Block partitioning			use restricted by rule 6	
Channel with delay			use restricted by rules 7 and 8	
Channel without delay			use restricted by rules 8	
Channel partitioning				use prohibited
Comment		full use allowed		
Connection		full use allowed		
Continuous signal			use restricted by rule 15	
Create		full use allowed		
Dash nextstate		full use allowed		
Data definition constructs			use restricted by rules 9 and 10	
Data type definitions			use restricted by rules 9 and 10	
Decision			use restricted by rules 20 and 21	
Enabling condition				use prohibited
Expressions			use restricted by rule 15	
External synonym		full use allowed		
Imported and exported value				use prohibited
Input		full use allowed		
Internal input and output				use prohibited
Implicit transition		full use allowed		
Join		full use allowed		
Label		full use allowed		
Macro call		allowed, but use is discouraged		
Macro definition		allowed, but use is discouraged		
Nextstate		full use allowed		
Optional definition			use restricted by rule 11	
Optional transition			use restricted by rule 18	
Output			use restricted by rule 22	
Package		full use allowed		

Table A.1: Use of SDL concepts (concluded)

Concept		Full use allowed	Restricted use allowed	Use prohibited
Predefined data			use restricted by rule 10	
Priority input		full use allowed		
Procedure		full use allowed		
Procedure call		full use allowed		
Process symbol			use restricted by rule 19	
Process diagram		full use allowed		
Referenced definition		full use allowed		
Refinement				use prohibited
Remote procedures		full use allowed		
Return		full use allowed		
Save		full use allowed		
Select			use restricted by rule 11	
Service				use prohibited
Signal			use restricted by rules 9, 10, 27	
Signal list definition		full use allowed		
Signal route		full use allowed		
Simple expression		full use allowed		
Spontaneous transition			use restricted by rule 14	
Start		full use allowed		
State		full use allowed		
Stop		full use allowed		
System diagram			use required, and restricted by rule 3 and 4	
Task			use restricted by rule 19	
Text extension		full use allowed		
Text symbol		full use allowed		
Timer			use restricted by rules 16 and 17	
Values and literals			use restricted by rule 2	
Variable		full use allowed		

A.2 List of rules

Rule 1: In the printed version of an ETS only the graphical representation of SDL shall be used.

Rule 2: The following SDL symbols shall not be used in ETSs: channel partitioning, signal refinement, enabling condition, internal input and output, view and reveal, import and export, service, and name class literals.

Rule 3: A system diagram shall be used to describe how the system is composed of functional units (modelled with blocks).

Rule 4: In a system diagram, the blocks, channels shall appear on page 1. Signal definitions shall come before the data definitions.

Rule 5: Block diagrams shall be used to describe how the functional units are composed of processes or blocks.

Rule 6: A block diagram in the normative part of a specification shall not have alternative sub-block definitions, i.e. the feature of SDL (ITU-T Recommendation Z.100 [1], §§ 3.2.2) to describe the decomposition of a block in blocks as well as in processes is not allowed.

Rule 7: Implicit non-determinism arising as an effect of using channels with delay shall be avoided.

Rule 8: To every normative channel of the standardised system a comment "normative" shall be attached.

Rule 9: The data types of all parameters of all signals relevant to the system and conveyed over normative channels shall be specified.

Rule 10: The data types of parameters of signals that are conveyed over normative channels shall have a finite size.

Rule 11: The selection expression in a select symbol shall depend on implementation options.

Rule 12: If the process diagram contains a create symbol, this shall be shown in the block diagram by using the create line symbol.

Rule 13: In the normative part of a specification, the maximum number of instances of a process shall be limited.

Rule 14: A comment shall be attached to a spontaneous transition that explains the condition for the transition to fire.

Rule 15: The Boolean expression in a continuous signal shall not contain NOW, ANY, or remote procedure calls.

Rule 16: The basic real time associated with a unit of a timer shall be one second.

Rule 17: A timer shall be started as "set(now + <Duration>, <TimerName>)", or if a default timer value is specified: set(<TimerName>).

Rule 18: The condition of the option symbol shall depend on implementation options.

Rule 19: Task symbols with informal text shall not occur in the normative part of a standard.

Rule 20: Decisions with informal text shall not occur in the normative part of the final version of a standard.

Rule 21: Whenever **any** is used, a comment shall be given explaining how the value is determined.

Rule 22: The receiving process instance shall always be uniquely identified in order to avoid non-deterministic behaviour.

Rule 23: The following MSC symbols shall not be used in ETSs: co-region, sub-MSC.

Rule 24: Message sequence charts shall be used to give at least one example of message exchange for each required system function. The message sequence charts shall also give examples of message exchange in exceptional situations.

Rule 25: Message sequences shown in MSCs shall be in accordance with the allowed behaviour of instances specified in the related SDL diagrams.

Rule 26: The instances shown in MSCs shall be related to system, blocks, processes or parts of the environment (related to channels connected with the environment) in the related SDL specification.

Rule 27: Every message in an MSC shall be defined in a signal definition in the related SDL system or block diagram, and as an input and/or output in the related SDL process diagram.

Rule 28: A timer in an MSC shall be defined as a timer in a process that occurs in the SDL diagram that is related to the MSC instance.

Rule 29: The created MSC instance shall be of type "process".

Rule 30: The following ASN.1 concepts shall not be used in combination with SDL: ASN.1 SET, ASN.1 macro, value notation of ASN.1 ANY type.

Rule 31: The definition of types in ASN.1, whose names only differ in the case of the letters that compose the name shall be avoided.

Rule 32: Names of ASN.1 identifiers that are used in combination with SDL shall not contain the dash character ("-").

Rule 33: Names of ASN.1 identifiers that are used in combination with SDL may contain underscore characters ("_").

Rule 34: An ASN.1 module which is directly or indirectly imported in an SDL diagram shall meet the rules stated in this ETS, with the exceptions that names of types or values are allowed to contain dashes.

Rule 35: Recursive ASN.1 data structures shall not be used in combination with SDL diagrams.

Rule 36: In a type definition, a name shall be provided for every component type of an ASN.1 SEQUENCE type.

Rule 37: In a value definition, a name shall be provided for every component type of an ASN.1 SEQUENCE type.

Annex B (informative): Motivation for exclusion of SDL, MSC and ASN.1 concepts

This annex provides a motivation for why some SDL, MSC, and ASN.1 concepts are prohibited for use in ETSs.

B.1 Motivation for exclusion of SDL concepts

B.1.1 Channel partitioning

Channel partitioning (ITU-T Recommendation Z.100 [1], §§ 3.2.3) is an alternative for the normal way of sub structuring in which a block is partitioned into lower level blocks or processes. It introduces consistency requirements, for example that the substructured channel and the block have the same signal sets. The construct is forbidden because of the consistency principle of clause 6.

Channel partitioning can be easily replaced by putting the channel substructure as a block, and replace the substructure by two channels leading to the block.

B.1.2 SDL signal refinement

Signal refinement (ITU-T Recommendation Z.100 [1], §§ 3.3) provides a way to hide low-level signals on higher levels of abstraction (for example at system level). Signal refinement is prohibited for use because of the principle of having one level of abstraction.

The signal refinement can be replaced by putting the low-level signals at the channel definitions, instead of the abstract signals.

B.1.3 SDL service

The SDL service interaction diagram (ITU-T Recommendation Z.100 [1], §§ 2.4.5) is related to a process definition, with some exceptions, for example that the services associated with one process share the same input port, and they execute interleaved. The SDL service diagram shall not be used because it has a complicated formal SDL semantics, which makes it hard to understand (clarity principle), especially when there are several services associated with the same process. Also the name service is confusing, since the term service as it is used in telecommunication (telecommunication service, teleservice, bearer service, OSI service) has a different meaning than the SDL service.

There exist several alternatives for the SDL service diagram, for example:

- replace the corresponding process diagram of the service by a block diagram, and the attached service diagrams by process diagrams;
- use several processes instead of just one. A difference is that the processes have no access to each other's variables, while services have access to the variables of their process;
- use procedures.

B.1.4 Revealed and viewed variable

The value of a revealed variable is visible for other processes in the same block, that have defined this variable as being VIEWED. A revealed and viewed variable is essentially a shared piece of memory. This is a dangerous notion, since it cannot be controlled when another process views a variable that is being changed. Unpredictable behaviour of the system may result, as is shown in figure B.1, where viewed variable x is not 0 at the time that the decision is made. However the process that reveals x may set x to 0 after the decision and before the task. This has the result that in the task, 1 is divided by 0, and a runtime error results.

From the perspective of testability it is not acceptable that the behaviour of the specification is unpredictable (implicit non-determinism principle). Hence, the use of viewed/revealed variables are prohibited.

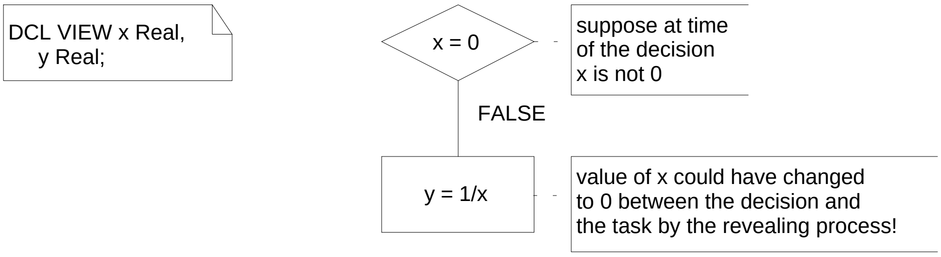

DCL VIEW x Real,
y Real;
x = 0
suppose at time of the decision x is not 0
FALSE
y = 1/x
value of x could have changed to 0 between the decision and the task by the revealing process!

Figure B.1: Danger of view/reveal

Viewed and revealed variables can be replaced by explicit signal exchange to get the value of a variable in another process, or by introducing a remote procedure call for this purpose.

B.1.5 Imported and exported variable

The imported/exported variable mechanism (ITU-T Recommendation Z.100 [1], §§ 4.13) is used for importing the value of a variable that is defined in another process. To use the import/export mechanism on a variable it can be defined as **remote**. The remote definition needs to be given in the block diagram if the importing/exporting processes are located in the same block, or on the system level if the processes are located in different blocks.

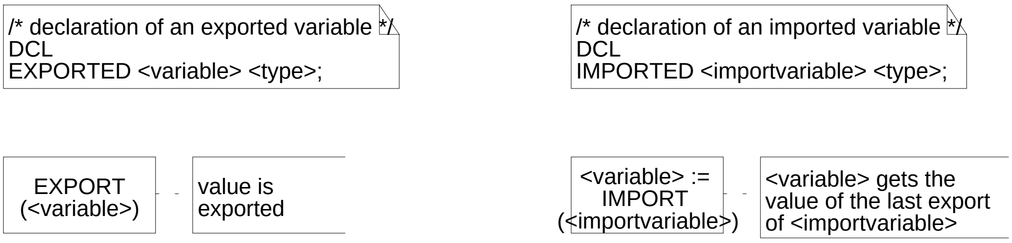

/* declaration of an exported variable */
DCL
EXPORTED <variable> <type>;
/* declaration of an imported variable */
DCL
IMPORTED <importvariable> <type>;
EXPORT
(<variable>)
value is exported
<variable> :=
IMPORT
(<importvariable>)
<variable> gets the value of the last export of <importvariable>

Figure B.2: Declarations and operations to export and import values of variables

The motivation to prohibit imported/exported variables is given by the principle of clarity.

Imported and exported variables can be replaced by explicit signal exchange to get the value of a variable in another process, or by introducing a remote procedure call for this purpose.

B.1.6 SDL internal input and output

SDL internal input and output (ITU-T Recommendation Z.100 [1], §§ 2.9) are outdated concepts, that are only in Z.100 [1] for reasons of upward compatibility. In ITU-T Recommendation Z.100 [1] their future use is discouraged. For this reason these concepts are prohibited for use in ETSs. Normal input and output can be used instead.

B.1.7 SDL enabling condition

An enabling condition can be used in a process graph, directly following an input symbol (ITU-T Recommendation Z.100 [1], §§ 4.12). The branch that contains the condition can only be entered if the expression in the condition evaluates to true. If this is not the case, the input signal is saved.

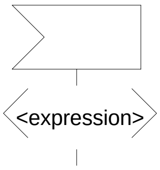


Figure B.3: Enabling condition symbol

The interpretation of enabling condition relies on a complicated semantics of queue contexts, which makes it hard to understand (clarity principle).

Instead, the state to which the enabling condition is attached can be mapped on two states: one in which input symbol is present (the expression in the enabling condition is true), another one in which the input symbol is not present (the expression in the enabling condition is false). Nextstate symbols to that state can be replaced by a decision symbol containing the expression of the enabling condition, with the **true**-branch leading to the state with the input present, and the **false**-branch leading to the state with the input missing.

B.1.8 Name class literals

Name class literals (ITU-T Recommendation Z.100 [1], §§ 5.3.14) provide a means to define a (possibly infinite) number of values for a data type by means of regular expressions.

For example **nameclass** ('x' or 'y') +

defines values x, y, xx, xy, yx, yy, xxx, etc.

The motivation to prohibit name class literals are given by the principles of correct use of formalisms: the name class literals construct is very difficult to implement in tools.

B.2 Motivation for exclusion of MSC concepts in combination with SDL

B.2.1 MSC co-regions

A co-region in a message sequence chart (ITU-T Recommendation Z.120 [3], §§ 6.9) can be used to indicate that the signals attached to the co-region can arrive in any order, although the MSC depicts only a specific order.

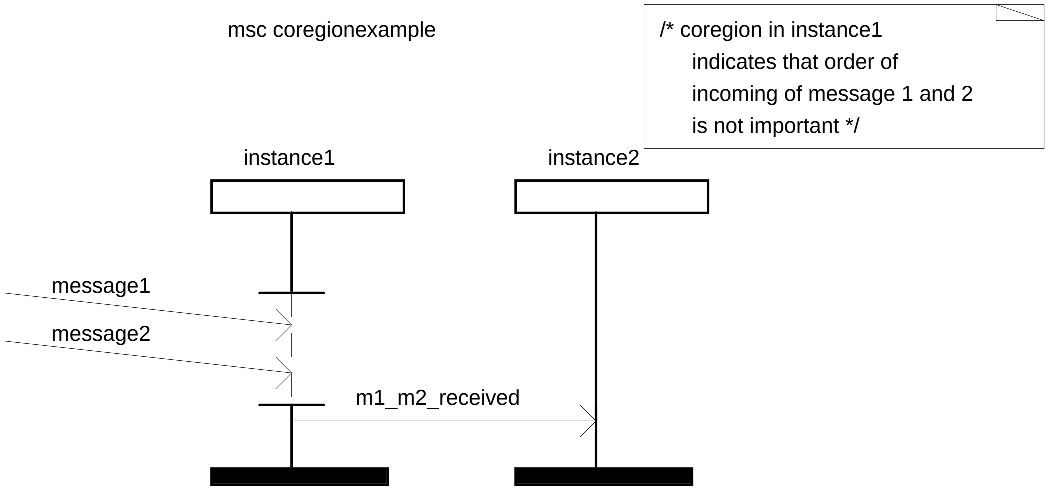


Figure B.4: Co-region symbol

The semantics of the co-region concept is questionable. For example, it is not clear what the co-region means for sending of signals. For this reason the co-region is not used.

An MSC containing a co-region can be replaced by several MSCs, each showing a different order of signal exchange.

B.2.2 MSC sub message sequence charts

A sub message sequence chart (ITU-T Recommendation Z.120 [3], §§ 6.10) gives a more detailed view of one instance of another MSC. This introduces a consistency problem: the sub MSC needs to be consistent with the more abstract MSC.

If MSCs are used in combination with SDL, the sub MSC is not needed, because it is possible to replace the sub MSC by a normal MSC, showing exactly the same behaviour. The consistency is then implicitly guaranteed because the more abstract MSC and the more detailed MSC will both be consistent with the SDL diagrams.

B.3 Motivation for exclusion of ASN.1 concepts in combination with SDL

This clause contains a rationale why some ASN.1 concepts are not allowed for use in combination with SDL. Some concepts have not been selected because they have a harmful effect on the testability of the specification, or complicate validation. Some others are not included in SDL (for example the macro), because they are too difficult to handle by tools.

B.3.1 ASN.1 comment

ASN.1 comments start with "--", and end at the end of line, or the next "--" . Use of ASN.1 comment would introduce difficulties in the definition of the SDL grammar, since line feeds have no meaning in SDL. The SDL comment symbol can be used instead.

B.3.2 Set

The ASN.1 SET construct is very similar to the SEQUENCE construct. The order in which the component values appear in a value notation of a SET can be in any order, whereas in a SEQUENCE value notation this order is fixed. SET types are mainly in ASN.1 for historic reasons. Their use leads to less efficient implementations. Instead of a SET type, a SEQUENCE type with exactly the same components can be used.

B.3.3 Macro mechanism

The ASN.1 macro mechanism is excluded from use. The reason is that macros are too complex to handle, and at ETSI they are mainly used to specify remote operations (see CCITT Recommendation X.229 [6]). These remote operations are, in fact, nothing but syntactic suggestion of specification of communication in ASN.1. A remote operation does not have any "real" semantics in ASN.1 (no communication is suggested by the ASN.1 semantics). Therefore, the SDL facilities to specify communication are preferred. In annex C, an example can be found on how to specify remote operations in SDL.

NOTE: If SDL is used as shown in the example, an operation code is not part of the specification. It needs to be specified separately in the encoding part of the standard.

B.3.4 Value notation for ASN.1 type ANY

The motivation for exclusion of values of type ANY is given by the principles of correct use of formalisms, and clarity the ASN.1 ANY construct is sometimes used to indicate that a parameter of a message is reserved for implementation specific purposes (allows implementation freedom). In that case, the specification should not specify specific values for that parameter. The other application of ASN.1 ANY is to indicate in a preliminary version of the specification that the exact type is not yet known. In the final version of an ETS, such use of ANY is not permitted.

This means that the only way to use ANY in combination with SDL is to receive parameters of type ANY in signals coming from the environment. These parameters can then be dropped, or forwarded to other processes. They cannot be modified.

B.3.5 ASN.1 encoding rules

There are no encoding rules specified for SDL data. Because the meaning of ASN.1 data is given in terms of SDL data (see annex D), there is no encoding imposed on ASN.1 in SDL diagrams either. It is possible to specify that certain encoding rules apply, for example the basic encoding rules for ASN.1 as defined in CCITT Recommendation X.209 [5]. However this is outside the scope of the SDL specification. Therefore, no assumptions on the specified ASN.1 encoding rules can be made in the SDL specification.

Annex C (informative): Examples

C.1 Addressing in SDL

An address in the sense of OSI is different from an address in the sense of SDL. In SDL, processes are addressed by a PId value. A process instance PId value is not known before the specification is interpreted, the PId value is assigned at the creation of a process instance. Hence, to model OSI addressing in SDL, a conversion of OSI addresses to PId values needs to be made.

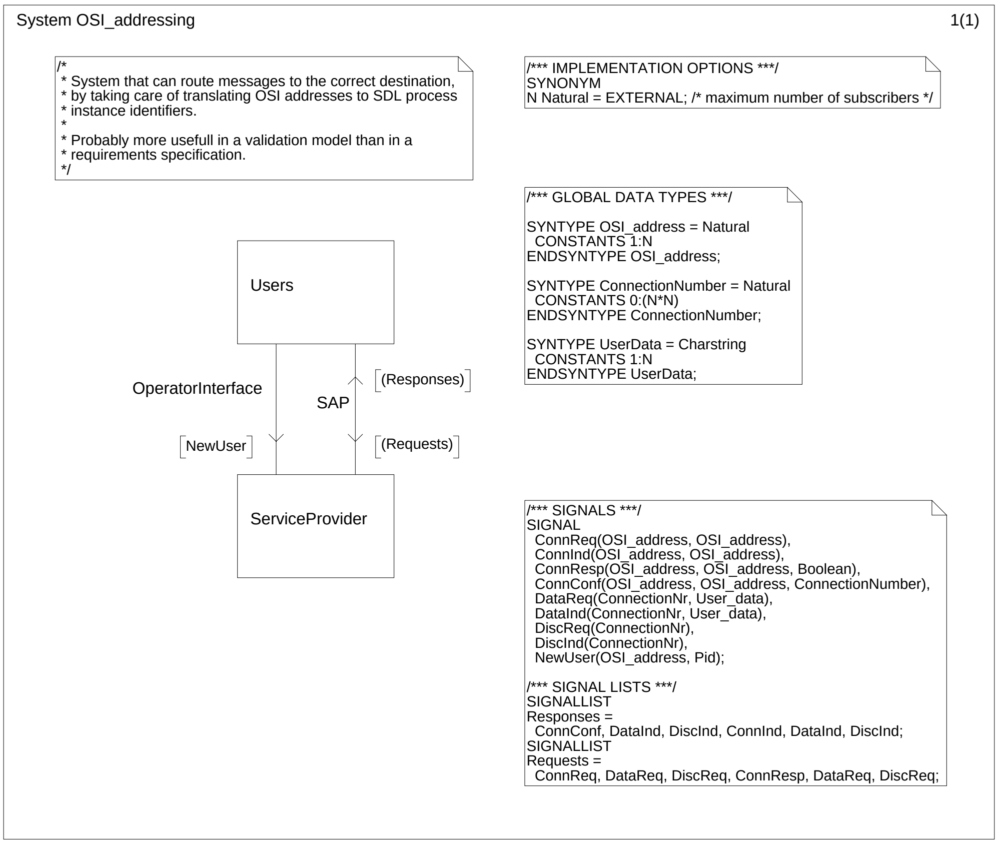
This example presents one approach to model addressing using SDL. Basis for the example is the enclosed SDL system OSI_Addressing. The system consists of two blocks: Users and ServiceProvider. The block Users models the users of the provided service, and the block ServiceProvider models the network offering the service.

NOTE: The example is not complete, only the main principles of the modelling of addresses are shown.

In the block network there is one instance of a process "Manager" that maintains a data structure containing a mapping OSI addresses -> PId values. This data structure is modelled as an array of PId values indexed with OSI addresses. Each time a new user is created in the environment of the network, the array is updated with the created user's PId value.

A connection is modelled by an instance of the process Connection. There is one instance of this process for each established connection. The instances are dynamically created by the Manager process upon receipt of a connect request from a user. The PId value of each such instance is noted by the Manager process, and every subsequent communication on the established connection is routed through the Manager process. Every service primitive received from a user is directed to the correct process instance by retrieving the PId value from the OSI address.

By using this approach, the information about the structure of the network (addresses of connections) is located within the network. The only address information residing at the user side is the OSI addresses of other users.



```
/*
 * System that can route messages to the correct destination,
 * by taking care of translating OSI addresses to SDL process
 * instance identifiers.
 *
 * Probably more usefull in a validation model than in a
 * requirements specification.
 */
```

```
/*** IMPLEMENTATION OPTIONS ***/
SYNONYM
N Natural = EXTERNAL; /* maximum number of subscribers */
```

```
/*** GLOBAL DATA TYPES ***/

SYNTYPE OSI_address = Natural
  CONSTANTS 1:N
ENDSYNTYPE OSI_address;

SYNTYPE ConnectionNumber = Natural
  CONSTANTS 0:(N*N)
ENDSYNTYPE ConnectionNumber;

SYNTYPE UserData = Charstring
  CONSTANTS 1:N
ENDSYNTYPE UserData;
```

```
/*** SIGNALS ***/
SIGNAL
  ConnReq(OSI_address, OSI_address),
  ConnInd(OSI_address, OSI_address),
  ConnResp(OSI_address, OSI_address, Boolean),
  ConnConf(OSI_address, OSI_address, ConnectionNumber),
  DataReq(ConnectionNr, User_data),
  DataInd(ConnectionNr, User_data),
  DiscReq(ConnectionNr),
  DiscInd(ConnectionNr),
  NewUser(OSI_address, Pid);

/*** SIGNAL LISTS ***/
SIGNALLIST
Responses =
  ConnConf, DataInd, DiscInd, ConnInd, DataInd, DiscInd;
SIGNALLIST
Requests =
  ConnReq, DataReq, DiscReq, ConnResp, DataReq, DiscReq;
```

Figure C.1: Addressing example, system OSI_addressing

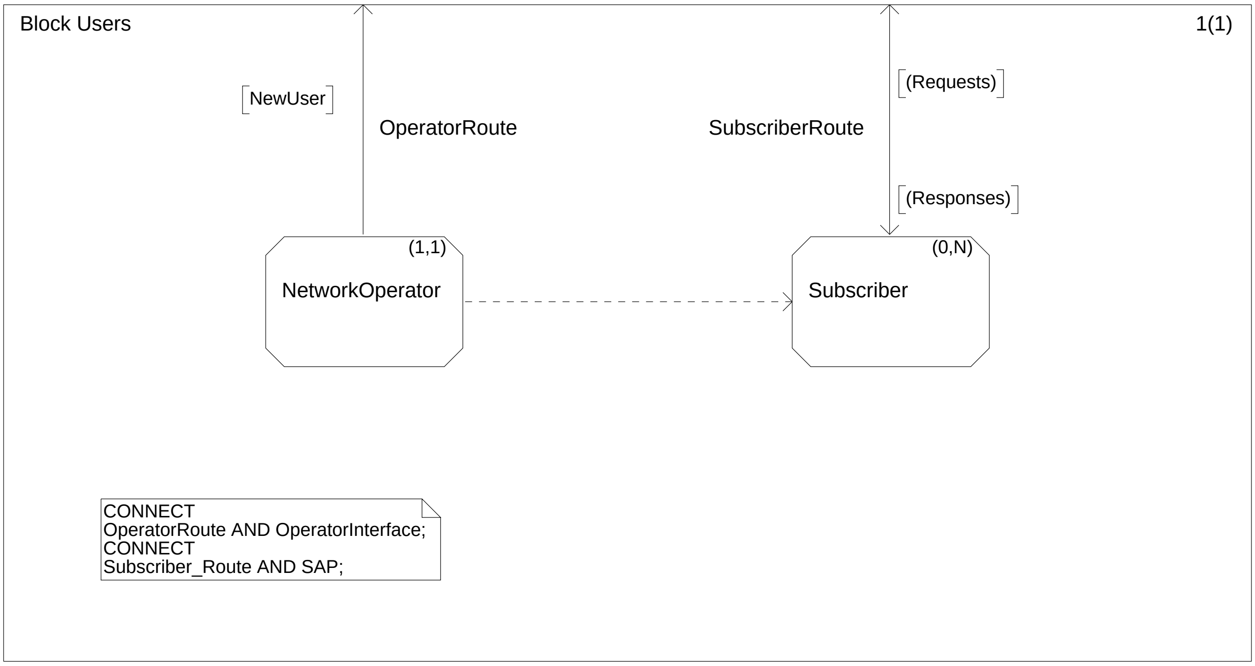


Figure C.2: Addressing example, block Users

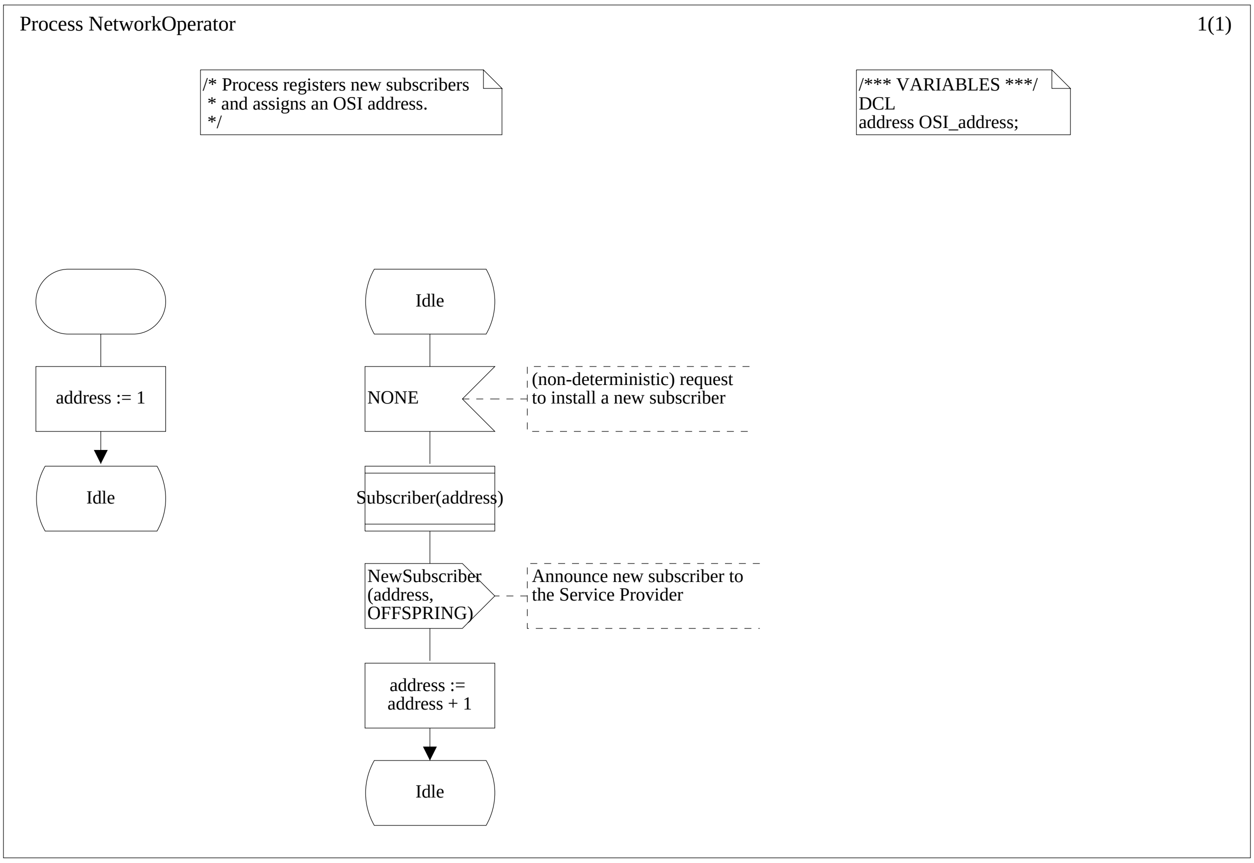


Figure C.3: Addressing example, process NetworkOperator

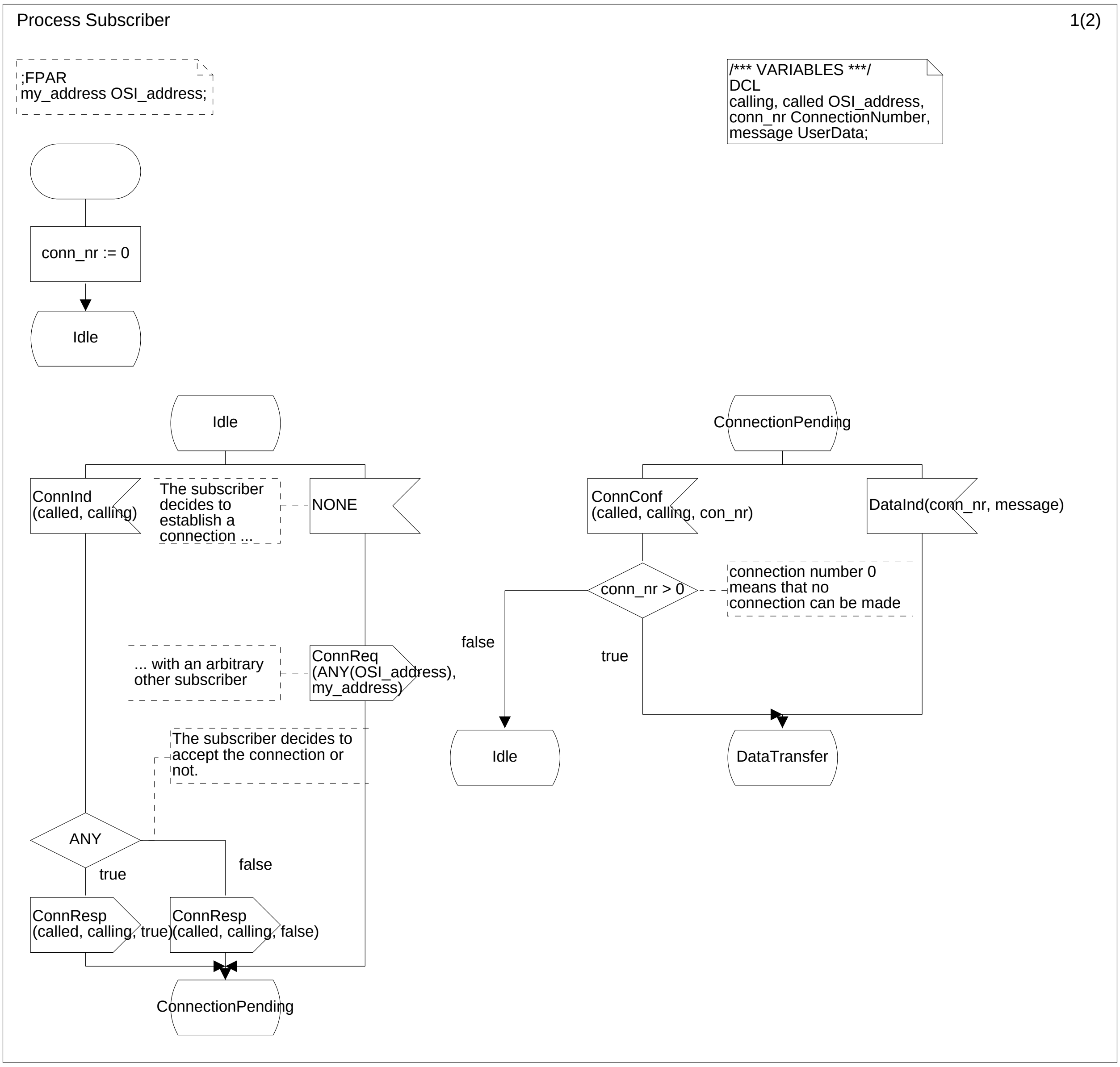


Figure C.4 (sheet 1 of 2): Addressing example, process Subscriber

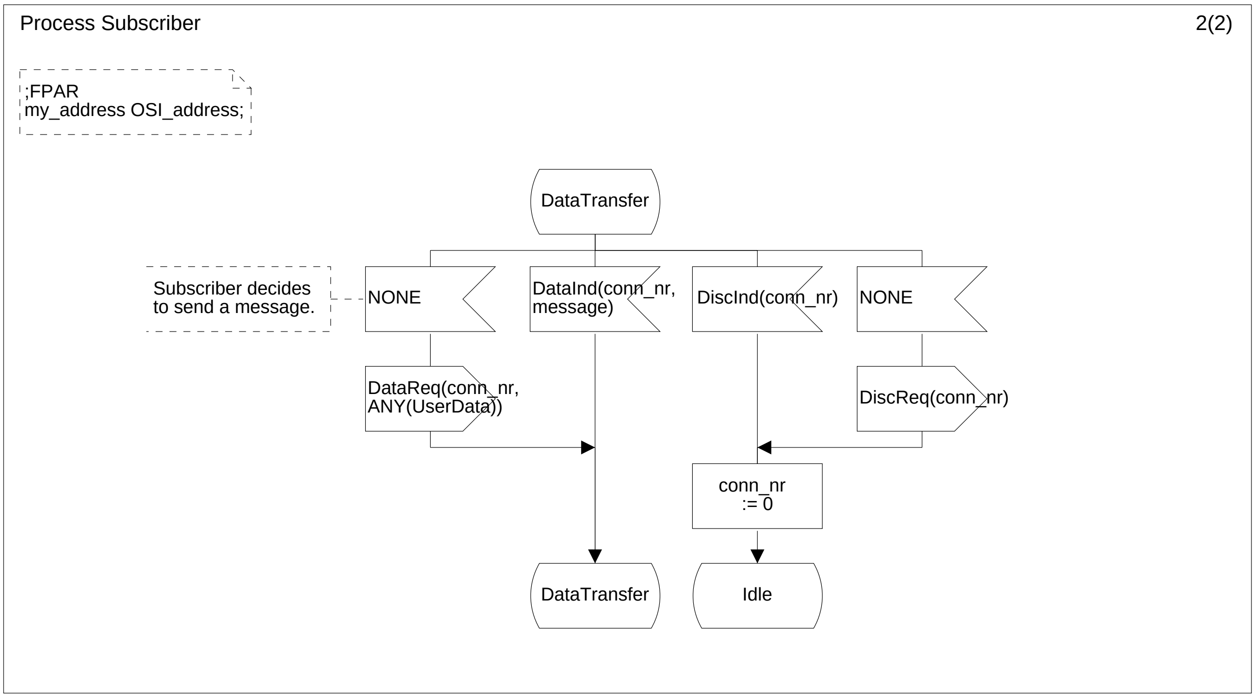


Figure C.4 (sheet 2 of 2): Addressing example, process Subscriber

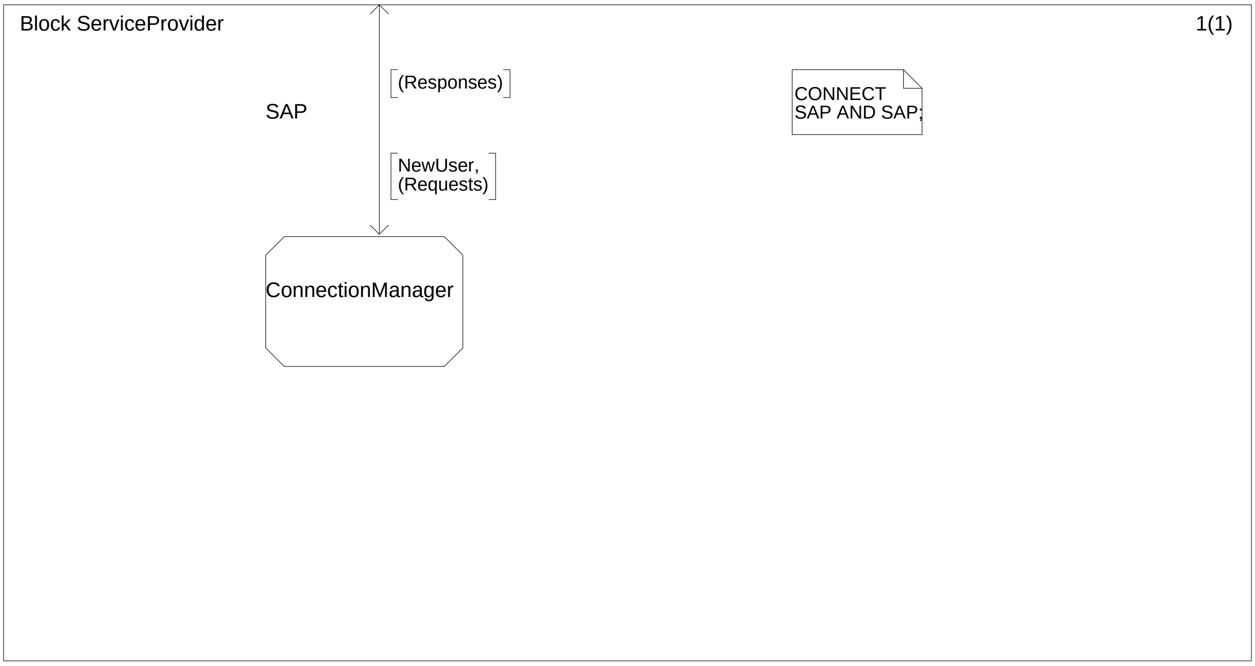
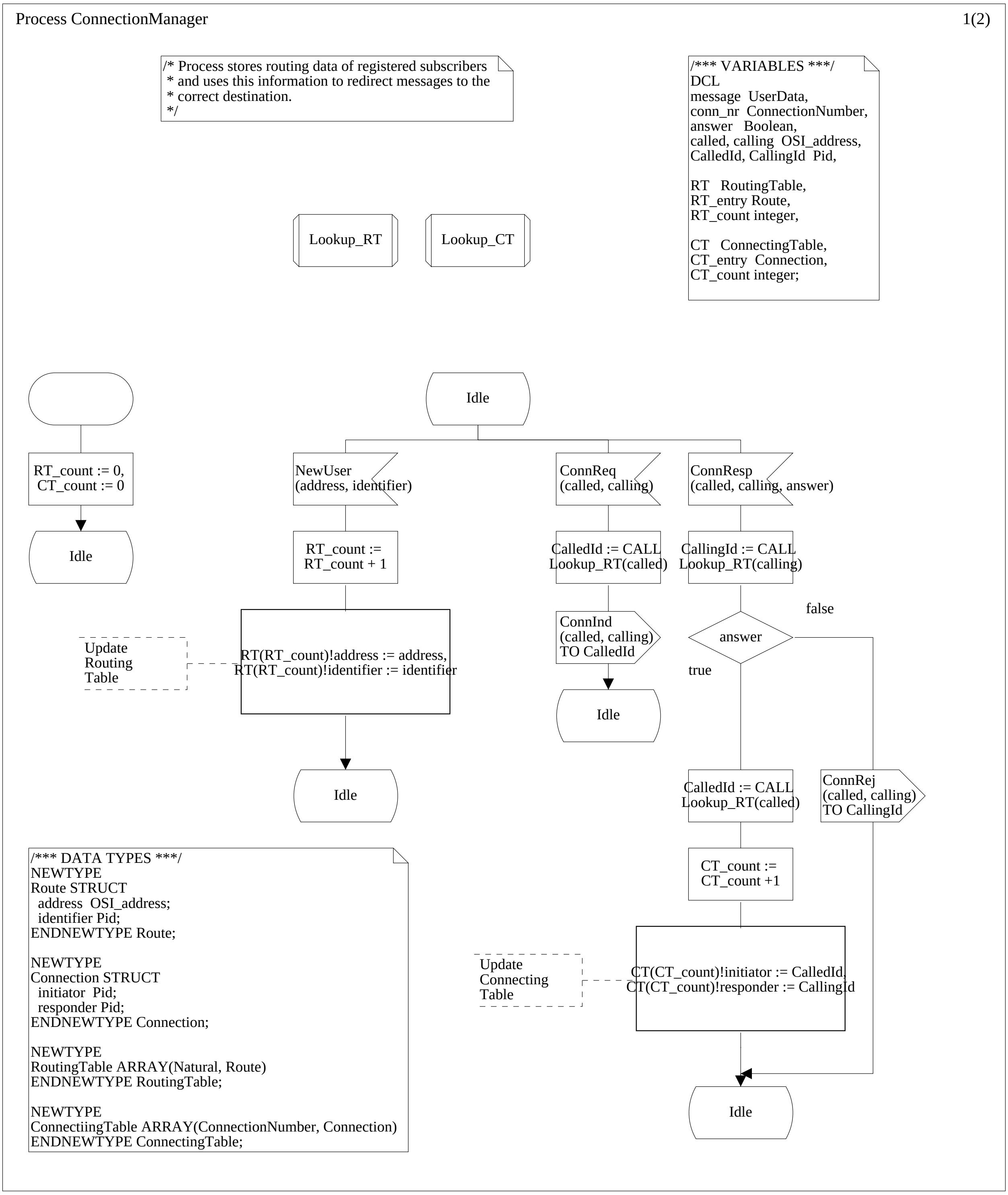


Figure C.5: Addressing example, block ServiceProvider



```
/* Process stores routing data of registered subscribers
 * and uses this information to redirect messages to the
 * correct destination.
 */
```

```
/*** VARIABLES ***/
DCL
message  UserData,
conn_nr  ConnectionNumber,
answer   Boolean,
called, calling  OSI_address,
CalledId, CallingId  Pid,

RT   RoutingTable,
RT_entry Route,
RT_count integer,

CT   ConnectingTable,
CT_entry  Connection,
CT_count integer;
```

```
/*** DATA TYPES ***/
NEWTYPE
Route STRUCT
  address  OSI_address;
  identifier Pid;
ENDNEWTYPE Route;

NEWTYPE
Connection STRUCT
  initiator  Pid;
  responder Pid;
ENDNEWTYPE Connection;

NEWTYPE
RoutingTable ARRAY(Natural, Route)
ENDNEWTYPE RoutingTable;

NEWTYPE
ConnectiingTable ARRAY(ConnectionNumber, Connection)
ENDNEWTYPE ConnectingTable;
```

Figure C.6 (sheet 1 of 2): Addressing example, process ConnectionManager

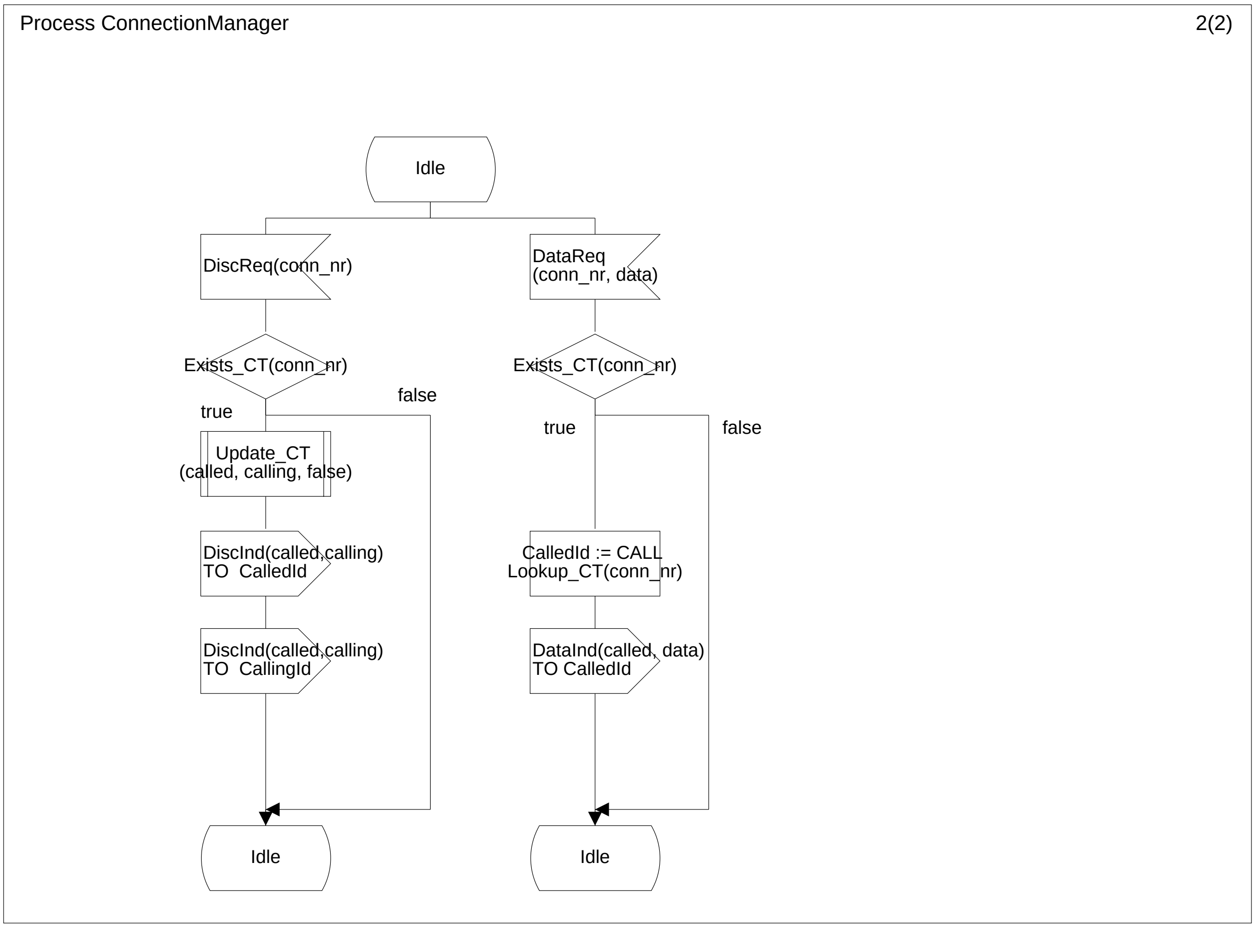


Figure C.6 (sheet 2 of 2): Addressing example, process ConnectionManager

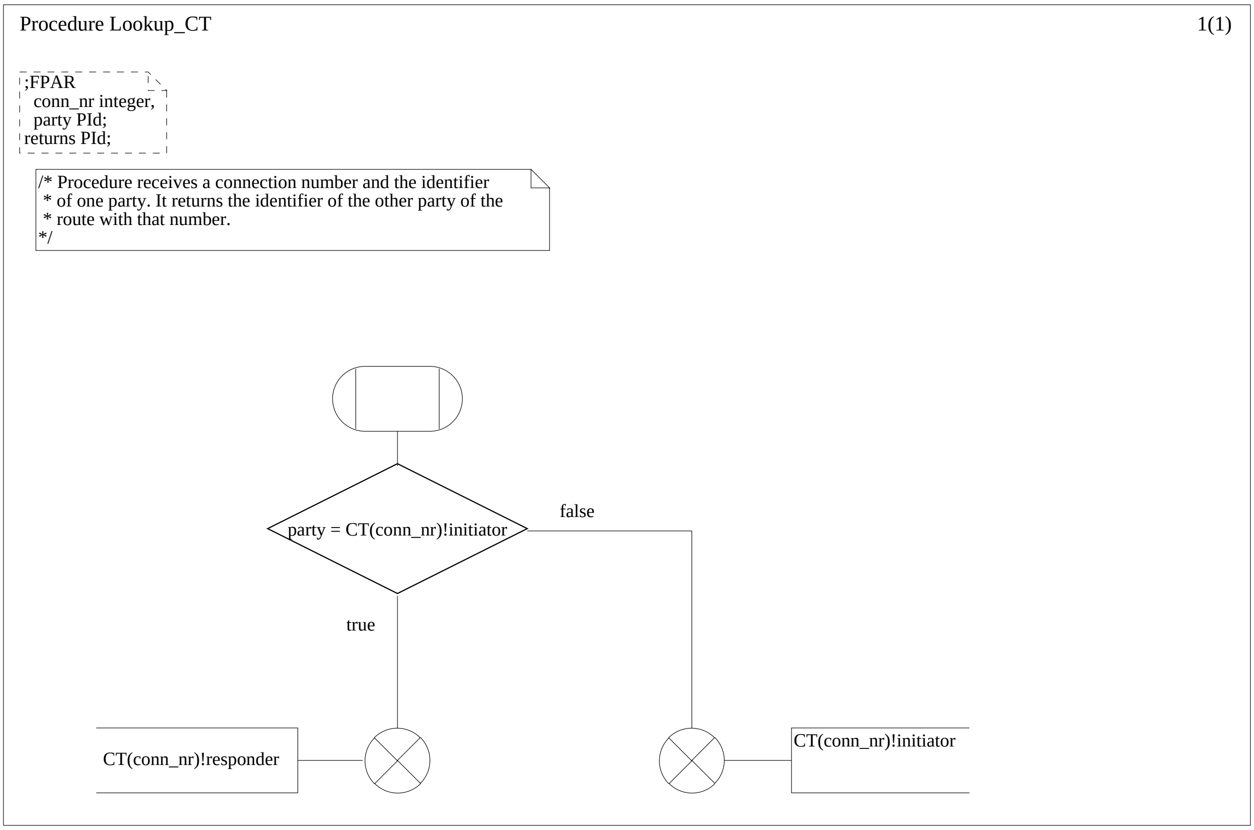

Procedure Lookup_CT
1(1)
;FPAR
conn_nr integer,
party PId;
returns PId;
/* Procedure receives a connection number and the identifier
* of one party. It returns the identifier of the other party of the
* route with that number.
*/
party = CT(conn_nr)!initiator
false
true
CT(conn_nr)!responder
CT(conn_nr)!initiator

Figure C.7: Addressing example, procedure Lookup_CT

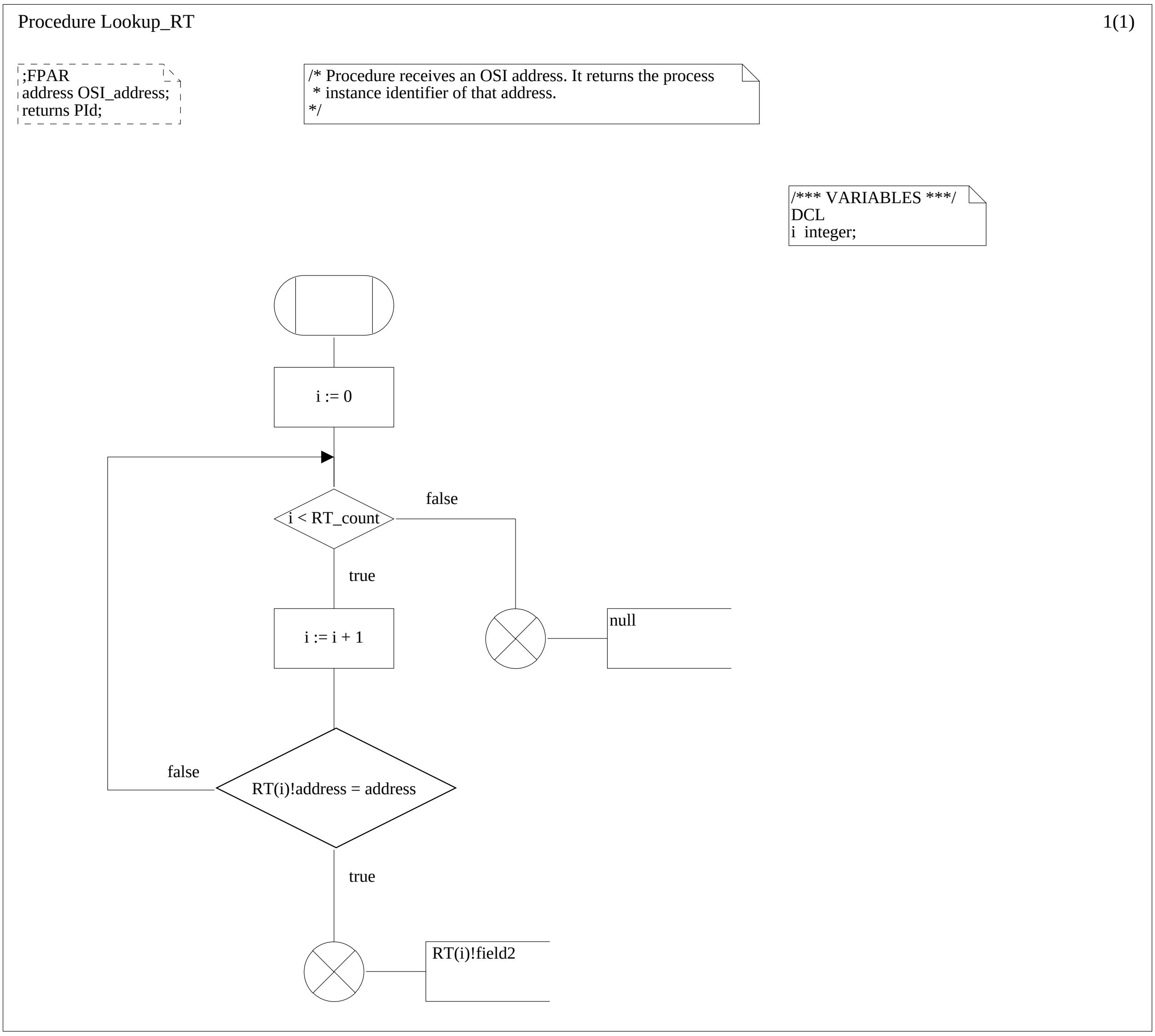


Figure C.8: Addressing example, procedure Lookup_RT

C.2 Remote operations

The ASN.1 operation macro is not allowed in the selected subset of ASN.1. The operation macro is heavily used in higher layer application protocols, for example in protocols built on Remote Operations Service Element (ROSE) or Transaction Capability Application Part (TCAP). This example shows what can be done instead, using SDL. The idea is that a remote operation is specified in SDL using several signals:

- one for the invocation of the operation;
- one for returning the result of the operation (if applicable);
- one for returning an error in case the operation fails (if applicable);
- one for the rejection of an operation (if applicable).

This example shows a way how to model in SDL the part of the signalling protocol needed for the activation of the call forwarding supplementary service. It is inspired by ECMA-QSIG-CF (see annex E). The remote operation "ActivateDiversion" is used to activate the call forwarding. This operation is defined in the following ASN.1 module:

```
Call-Forwarding-Operations
     -- Example of a module that defines activation operation.
DEFINITIONS ::=
BEGIN
EXPORTS        ActivateDiversion,
               ActivateDiversionInvArg,
              ActivateDiversionResArg,
              ActivateDiversionErrors;

ActivateDiversion  OPERATION
     ARGUMENT      ActivateDiversionInvArg
     RESULT        ActivateDiversionResArg
     ERRORS        ActivateDiversionErrors

ActivateDiversionInvArg ::= SEQUENCE {
     procedure          ENUMERATED { unconditional (0), on_busy (1),
                                     no_reply (2)},
     divertedToAddress  Address,
     servedUserNr       PartyNumber,
     activatingUserNr   PartyNumber }

ActivateDiversionResArg ::= NULL

ActivateDiversionErrors ::= ENUMERATED {
              notSubscribed (0), invalidServedUserNr (1),
              invalidDivertedNr (2), notAuthorized (3), unspecified (4) }

END
```

The call forwarding is activated with the protocol given below. The exchange serving the user that activates the call forwarding service is abbreviated as Activating X. The exchange serving the served user (= the user whose incoming phone calls are diverted) is abbreviated as Served User X. The setting up and clearing down of signalling connections is for simplicity reasons not taken into account.

C.2.1 Actions at the Activating X

On the receipt of an activation request from the user, the Activating X shall send an activateDiversionQ invoke Application Protocol Data Unit (APDU) to the Served User X. The Activating X starts timer T1. On receipt of the activateDiversion return result APDU, the Activating X shall convey the return result back to the activating user. On receipt of the activateDiversion return error or reject APDU from the served user, or on expiration of timer T1, rejection shall be indicated to the activating user.

C.2.2 Actions at the Served User X

On receipt of an activateDiversionQ invoke APDU, the Served User X shall verify that remote activation is supported and enabled. The Served User X may use any techniques for verifying, as far as possible, that the diverted-to user's number is valid.

If the activation request is acceptable, the Served User X shall activate diversion of the type indicated by the element procedure, answer with a return result APDU, store the received diverted-to number, and optionally convey an appropriate notification to the served user. If the diverted-to user's number is detected as an invalid number, or if the activation request can not be accepted for other reasons, the Served User X shall send back a return error APDU with an appropriate value.

C.2.3 Timer values

The timer T1 shall have a value of 5 seconds.

C.2.4 Message sequence chart

The following MSC shows the working of the protocol for the successful activation of call forwarding.

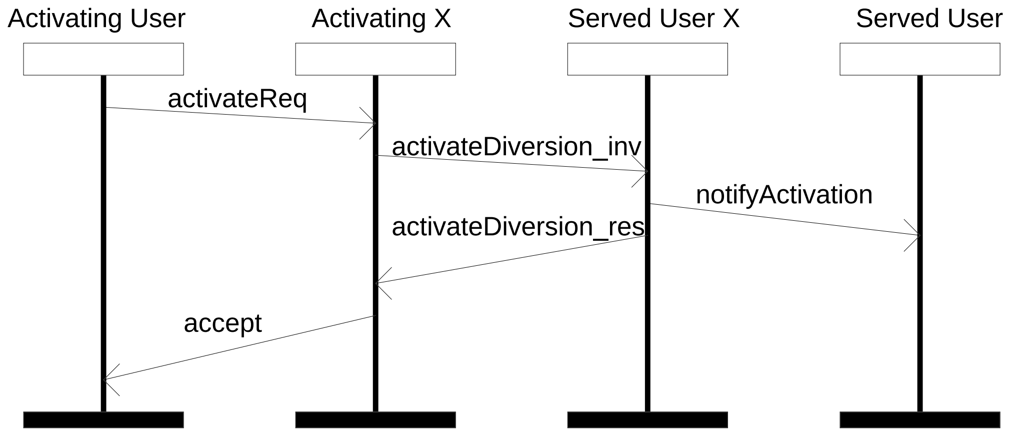


Figure C.9: MSC for the successful activation of call forwarding

C.2.5 SDL diagrams

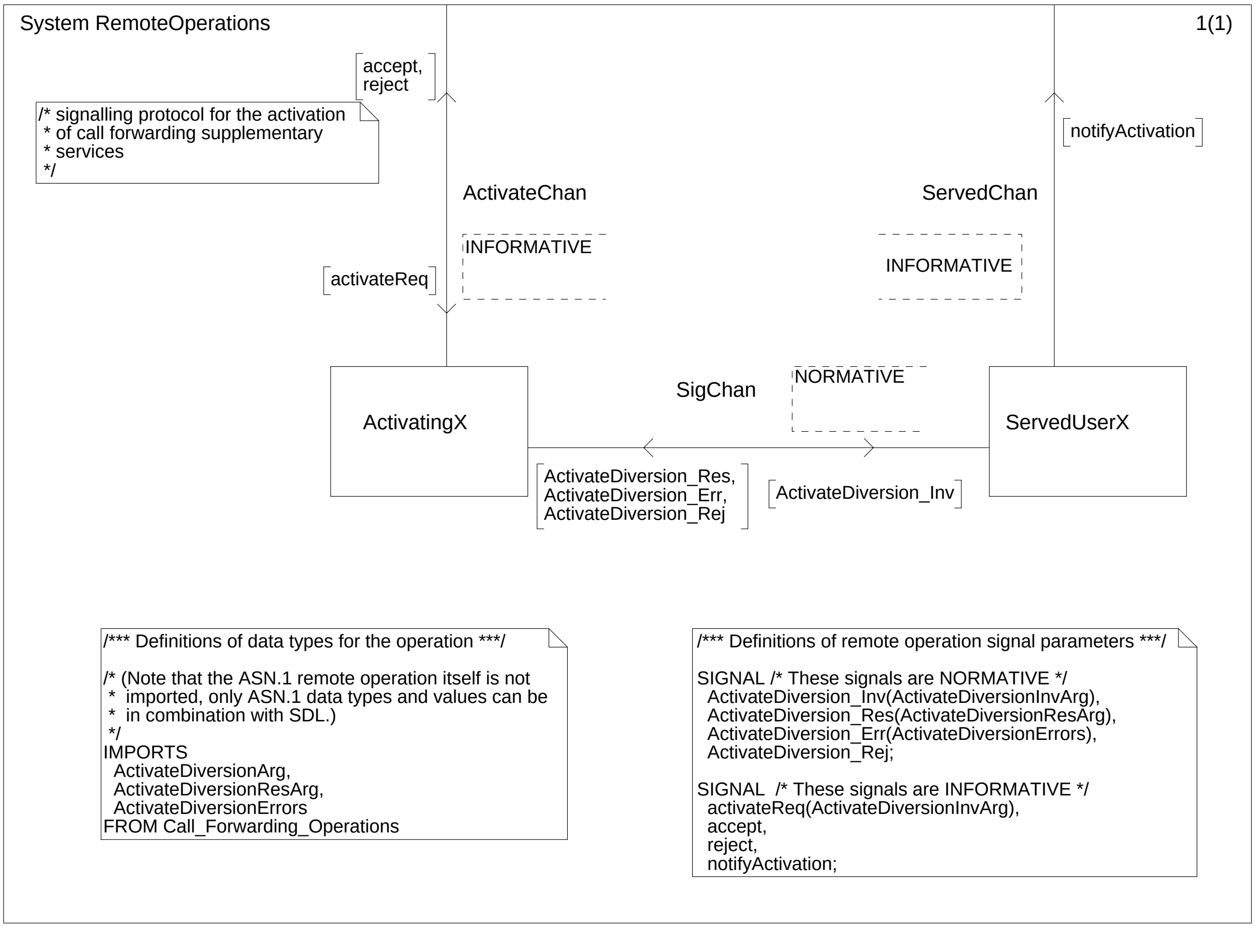
For the specification of the protocol in SDL, it is impossible to use the ASN.1 module as it is, because there is a remote operation in it. But the module can be easily changed, so that only the argument, result and errors are defined. These types will be imported in the SDL part. The changed ASN.1 module is given below.

```
Call-Forwarding-Operations
     -- Example of a module that defines activation operation.
     -- (made suitable for inclusion in SDL by removing the operation itself,
     -- only keeping the argument, result and error types)
DEFINITIONS ::=
BEGIN
EXPORTS       ActivateDiversionInvArg,
              ActivateDiversionResArg,
              ActivateDiversionErrors;

ActivateDiversionInvArg ::= SEQUENCE {
     procedure          ENUMERATED { unconditional (0), on_busy (1),
                                     no_reply (2)},
     divertedToAddress  Address,
     servedUserNr       PartyNumber,
     activatingUserNr   PartyNumber }

ActivateDiversionResArg ::= NULL

ActivateDiversionErrors ::= ENUMERATED {
              notSubscribed (0), invalidServedUserNr (1),
              invalidDivertedNr (2), notAuthorized (3), unspecified (4) }

END
```

The following SDL diagrams show the detailed protocol. The remote operation ActivateDiversion is mapped on four SDL signals: one for the invocation, going from Activated X to Served User X, and three for the possible responses (result, error, or reject), going from Served User X to Activated X.

The ASN.1 definitions of the remote operation argument, result and errors are imported at the system diagram using the IMPORTS construct (see section ..). Note that the operation ActivateDiversion itself is not imported: remote operations shall not be used in SDL.

The procedures that are called in the process ServedUserProc are not presented to limit the space.

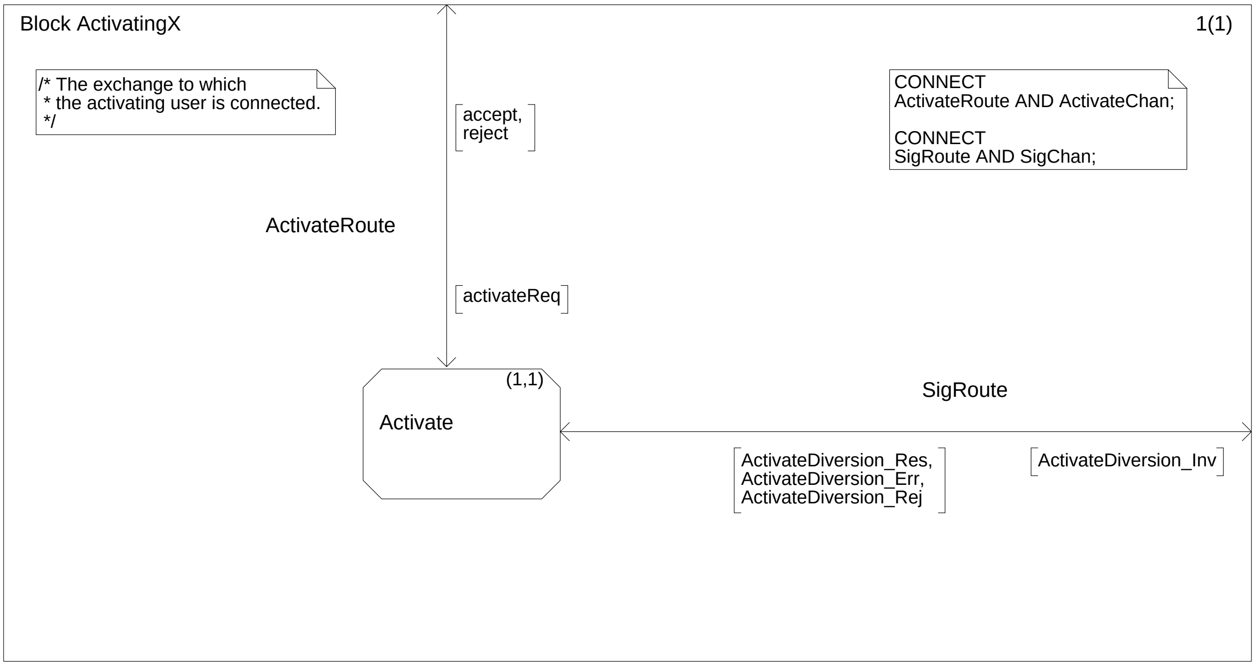


```
/* signalling protocol for the activation
 * of call forwarding supplementary
 * services
 */
```

```
/*** Definitions of data types for the operation ***/

/* (Note that the ASN.1 remote operation itself is not
 *  imported, only ASN.1 data types and values can be
 *  in combination with SDL.)
 */
IMPORTS
  ActivateDiversionArg,
  ActivateDiversionResArg,
  ActivateDiversionErrors
FROM Call_Forwarding_Operations
```

```
/*** Definitions of remote operation signal parameters ***/

SIGNAL /* These signals are NORMATIVE */
  ActivateDiversion_Inv(ActivateDiversionInvArg),
  ActivateDiversion_Res(ActivateDiversionResArg),
  ActivateDiversion_Err(ActivateDiversionErrors),
  ActivateDiversion_Rej;

SIGNAL  /* These signals are INFORMATIVE */
  activateReq(ActivateDiversionInvArg),
  accept,
  reject,
  notifyActivation;
```

Figure C.10: Remote operations example, system RemoteOperations



```
/* The exchange to which
 * the activating user is connected.
 */
```

```
CONNECT
ActivateRoute AND ActivateChan;

CONNECT
SigRoute AND SigChan;
```

Figure C.11: Remote operations example, block ActivatingX

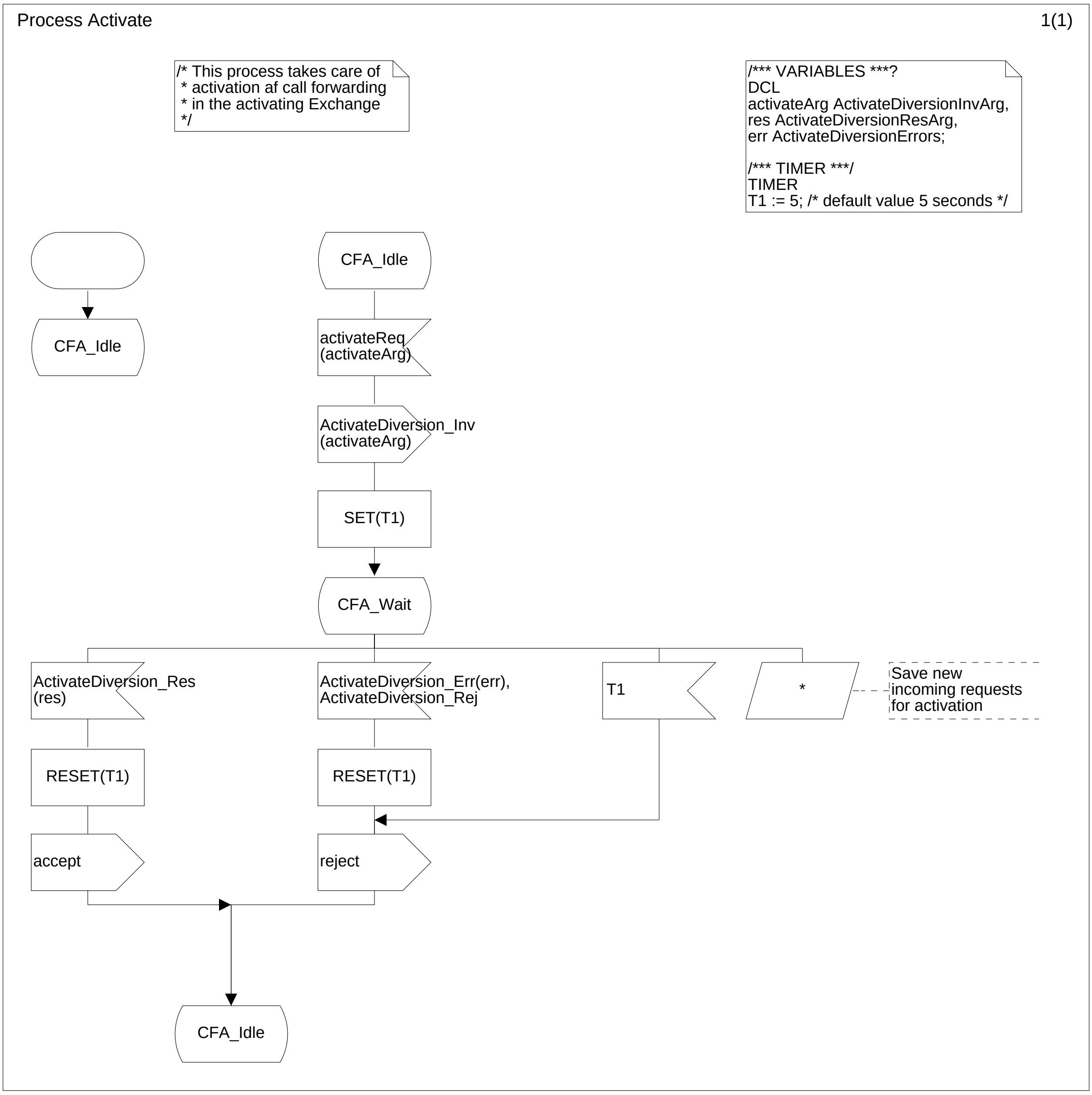


Figure C.12: Remote operations example, process Activate

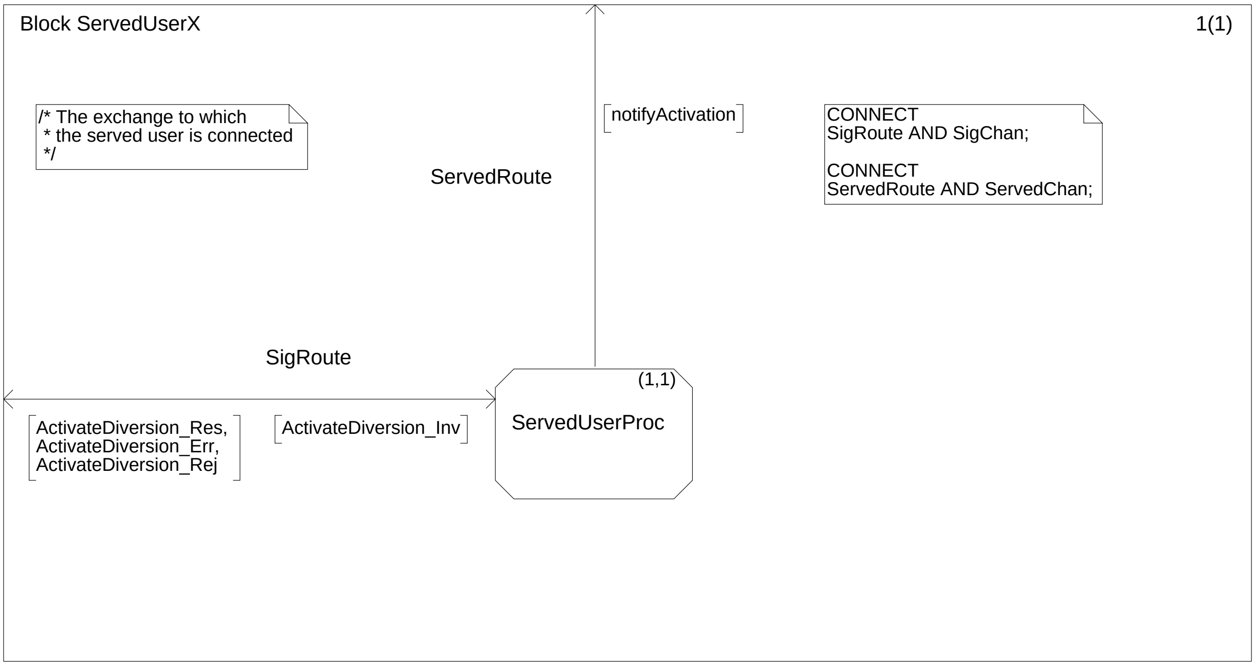


Figure C.13: Remote operations example, block ServedUserX

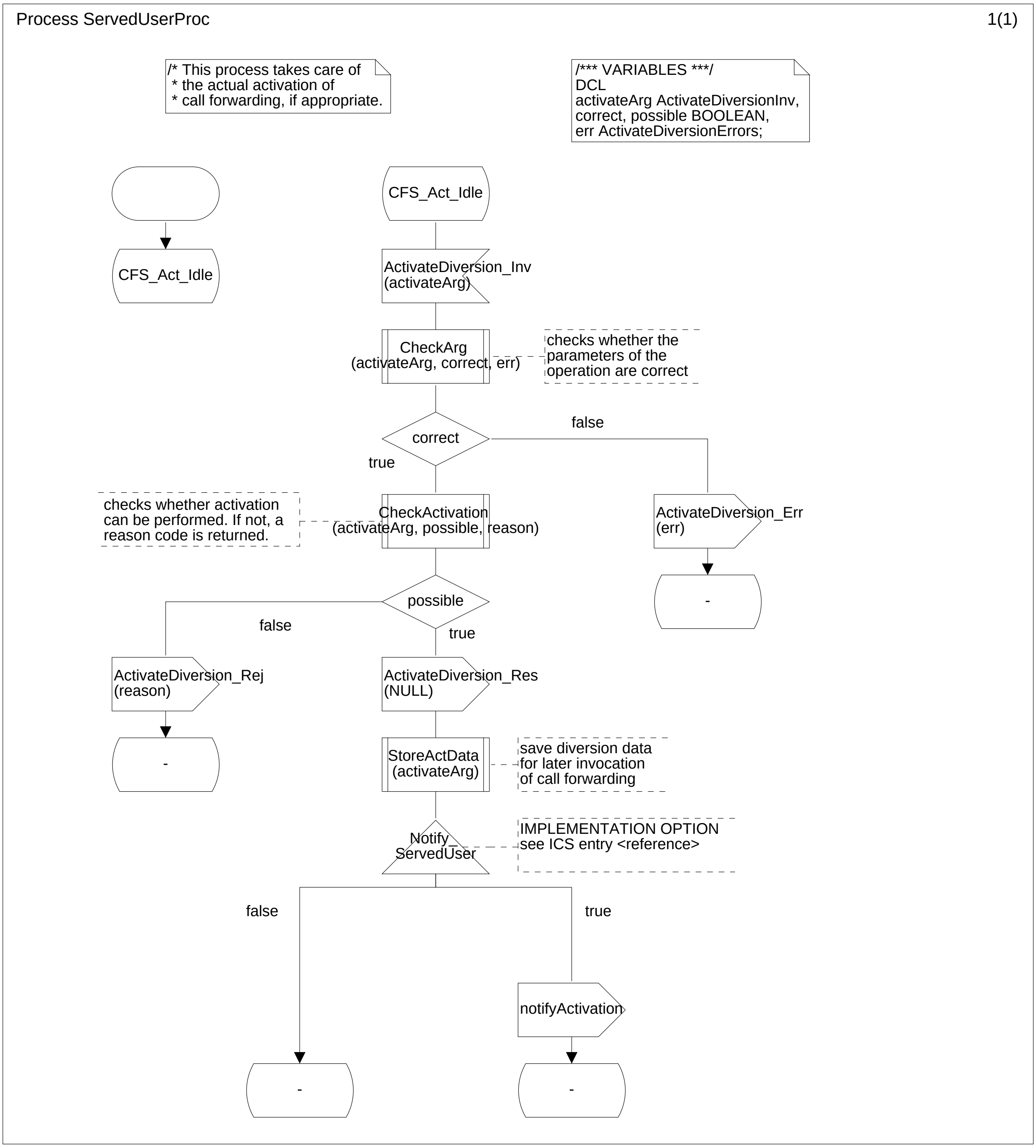

Process ServedUserProc
1(1)
/* This process takes care of
* the actual activation of
* call forwarding, if appropriate.
/*** VARIABLES ***/
DCL
activateArg ActivateDiversionInv,
correct, possible BOOLEAN,
err ActivateDiversionErrors;
CFS_Act_Idle
CFS_Act_Idle
ActivateDiversion_Inv
(activateArg)
CheckArg
(activateArg, correct, err)
checks whether the parameters of the operation are correct
correct
false
true
checks whether activation can be performed. If not, a reason code is returned.
CheckActivation
(activateArg, possible, reason)
ActivateDiversion_Err
(err)
-
possible
false
true
ActivateDiversion_Rej
(reason)
ActivateDiversion_Res
(NULL)
-
StoreActData
(activateArg)
save diversion data for later invocation of call forwarding
Notify_
ServedUser
IMPLEMENTATION OPTION
see ICS entry <reference>
false
true
notifyActivation
-
-

Figure C.14: Remote operations example, process ServedUserProc

C.3 Specification of implementation options

Two examples of implementation options (taken from the DECT standard, ETS 300 175 [11]) are:

a) an implementation of the DECT portable handset does not have to support the capability "incoming calls". In that case the mobile phone can only be used for initiating a telephone call, not for receiving calls;

b) when an unexpected message, other than CC_RELEASE or CC_RELEASE_COMPLETE is received, or an unrecognised message is received in any state, the message should be ignored. Alternatively, a release procedure may be initiated by sending a CC_RELEASE_COMPLETE message, indicating the release reason as "unexpected message".

If an SDL specification is parameterized with the possible choices between implementation options, it is possible to specify how the system behaviour depends on these choices. Furthermore it is possible to use information in an Implementation Conformance Statement, as actual parameters of the specification. This results in a specification that closely describes the system under test.

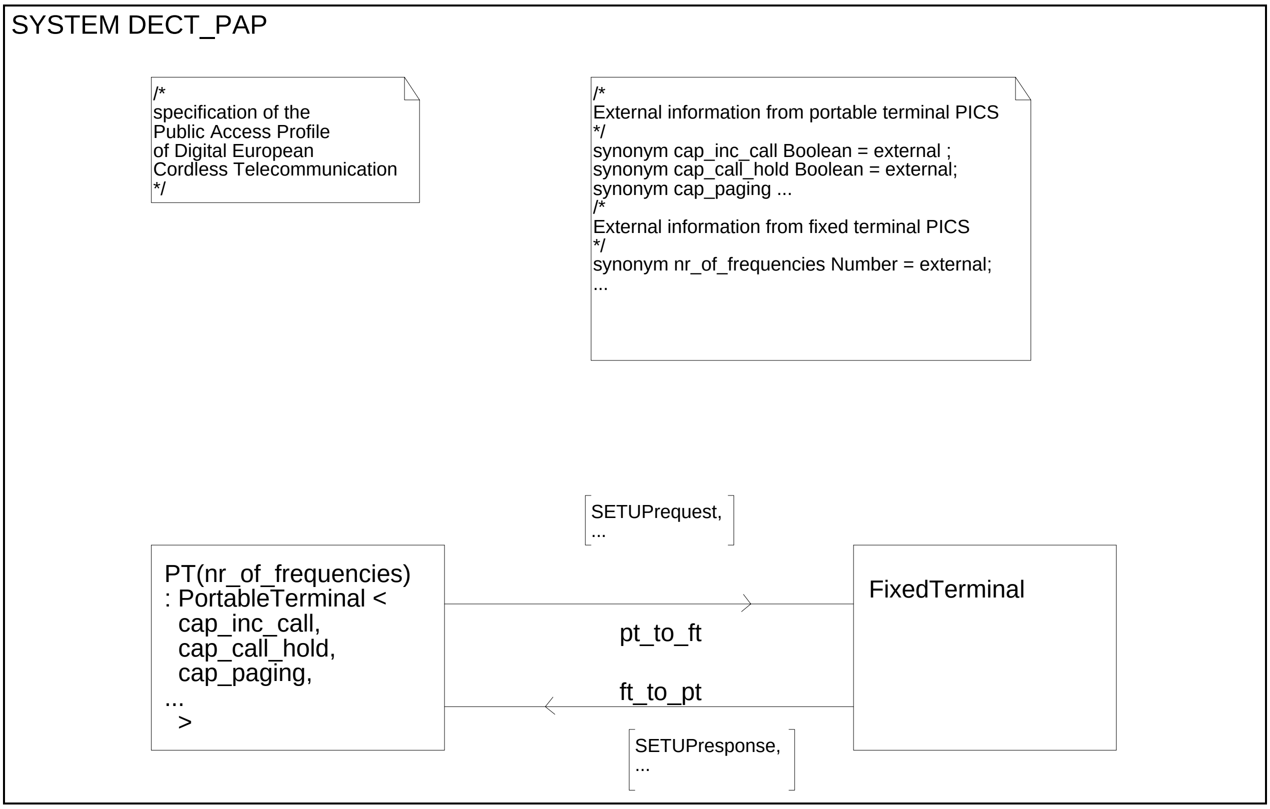


Figure C.15: System diagram with declaration of external synonyms

Figure C.15 shows a system diagram that uses external synonyms to model basic implementation options. The constant cap_inc_call has an unknown value, that has to be provided externally. Still the synonym can be used within the specification, as will be shown in the sequel.

C.4 Optional functionality

When a complete function of the system is optional to implement, it is possible to describe this function in a separate process and make the presence of this process dependant on the choice between implementation options.

Figure C.16 shows how the option a) can be specified in SDL. The dashed rectangle with a condition is named the SELECT symbol. This symbol indicates that the part of the specification within the dashed rectangle shall be considered to be non-existent if the condition evaluates to False. If the condition evaluates to True, the part of the specification within the dashed rectangle is valid, with all requirements that are imposed by it.

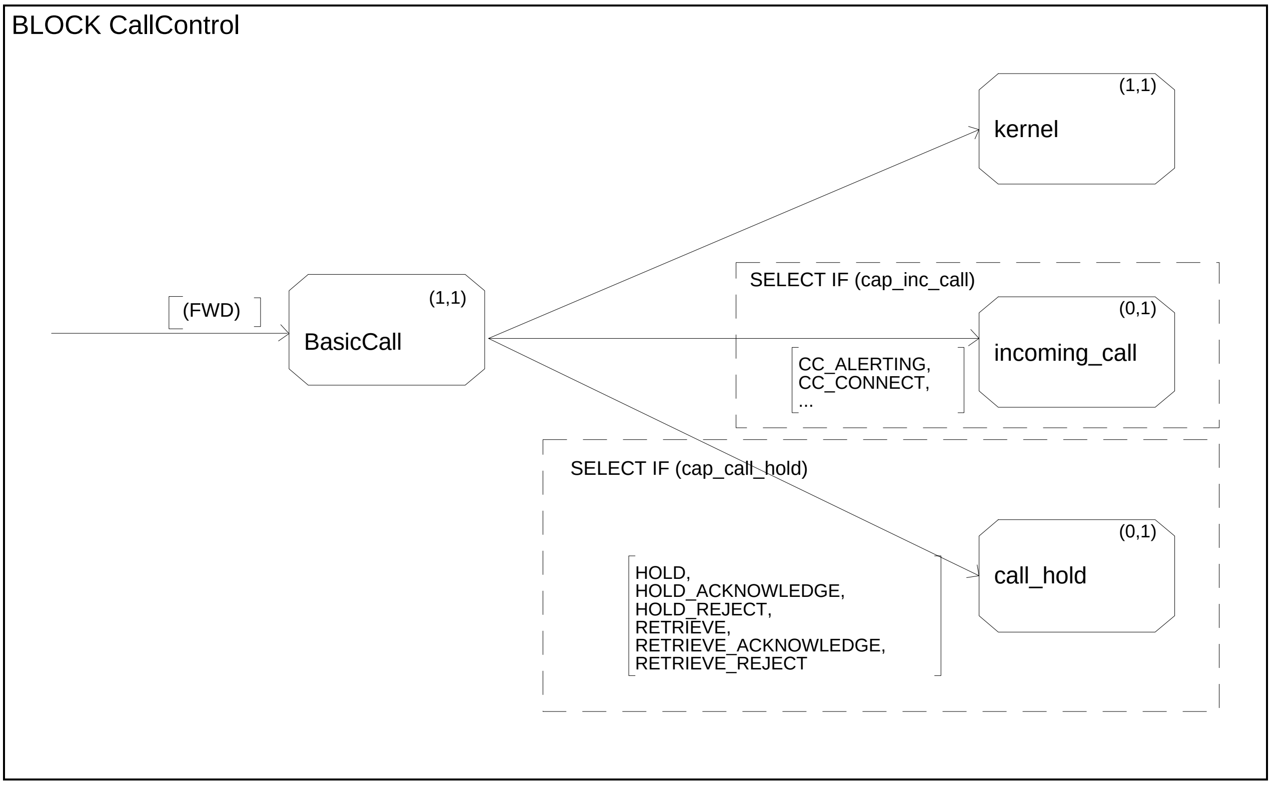


Figure C.16: Block diagram of DECT Call Control, with optional processes

C.5 Alternative behaviour

If the implementation option deals with alternative behaviour, the optional transition symbol can be used. Figure C.17 shows a part of a process diagram that uses the Optional transition symbol (the triangle). In this triangle the transition splits in two possible paths. An implementer has to choose which of the two possibilities will be supported in his product.

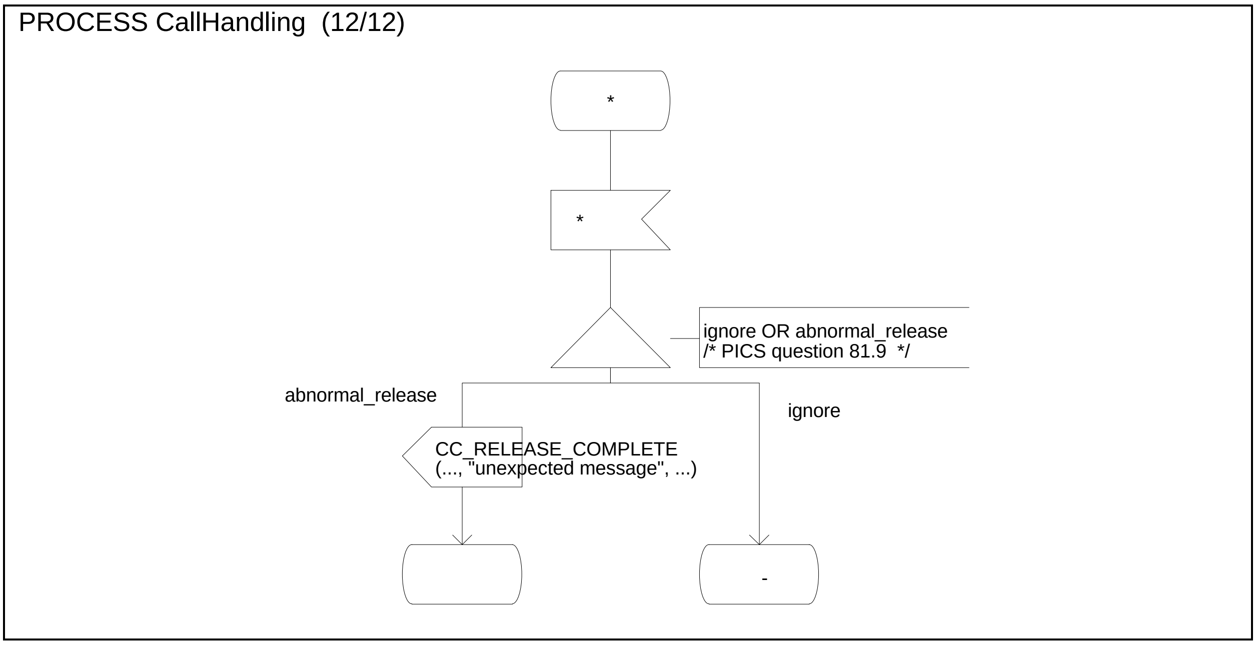

PROCESS CallHandling (12/12)
*
*
ignore OR abnormal_release
/* PICS question 81.9 */
abnormal_release
ignore
CC_RELEASE_COMPLETE
(..., "unexpected message", ...)
-

Figure C.17: Options for dealing with unexpected messages

C.6 Optional fields in a message

Cannot be dealt with in SDL. If ASN.1 is used for definition of data types, the optional fields can be indicated with the keyword OPTIONAL.

C.7 Shared data between processes

This example shows how small databases can be modelled in SDL.

SDL does not have the concepts of global variables: a variable is always local to a process, and can not be viewed or changed by another process. However, often it is necessary to have access to variables in other processes.

For example supplementary services in ISDN need to access information on subscribers, and their preferences regarding the provision of services. The Call Forwarding supplementary service, for example, needs to know whether the served user wants to be notified when a call is being forwarded.

Several processes may want to access this information. It is possible to specify this by introducing a data manager process. Other processes can get access to the information by means of signal exchange.

For example, on the following pages a block diagram is shown for call control. The processes BasicCallControl and SupplementaryServices both need access to subscriber data: they need to know which supplementary services are subscribed to. Process Management can update subscriber data: it can add new subscribers, or can add new supplementary services for a given subscriber.

These subscriber data are maintained by process DataManager. This process maintains information about the supplementary services that each subscriber subscribes to. A subscriber is represented in the data base as its address, which in this case is an ASN.1 OCTET STRING. The services that are subscribed to are represented by an ASN.1 BIT STRING. These data are defined in the following ASN.1 module:

```
SubscriberData
 {ccitt identified_organisation etsi(0) standards(0) 98 2} DEFINITIONS ::=
BEGIN
EXPORTS Address,
     Services,
     SubscrData;
Address ::= OCTET STRING (SIZE (1..20))

Services ::= BIT STRING {
     CallForwardUnconditional(0),
     CallForwardNoReply(1),
     CallWaiting(2),
     ThreeParty(3),
     CallingLinePresentation(4) }

SubscrData ::= SEQUENCE {
address Address,
subscrTo Services }
END -- of subscriber data definitions
```

By sending signals to DataManager, data can be updated and retrieved. As the block diagram shows, process Management can update the data base by sending signal AddAddress to add a subscriber, and by sending signal AddServices to add subscribed services for a given subscriber. e.g. "OUTPUT AddServices ('1234'H, {CallWaiting, ThreeParty})" would add Call Waiting and Three Party as subscribed services for subscriber with address '1234'H.

For simplicity reasons, signals to remove services and subscribers have been left out from the example.

Processes BasicCallControl and SupplementaryServices can retrieve which services a client subscribes. This can be done by sending signal SubscrInfReq with the clients address. DataManager will reply with SubscrInfResp, with the subscribed services as parameter.

The process diagram of Datamanager makes use of a value returning procedure FindAddress, that takes as parameter an address, and returns the index in the data base where this address occurs. This index can than be used to update or retrieve the supplementary services that are subscribed to.

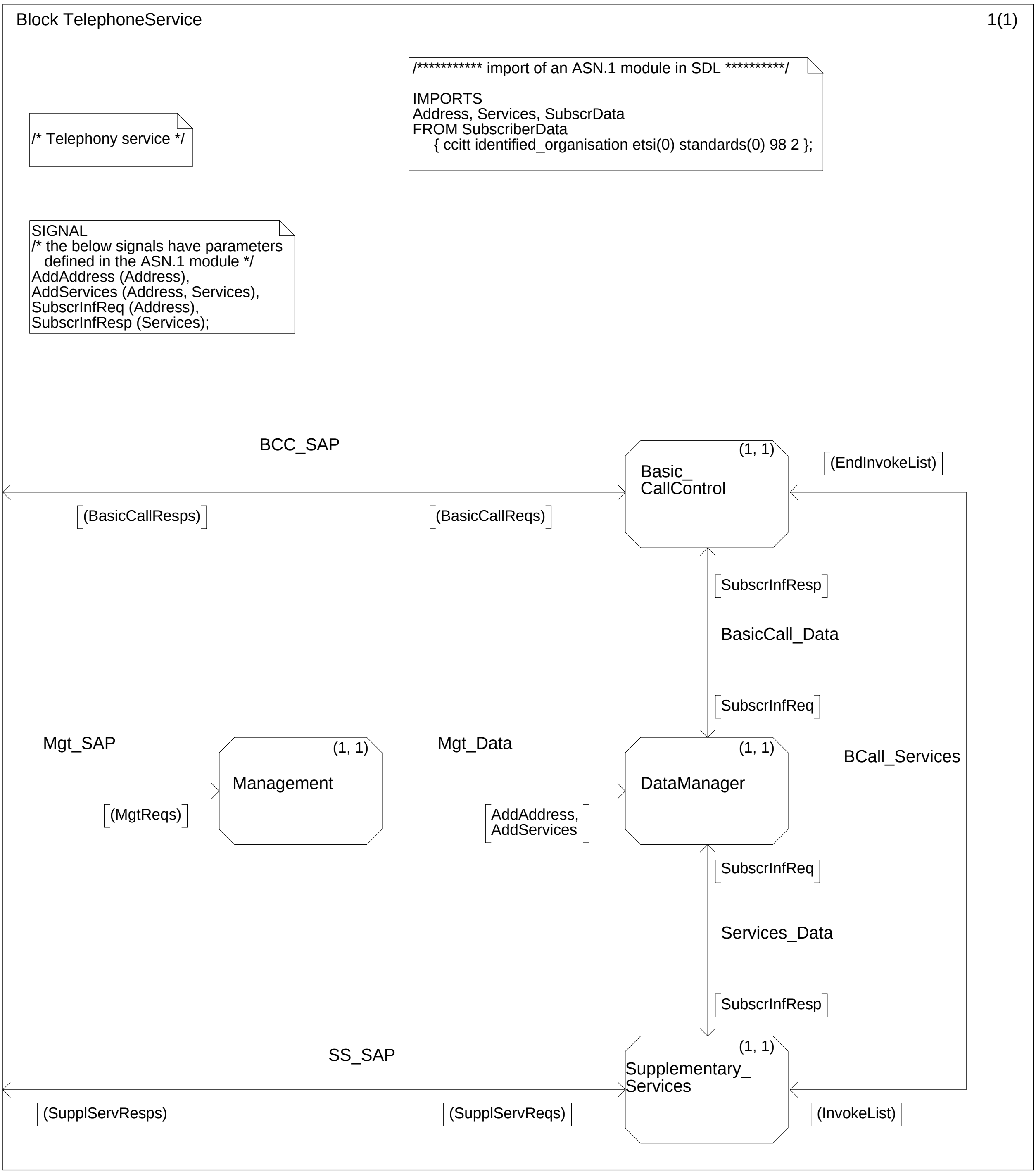


Figure C.18: Shared data example, block TelephoneService

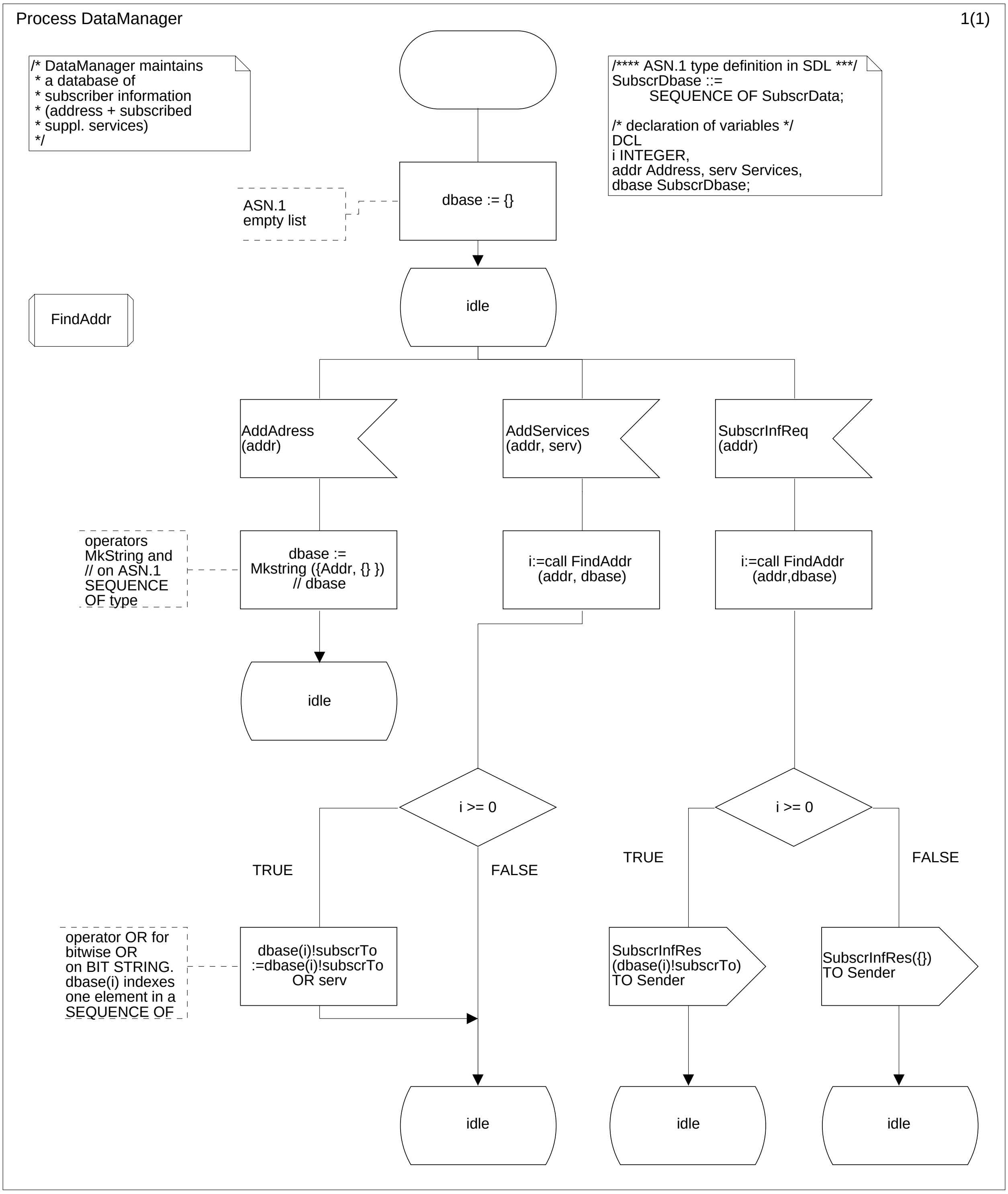


Figure C.19: Shared data example, process DataManager

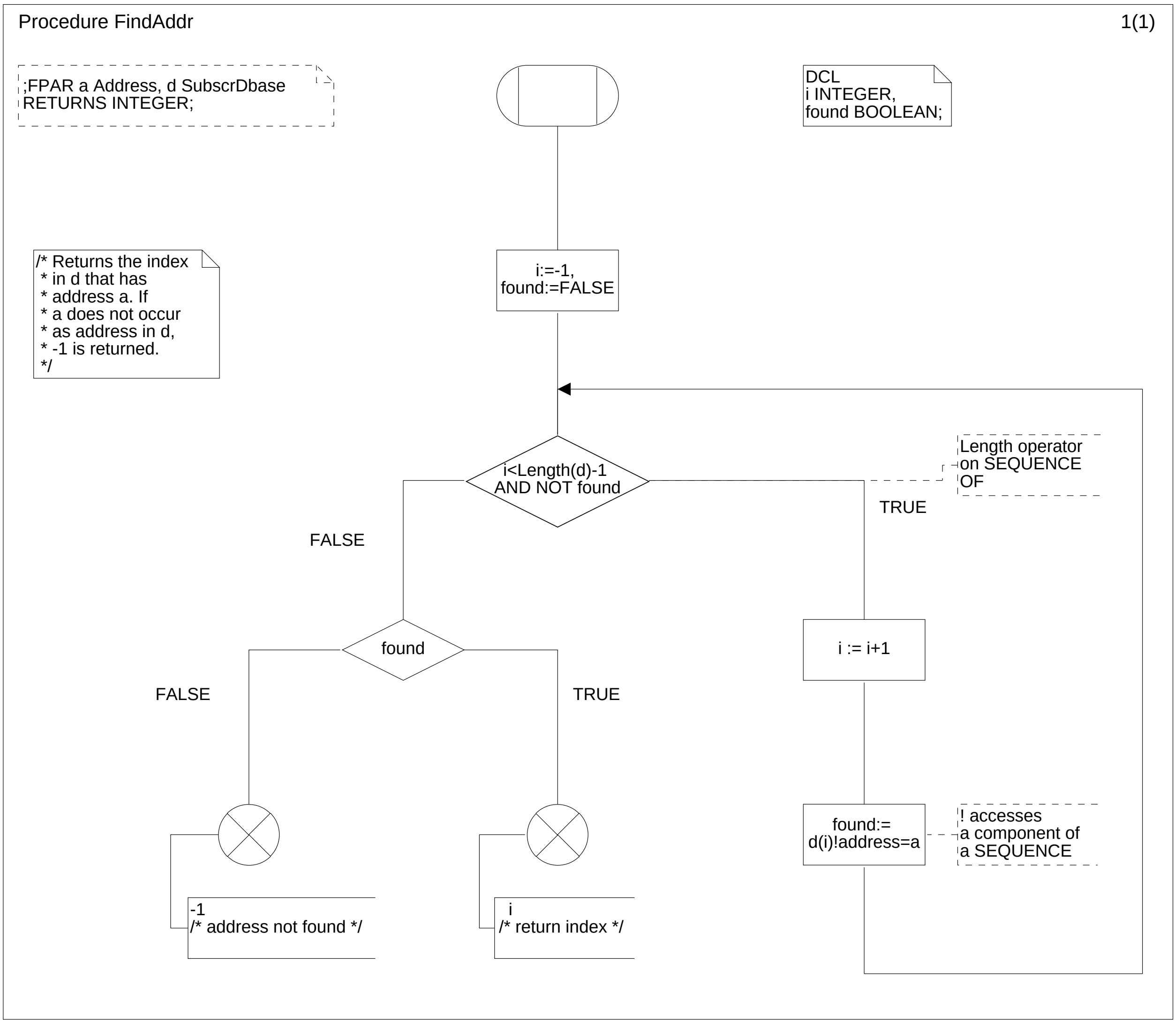


Figure C.20: Shared data example, procedure FindAddr

C.8 Informative parts of a specification

Informative parts may be present in a specification. For example, to model the environment of a system. In that case, it is desirable to model only those aspects of the environment that are relevant to the specified system, and abstract from the others.

SDL has several concepts to abstract from:

-	the precise cause of an event	NONE input;
-	the precise reason for a decision	ANY decision;
-	the precise assignment of a value	a := ANY Integer;
-	the data type of a signal parameter	ASN.1 type ANY;
-	the detailed composition of a signal	(..., parameter, ...).

Example models a packet switching network, that may loose packets, and does not guarantee that the order in which the packets are delivered is the same as the order in which they were submitted to the network.

Annex D (informative): Bibliography

The following references are given for informative purposes.

Tutorials

1) F. Belina, D. Hogrefe, A. Sarma (Prentice Hall, 1991): "SDL with applications from protocol specification".

2) R. Braek, O. Haugen, (Simon & Schuster international, 1993): "Engineering Real Time Systems".

3) O. Faergemand, A. Olsen (Forte conference, 1992): "New features in SDL 1992".

4) R. Saracco, J.R.W. Smith, R. Reed (North Holland, 1989): "Telecommunications systems engineering using SDL".

5) D. Steedman (Technology Appraisals Ltd., UK, 1990): "Abstract Syntax Notation One (ASN.1) - The Tutorial and Reference".

6) K.J. Turner (editor), (Wiley and sons, Sussex England 1993): "Using Formal Description Techniques (an introduction to LOTOS, Estelle and SDL)".

Background on specification, testing, and validation techniques

7) A. Ek, J. Ellsberger (Proceedings Fifth SDL Forum: Evolving Methods, North Holland, 1991): "A Dynamic Analysis Tool for SDL".

8) ETR 060 (1992): "Signalling Protocols and Switching (SPS); Guidelines for using Abstract Syntax Notation One (ASN.1) in telecommunication application protocols".

9) ETR 071 (1993): "Methods for Testing and Specification (MTS); Semantic relationship between SDL and TTCN. A common semantics representation".

10) ETSI92: ETSI 15th Technical Assembly, "TC ATM Reflections on standards validation", Temporary Document 29, July 1992.

11) J. Fischer, R. Schröder, "Combined Specification using SDL and ASN.1"

12) G.J. Holzmann, "Design and Validation of Computer Protocols", Prentice Hall, 1991.

13) Institution of Electrical Engineers (IEE guidelines, London 1988): "Guidelines assuring testability".

14) C. H. West (Computer networks and ISDN systems 24, p. 219-242, North Holland 1992): "Protocol validation - principles and applications".

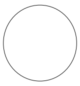
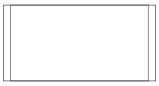
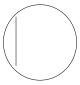
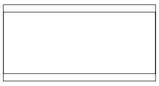
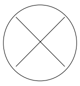
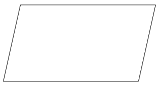
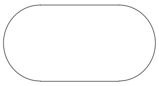
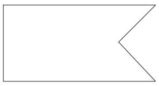
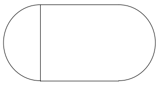
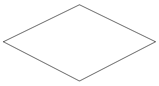
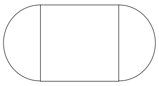
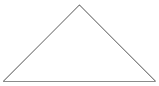
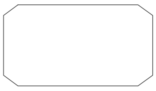
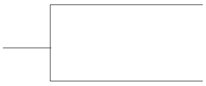
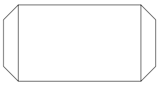
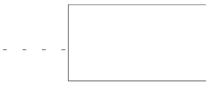
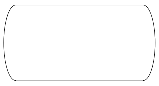
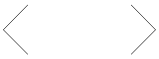
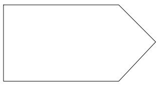
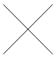
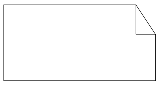
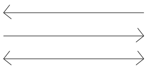
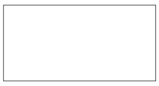
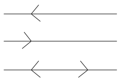
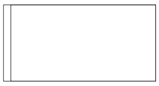
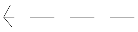
Other

15) ECMA-QSIG-CF (1992): "Private Telecommunication Networks - Inter-exchange signalling protocol - Diversion supplementary services".

Annex E (informative): Index

E.1 Textual index

E.2 Graphical index

symbol	name of the symbol + reference	symbol	name of the symbol + reference
	Join (layout mechanism to connect two lines) subclause 8.5.21		Procedure call subclause 8.5.16
	Macro outlet (end of macro) subclause 8.7.1		Process creation subclause 8.5.15
	Procedure return subclause 8.6.2		Save (prevents inputs from being thrown away) subclause 8.5.8
	Process start subclause 8.5.4		Input or timer expiry subclauses 8.5.6, 8.5.11
	Macro connection subclause 8.7.1		Decision subclause 8.5.14
	Procedure start subclause 8.6.1		Optional transition (option depending on PICS) subclause 8.5.12
	Process symbol subclause 8.4.1		Text extension subclause 8.3.7
	Procedure symbol subclause 8.3.9		Comment subclause 8.3.8
	State subclause 8.5.5		Continuous signal (transition triggered by a condition) subclause 8.5.10
	Output subclause 8.5.18		Process stop subclause 8.5.20
	Text symbol subclause 8.3.6		Channel without delay or signal route subclauses 8.3.2, 8.4.2
	Block, task symbol, timer set or timer reset subclauses 8.3.1, 8.5.13, 8.5.11		Channel with delay subclause 8.3.2
	Macro call subclause 8.3.10		Create line subclause 8.4.3

History

Document history			
May 1995	Public Enquiry	PE 62:	1994-05-09 to 1994-09-02
August 1995	Vote	86: extended:	1995-08-21 to 1995-10-13 1995-08-21 to 1995-10-27
December 1995	First Edition		
February 1996	Converted into Adobe Acrobat Portable Document Format (PDF)		

ISBN 2-7437-0405-5
Dépôt légal : Décembre 1995