TSG-RAN Working Group 2 (Radio layer 2 and Radio layer 3)
Stockholm 8th to 11th March 1999

TSGR2#2(99)082

Agenda Item: 6.1

Source: Nokia

Title: Descriptive SDL guidelines

Document for: Information

This contribution contains the descriptive SDL guidelines from ETSI MTS.

DEG/MTS-00050 V1.5 (Sep-98)

European Standard

Methods for Testing and Specification (MTS); Guidelines for the use of formal SDL as a descriptive tool

Reference
DEG/MTS-00050 (<Shortfilename>.PDF)

ETSI Secretariat

Postal address
F-06921 Sophia Antipolis CEDEX - FRANCE

Office address
650 Route des Lucioles - Sophia Antipolis
Valbonne - FRANCE
Tel.: +33 4 92 94 42 00 Fax: +33 4 96 65 47 16

X.400
c= fr; a=atlas; p=etsi; s=secretariat

Internet
secretariat@etsi.fr
http://www.etsi.fr

Contents

Foreword

This clause contains fixed text elements for the foreword.

1 Scope

This ETSI Guide (EG) establishes a set of guidelines for the formal use of Specification and Description Language (SDL) for descriptive, rather than detailed design, purposes. The objective of the guidelines is to provide assistance to rapporteurs of behaviour standards so that the SDL that appears in ETSI deliverables is formally expressed, easy to read and understand and at a level of detail consistent with other standards. This EG applies to all standards that make use of SDL to specify protocols, services or any other type of behaviour.

Users of this EG are assumed to have a working knowledge of SDL. The EG should not be considered to be an SDL tutorial and should be read in conjunction with ETS 300 414[1], ETR 298 [2] and EG 201 015 [3].

2 References

References may be made to:

a) specific versions of publications (identified by date of publication, edition number, version number, etc.), in which case, subsequent revisions to the referenced document do not apply; or

b) all versions up to and including the identified version (identified by "up to and including" before the version identity); or

c) all versions subsequent to and including the identified version (identified by "onwards" following the version identity); or

d) publications without mention of a specific version, in which case the latest version applies.

A non-specific reference to an ETS shall also be taken to refer to later versions published as an EN with the same number.

[1] ETS 300 414 (1995): "Methods for Testing and Specification (MTS); Use of SDL in European Telecommunication Standards; Rules for testability and facilitating validation"

[2] ETR 298 (1996): "Methods for Testing and Specification (MTS); Specification of protocols and services; Handbook for SDL, ASN.1 and MSC development"

[3] EG 201 015 (1997): "Methods for Testing and Specification (MTS); Specification of protocols and Services; Validation methodology for standards using SDL; Handbook"

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

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

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

[7] ITU-T Recommendations X.680 (1994): "Information technology - Open Systems Interconnection - Abstract Syntax Notation One (ASN.1): Specification of basic notation".

3 Definitions symbols and abbreviations

3.1 Definitions

data type: a set of data values with common characteristics (equivalent to the Z.100 term sort).

NOTE: When preceded by the word "abstract" then data type is always considered as part of the term "abstract data type" and not as the term "data type".

implementation option: a statement in a standard that may or may not be supported in an implementation.

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

validation: the process, with associated methods, procedures and tools, by which an evaluation is made that a standard can be fully implemented, conforms to rules for standards, satisfies the purpose expressed in the record of requirements on which the standard is based and that an implementation that conforms to the standard has the functionality expressed in the record of requirements on which the standard is based.

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

3.2 Abbreviations

For the purposes of the present document, the following abbreviations apply:

ASN.1	Abstract Syntax Notation N° 1
HMSC	High-level Message Sequence Chart
MSC	Message Sequence Chart
PId	Process Identity
SDL	Specification and Description Language

4 Introduction

The ITU-T Specification and Description Language (SDL) defined in Recommendation Z.100 [4] is a powerful tool for specifying the essential requirements of standardized protocols or services. The level of formality with which the SDL in a standard is expressed can depend on a large number of factors such as the size and complexity of the system to be standardized and the skills and experience of the standards writers. The specification of a protocol or service as a complete formal model enables the validation of the standard before approval and publication. However, well-constructed, formal SDL has a valuable role to play in providing a simple illustration of the process-related aspects of a standardized system. In order to gain the maximum benefit from the use of descriptive SDL, it is necessary for a consistent approach to be taken in its specification by all rapporteurs. By following the set of simple guidelines presented in this EG, it will be possible for the following benefits to be realized:

- Comprehension of the specification can be improved;
- Ambiguity can be avoided in the translation of the descriptive SDL into a validation model.

Achieving consistency in the presentation and level of detail specified across a wide range of standards is one of the keys to improving the perceived quality of ETSI's products.

The guidelines for the use of SDL for descriptive purposes are grouped in this EG according to the following broad classifications:

- naming conventions;
- presentation and layout of SDL processes;
- diagram structures;
- the use of procedures and operators;
- the use of decisions;
- communications and addressing;
- the specification and use of data;
- the use of Message Sequence Charts (MSC) in association with SDL.

Each of the guidelines is highlighted within the document in ***bold and italic text***. They are all collected together in tabular form in Annex B.

5 Naming Conventions

5.1 Identifiable entities

In common with most modern programming languages, SDL permits the use of alphanumeric names to identify individual entities within a specification. Examples of entities that can be identified in this way are:

- the system itself;
- blocks;
- processes;
- procedures;
- signal routes;
- signals;
- timers;
- variables and constants;
- signal lists.

5.2 Length of names

The SDL syntax places no restrictions on the number of characters that may be included in these names but, in practice, the limits associated with the target language (e.g., C or C++) must be respected. It is also worth noting that very long names can often be difficult to read. It is not possible impose a strict rule on the length of names but, as a general guideline, ***names of less than 6 characters may be too cryptic and names of more than 30 characters may be too difficult to read and assimilate.***

5.3 Use of non-significant characters

SDL is not sensitive to the case of characters within names and the capitalization of the first character of each word in a name is an acceptable method of delineation. As an example, the name "DeliverMessageContents" has exactly the same meaning if written as "delivermessagecontents". Although it works well in many cases, this method can result in names that are quite difficult to read if they contain acronyms or larger numbers of short words. Examples of these are:

InvokeCCBSSupplementaryService

AddOneToTheFirstItemOfOldData

One of the features of SDL that is different from most programming languages is its treatment of spaces and some other control characters. In most cases, these characters are ignored so that the name:

DeliverMessageContents

could equally well be written as

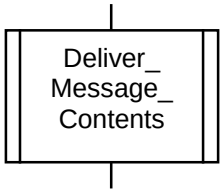
Deliver Message Contents

Note: Although the language allows the use of spaces within names, the available automatic tools do not currently support this feature and it is advisable to use only the underscore character as a word separator within a name. In addition, the ASN.1 notation that is generally used for specifying message structures in protocol standards does not permit the use of spaces in names.

It is also possible to split a name across more than one line by introducing an underscore followed by a sequence of spaces and/or the *carriage-return* and *line-feed* control characters. So, the example above could also be expressed as:

Deliver_
Message_
Contents

This is a very convenient notation when trying to fit a long name into a graphical symbol, thus:



In fact, the SDL syntax allows names to be wrapped across lines without the use of the underscore as a breaking character. However, some currently available automatic tools require that the underscore is used. This restriction can be beneficial as it makes it very clear to the reader that the lines of text are a single name rather than a comment or list of shorter names.

It is worth noting that the underscore character is only insignificant when used as a hyphenation symbol and that the name:

DeliverMessage

is not the same as:

Deliver_Message

although it is identical to

Deliver_
Message

When a name using underscores to separate words is wrapped over more than one line, it is necessary to include two underscore characters where the hyphenation occurs, thus:

Deliver_
_Message

Readability is improved if the same convention for separating words within names is used throughout a specification. The one case where a combination of methods is recommended is in the use of acronyms within names that use capitalisation as the method of separation. An underscore on each side of the acronym clearly delineates it from the remainder of the name, thus:

Invoke_CCBS_SupplementaryService

In most cases an underscore character between each word removes any possibility of misinterpretation and this is the approach that is recommended.

5.4 Reserved words

Although SDL permits great flexibility in the use of names, there are certain reserved words which are keywords of the language itself and which, consequently, cannot be used as names. In addition, reserved words may not be used within names where they are separated from other words by any non-significant characters except an underscore. A list of these reserved words can be found in .

5.5 Multiple use of names

SDL permits entities belonging to different classes to be given the same name. As an example, it is syntactically correct for a process within a block named "Alarm_Clock" also to be given the name "Alarm_Clock". In addition, because of the scoping rules of the language, it would be possible for a process within another block in the same system to be named "Alarm_Clock". If applied carefully, this multiple use of names can make a specification easier to read by apparently simplifying the system structure. For instance, in many protocol standards, particularly those specifying supplementary services, the system comprises

a small number of blocks, each of which contain only one process. In such situations, the use of the same name for the block and for its single process would improve the readability of the SDL (see Figure 1)

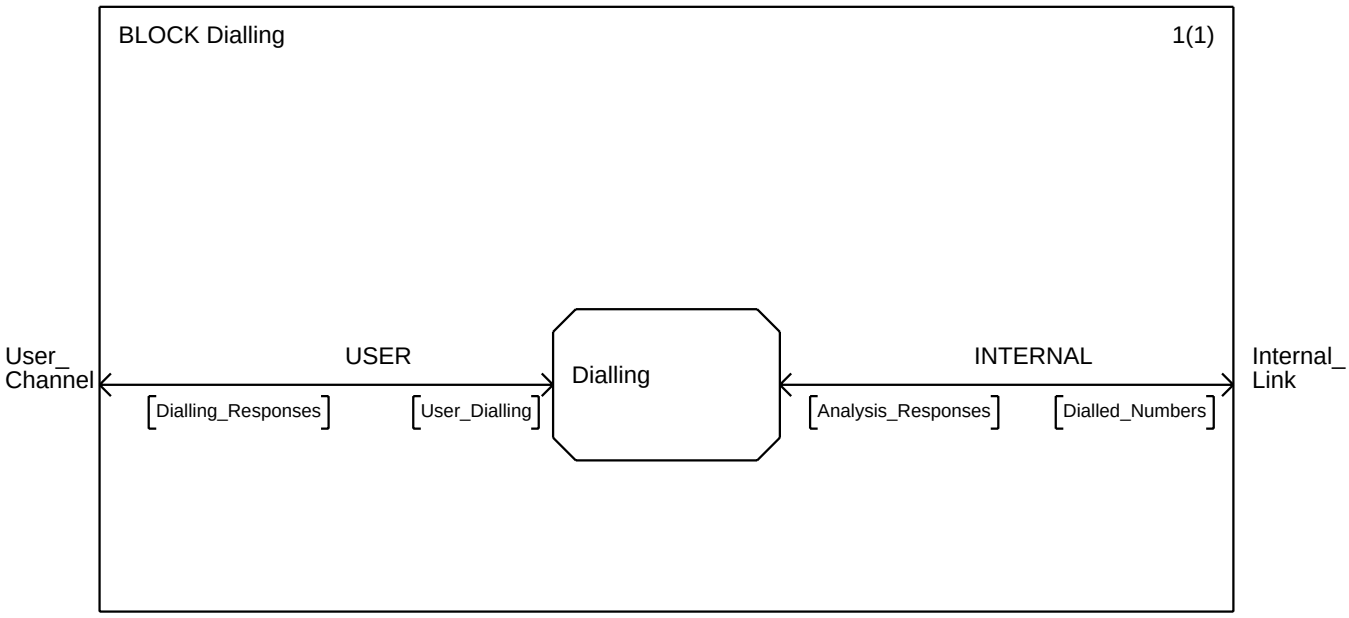


Figure 1: Example of a block and a process with the same name

In the example above, the use of the name "Dialling" for two entities causes no problems as there is little likelihood of any conflict. In fact, it helps to make the system easier to understand because it is clear that the process "Dialling" is logically equivalent to the block of the same name.

In more complex models where each block is made up of a number of processes and where there are many data items, the use of a single name for multiple entities is likely to cause confusion and should be avoided.

When using ASN.1 it is the convention that type references and values are distinguished by the consistent use of upper and lower case characters. ***The name of an ASN.1 type (i.e., a type reference) should start with an upper case letter, and names of values should start with a lower case letter***, for example:

```
Dog ::= SEQUENCE {
        poodle      Poodle,
        spaniel     Spaniel,
        alsation    Alsation,
        boxer       Boxer }

Poodle ::=   BOOLEAN
Spaniel ::=  BOOLEAN
Alsation ::= BOOLEAN
Boxer ::=    BOOLEAN
```

For readability the name *poodle* is preferable to *pOODLE*, even though the latter is, strictly speaking, allowed by the convention.

NOTE: Although SDL is not case sensitive the convention can still be applied in SDL specifications that use ASN.1 because ***name***s and type references are not the same kind of objects and can, according to SDL rules, have the "same" name.

5.6 Making names meaningful

The freedom and flexibility that SDL allows in the construction of names can be used to great benefit in improving the readability of a specification. If there is an entity whose function is to represent an alarm clock then it can be called "Alarm_Clock" and there are no constraints to force the use of a more cryptic name such as "Alm_Clk". However, this freedom can be abused and it would be quite legitimate for the alarm clock to be given the name "The_Thing_Beside_The_Bed_That_Makes_A_Loud_Noise_In_The_Morning" which is equally as unacceptable as the cryptic style.

Apart from the general recommendations above, certain specific guidelines apply to each group of identifiable entities.

5.6.1 Process names

By giving processes names that represent the overall role that they play within the system, it is possible to distinguish process names from procedure names. If carefully chosen, they can help to link the SDL back to the corresponding subclauses in the text description. Examples are:

Originating_PINX;

Scenario_Management;

Functional_Entity_FE2;

Alarm_Clock.

As can be seen, these names are all *nouns* which indicate the general function of the process.

5.6.2 Procedure names

Procedures are the key elements in breaking a complex process down into meaningful layers (see subclause 8.1). For his to be effective, ***the names chosen for procedures should indicate the specific action taken by the procedure***. Examples are:

Extract_Calling_Number_From_SETUP;

Get_User_Profile_From_Database;

Send_Response;

Ring_Alarm_Bell.

The names chosen here are all *verb phrases* indicating the specific activity to be carried out by the procedure.

5.6.3 Signal and Signal List names

There are often more constraints on the length of signal and signal list names as they usually have to appear in quite small spaces in SDL symbols. It is, therefore, more difficult to arrive at meaningful names for signals and signal lists. However, poor naming of signals can make SDL very difficult to read, even when most other aspects are well presented. For example, the name "Rep_Sgl_Err" could easily be interpreted to mean:

Report Signal Error;

Report Single Error;

Repeat Signal Error;

Repeat Single Error.

The obvious method for overcoming this problem is to express the name in full as, for example "Report_Signal_Error" but this is quite a long name. This can be overcome by using unambiguous abbreviations or abbreviations that are in common use. In the example above, "Err" is generally accepted as meaning "Error". Also, changing "Sgl" to "Sig" would make it much clearer that it was an abbreviations for "Signal" not "Single". ***If possible, it is advisable to leave at least one significant word in the name unabbreviated as this can help to provide the context for interpreting the remaining abbreviations***. So the example above would be acceptable if expressed as "Report_Sig_Err".

In straightforward SDL specifications where object orientation has not been used to any great extent and where all signals between one block and another can be logically grouped together, ***signal list names can be chosen to indicate the origin and the destinations of the associated signals***. Examples of this approach are as follows:

Home_PINX_to_Visitor_PINX:

HLRA_to_HLRB;

LocalExch_to_User

AccessManagement_to_CallControl

For bi-directional channels where the signals are symmetrical, i.e., the same signal list applies to both directions, it may still be possible to identify the two processes or blocks at each end of the channel. One option is to use "between...and..." when assigning names to signal lists, thus:

between_HLRA_and_HLRB

Although explicit in nature and quite attractive in instances where the process or block names are short, this construct can result in excessively long names such as:

between_AccessManagement_and_CallControl

In such cases it would be acceptable to simply link the two block or process names with an underscore:

AccessManagement_CallControl

A second alternative is to relate the signal list name to the functional grouping of the signals. Examples of this approach are:

UNI_Messages

Mobility_Management

User_Input

This approach does not provide any topographical information but it can be a very useful method of naming signal lists in a standard using object orientation in the specification of system blocks and processes.

5.6.4 State names

In most protocol standards, the SDL specification includes a large number of states and it is often tempting to assign cryptic and sequential names such as "state_5" or "N3". Taking the time to formulate meaningful names for each state can add significantly to the readability of an SDL specification.

A state name should clearly and concisely reflect the status of the process while in that state. Examples of such names are:

Idle

Wait_For_SETUP_Response

Timing_Signal_Delay

If it is important to number states then this should be done in conjunction with meaningful names such as:

Releasing_01

Timing_Response_4

5.6.5 Names of Variables and Constants

It is more difficult to specify some simple guidelines for the construction of names for variables and constants as they have widespread and diverse uses. It is still important to ensure that the name is meaningful in the context of the SDL specification. ***The name chosen for a variable should indicate in general terms what it should be used for***. For example:

SETUP_message_contents

User_Input

Alarm_Time

Names used to identify constants can be more specific by indicating the actual value assigned to the constant. For example:

User_Not_Known

Twenty_Five

Characters_A_To_Z

5.6.6 Timers

Although the use of meaningful timer names, such as Response_Sanity_Timer, would improve the overall readability of a specification, it has become universally accepted practice to use the shorthand T1, T2, T3 etc. for timers within standards for protocols. To avoid confusion, the "Tn" notation should be used when naming timers unless an opportunity arises to use extended names in a completely new project where the use of the shorthand is not already established.

6 Presentation and layout of process diagrams

The syntax of SDL allows great freedom in the presentation and layout of both text and graphical symbols. Good presentation can considerably improve the readability of an SDL specification whereas bad presentation can render it unintelligible. It is also worth noting that a single error resulting from the misunderstanding of a poorly presented diagram can be much more costly than all the pages of paper saved when packing symbols and diagrams tightly.

It is in process descriptions that presentation and layout have the most impact and the following aspects should be considered within a standard:

- process flow on a page;
- spreading process diagrams over more than one page:
- use of text extension symbols;
- alignment and orientation of symbols.

6.1 Process flow

SDL allows the lines connecting symbols to flow in any direction across a page. As an example, the process shown in Figure 2 is legal SDL but is quite difficult to read.

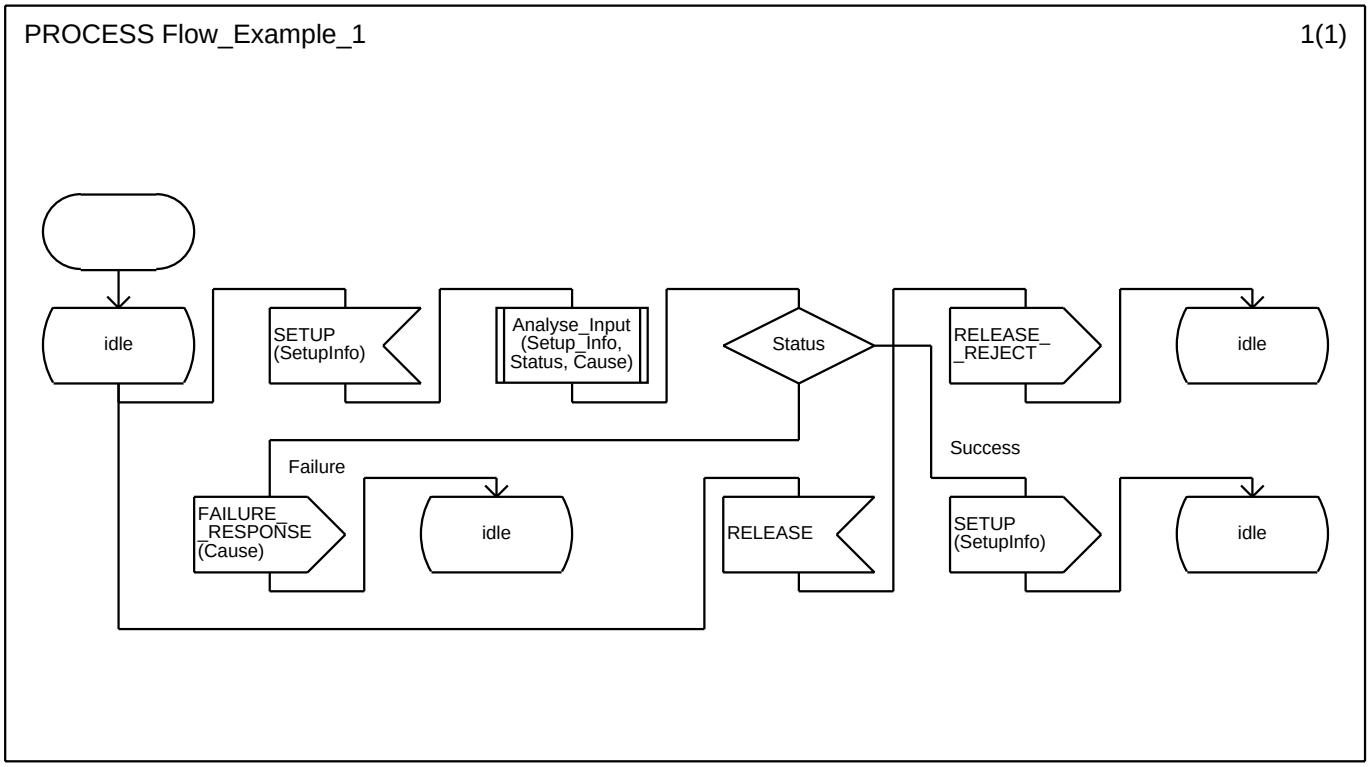


Figure 2: Example of poor layout of legal SDL

The readability of this process is greatly improved simply by laying it out in a "top-to-bottom" form, as in Figure 3.

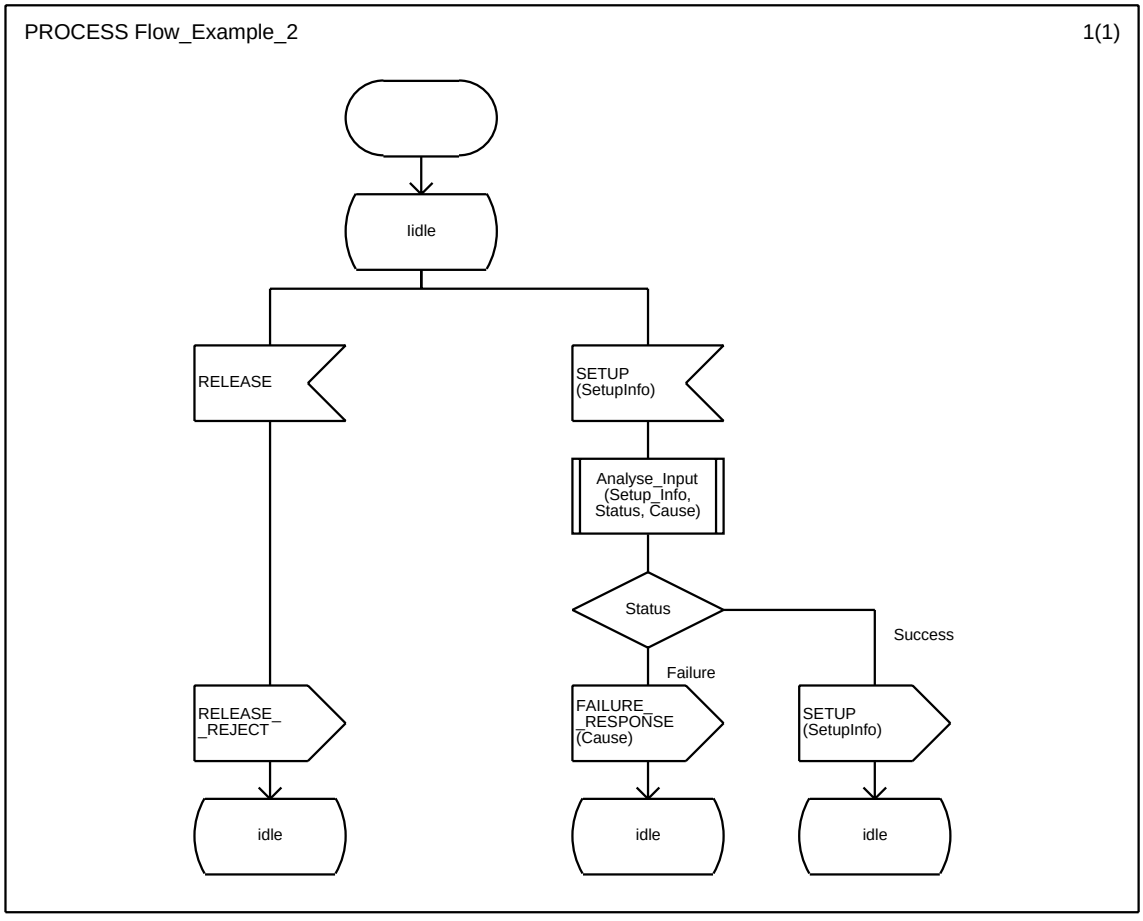


Figure 3: Example of improved layout

The orientation of SDL process symbols is such that they naturally flow vertically and it is, thus, easier to read diagrams that follow this convention. Thus, ***the flow of SDL process diagrams should be from the top of the page towards the bottom***.

6.2 Process diagrams covering more than one page

6.2.1 Linking process segments across page boundaries

In most cases within standards it is not possible to constrain SDL process descriptions to one page. Only two options exist for breaking a diagram across a page boundary without affecting the readability. These are:

- using the NEXTSTATE symbol;
- using a connector symbol;

If it can be accommodated within the general structure of a description, ***the flow on a page of an SDL process should terminate in a state*** (i.e. the NEXTSTATE symbol) as shown in Figure 4 and Figure 5. In general, this makes them easier to read. In addition, ***states that are entered from NEXTSTATE symbols on other pages should always be placed at the top of the page.***

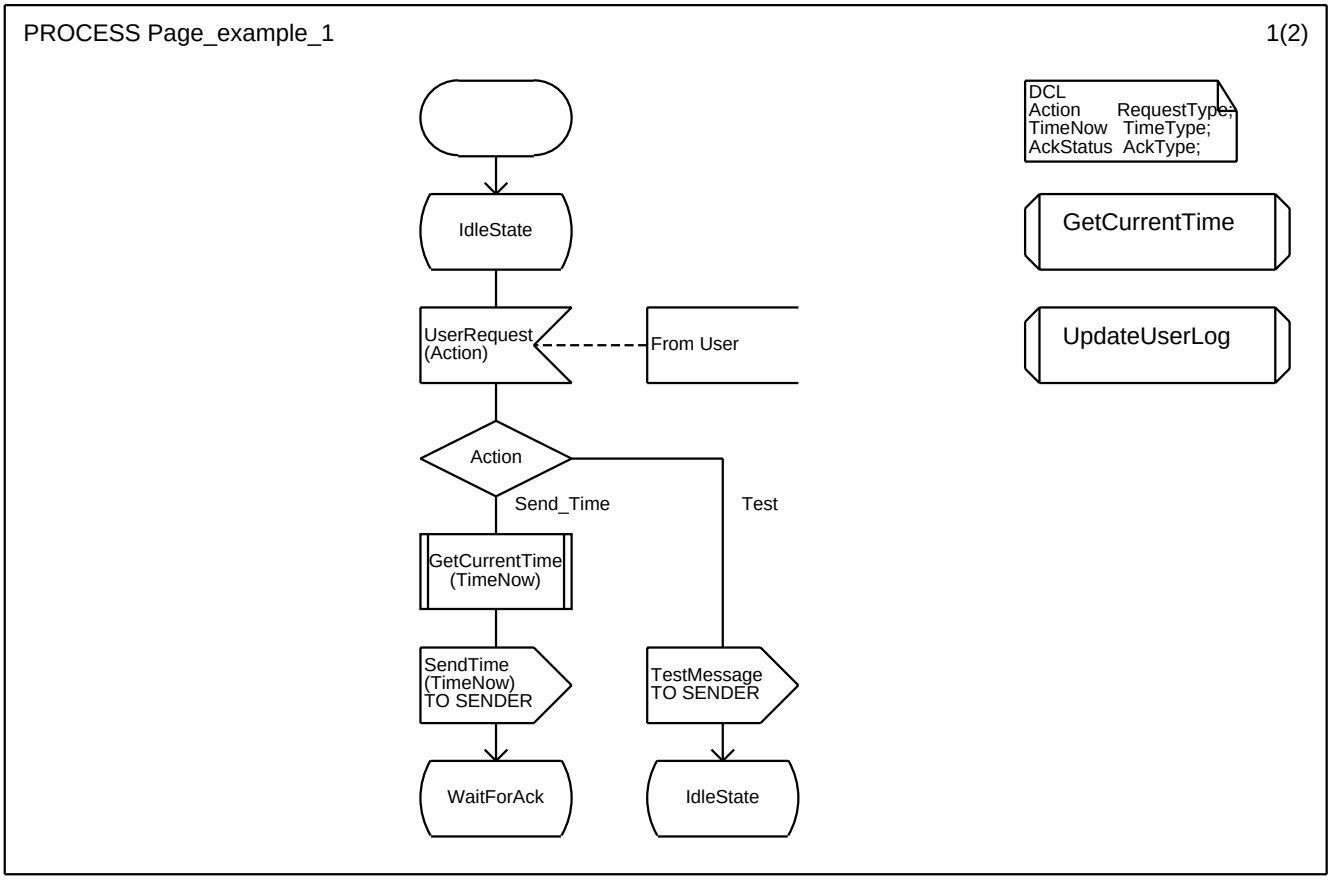


Figure 4: Paging using NEXTSTATE symbol (page 1)

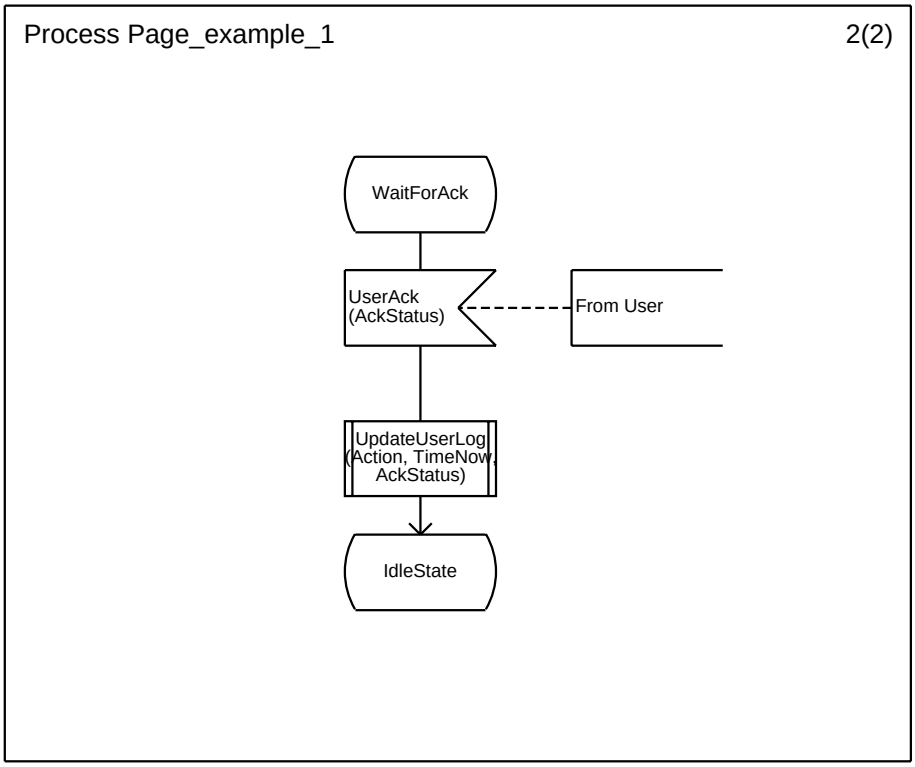


Figure 5: Paging using NEXTSTATE symbol (page 2)

Although it would be possible to draw the example shown in Figure 4 and Figure 5 in a single thread with the "*WaitForAck*" state embedded part-way through, it is easier to locate individual states in a more complex specification if each thread is limited to a single transition (the processing between one state and the next one). ***Where transitions are short and simple they can be arranged side-by-side on a single page*** as shown in Figure 6. However, ***when two or more transitions are shown on one page, there should be sufficient space between them to make their separation clear to the reader***.

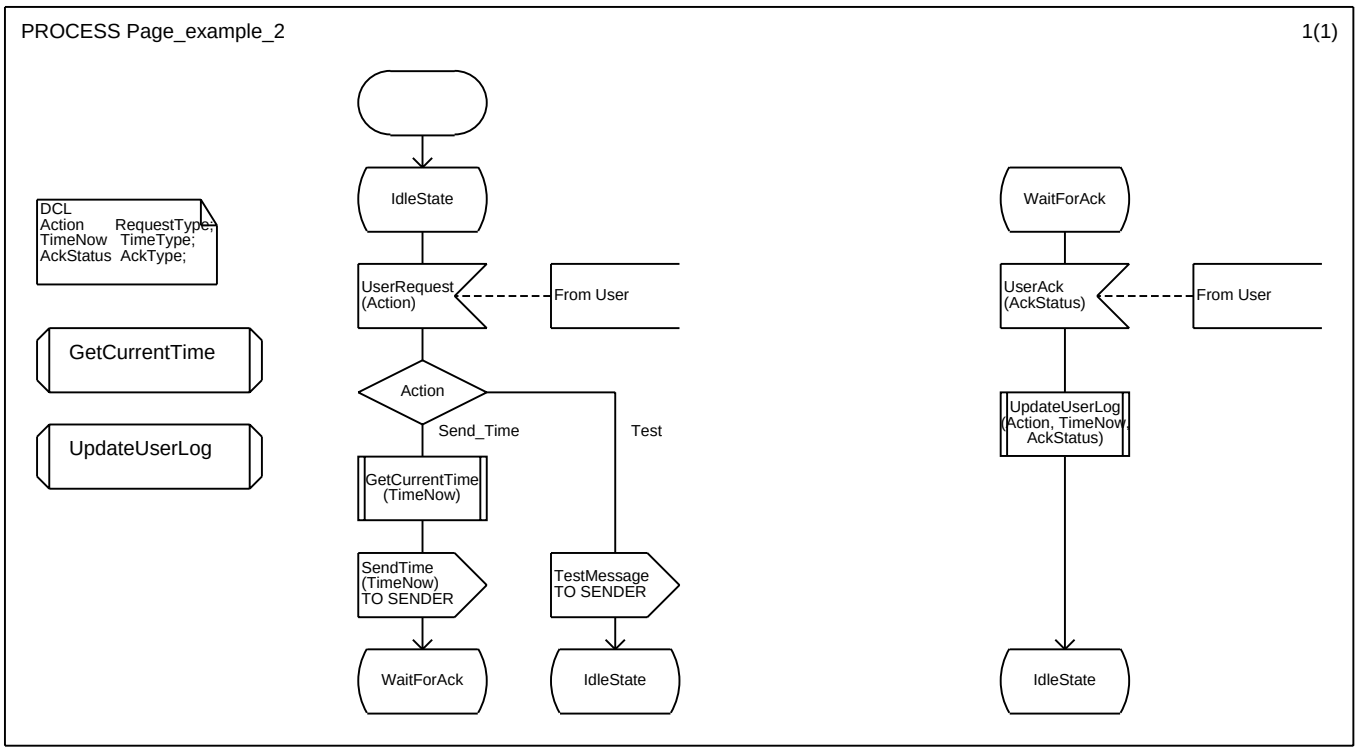


Figure 6: Transitions aligned on a single page

When a single transition extends beyond the length of one page, a connector symbol can be used to provide a link to the next page. An example is shown in Figure 7 and Figure 8.

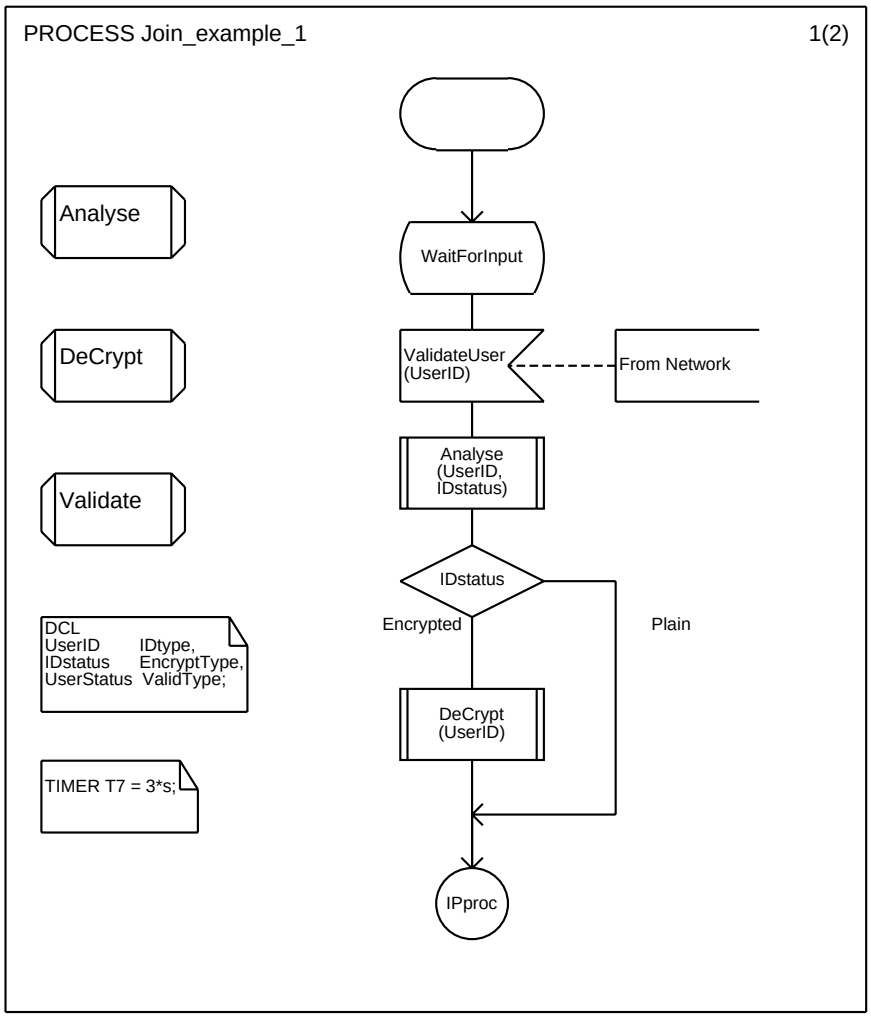


Figure 7: Paging using a connector symbol (page 1)

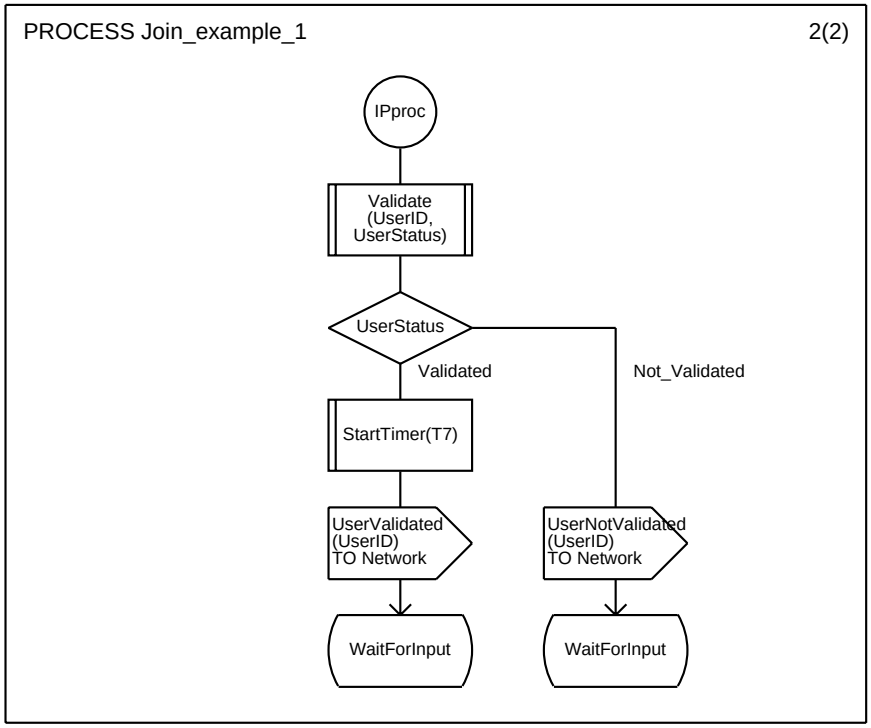


Figure 8: Paging using a connector symbol (page 2)

As can be seen in Figure 7 and Figure 8, the syntax of SDL allows a connector symbol to have a process flow line to it or from it but not both. Figure 9 shows how it is possible for a connector to be attached to a symbol anywhere on a page. These can be difficult to locate and so, to avoid confusion, ***connector symbols should generally only be used to provide a connection from the bottom of one page to the top of another***. However, long transitions can often be avoided by careful use of procedures (see subclause 8.1).

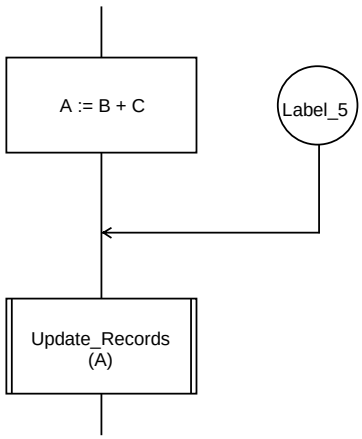


Figure 9: Example of poor use of a connector symbol

6.2.2 Symbols common to all pages

An SDL process description should not be considered to be simply a "flow-chart" specifying a sequence of actions and decisions to be taken by a particular entity. In order to be complete, a process description may include the following:

- a specification of formal parameters;
- variable, signal and data definitions;
- procedure references;
- the process graph, itself.

Symbols such as procedure references and text boxes containing DCL, TIMER and other declarative statements are valid for all pages of the process in which they appear. The language syntax allows them to be drawn on any page but, for easier reading, ***all reference symbols and text boxes containing common declarations should be collected together at a single point within the process chart.*** For simple processes, and where space allows, these symbols can be placed together on the first page with the first transition, as can be seen in Figure 6 and Figure 7. In other cases, a separate page (or pages, if necessary) can be used to collect these symbols together.

To further improve the readability of the SDL, ***a new text box symbol should be used for each different type of declaration*** (for example, variable declarations, timers, signal specifications, data type specifications and formal parameters)

6.3 Text extension symbols

The SDL symbols are not always large enough to contain all of the text necessary to specify the task represented by the symbol and if the character size is set to a value that makes it readable, the text spills over into the area surrounding the symbol as can be seen in Figure 10.

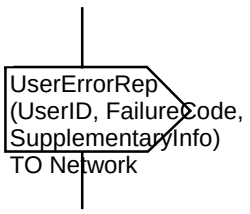


Figure 10: Text overflowing a symbol

This can be difficult to read and, in the strict sense of the language, is syntactically incorrect. Therefore, ***when the text associated with a task symbol overflows its symbol boundaries, a text extension should be used to carry the additional information*** as shown in Figure 11. The syntax of SDL specifies that the text in the extension symbol is added after the text in the task symbol. To avoid misinterpretation, care should be taken to ensure that the text extension symbol appears to the right of or below the task symbol unless all of the text is placed in the extension symbol.

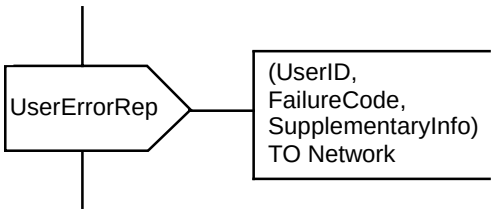


Figure 11: Use of Text Extension symbol

Even in cases where the text does not overflow the symbol, this is a useful presentation method which can be used to separate the signal name from the parameter list in inputs and outputs. For reasons of clarity, it is not advisable to split the parameter list between the primary symbol and the extension.

As an alternative to the use of a text extension symbol, SDL permits the re-sizing of both a task symbol and the text contained in it.

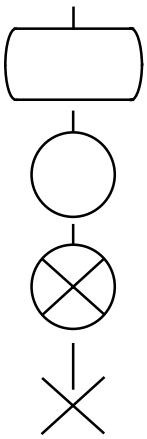
6.4 Alignment and orientation of symbols

6.4.1 Alignment

SDL places no semantic significance on the placement and alignment of symbols but a process page that is carefully arranged and not over filled with symbols and connecting lines will always be easier to read and interpret than one that is not.

There is no particular benefit to be gained by aligning symbols of a particular type except that ***symbols that terminate the processing on a particular page should be aligned horizontally*** to make it easier for the reader to identify all of the points where processing ceases or continues on a new page or thread. These symbols include:

- NEXTSTATE symbol
- Connector symbol
- RETURN symbol
- STOP symbol



In the example shown in Figure 12 , the processing on the page can end in a number of different states. The alignment of all of the associated NEXTSTATE symbols at the bottom of the page makes it clear what all of these possibilities are.

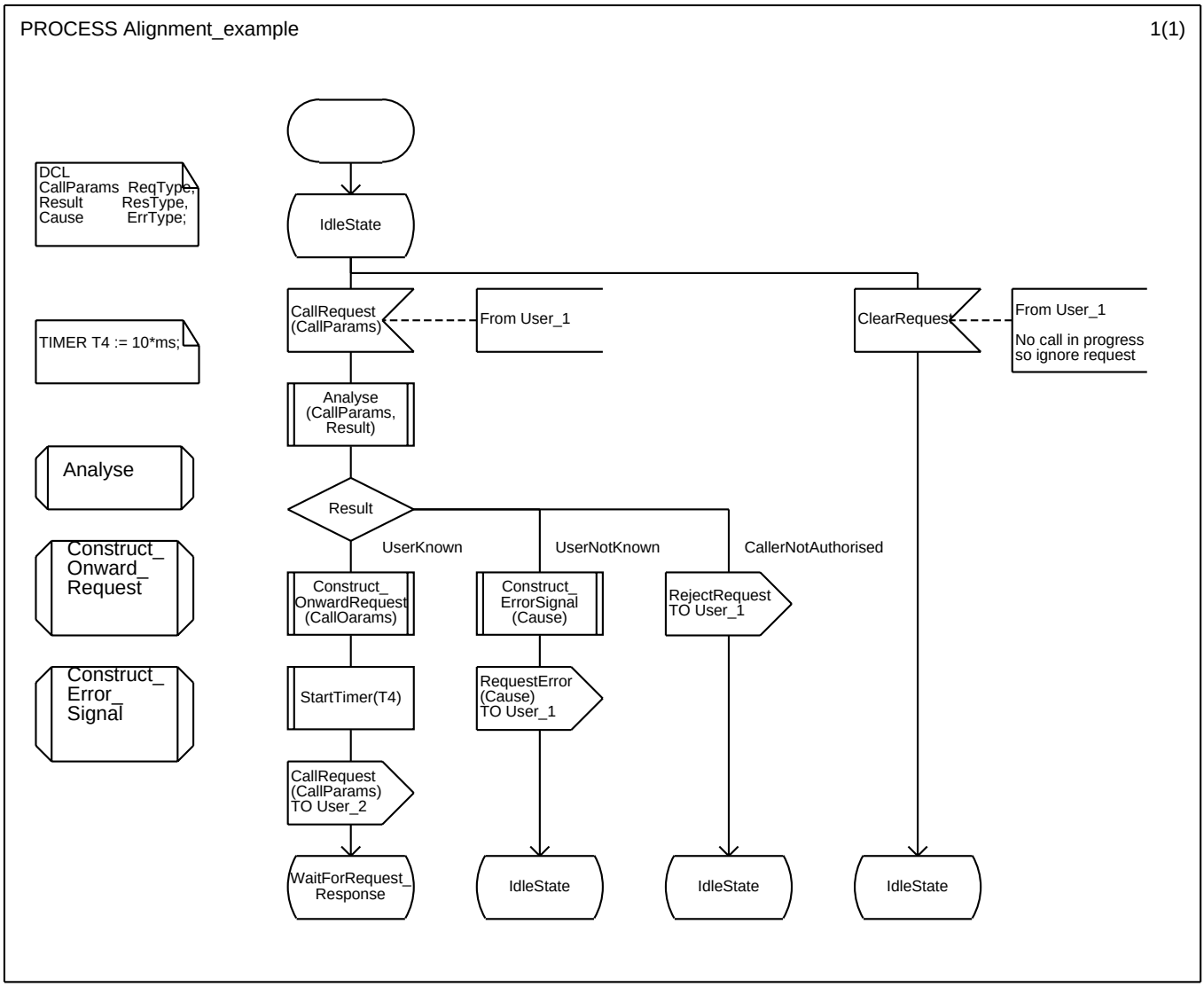


Figure 12: Example showing the alignment of NEXTSTATE symbols

6.4.2 Orientation

Most SDL symbols are symmetrical and, thus, cannot be shown in different orientations. INPUT and OUTPUT symbols are different in that they can be shown either right facing or left facing, thus

SDL accepts both orientations as correct but does not assign any specific meaning to either. However, ***in simple systems where each process communicates with only one or two other processes, the orientation of INPUT and OUTPUT symbols can be used to improve the readability of the SDL***. This should not be considered to be a substitute for the use of a "From" comment on an INPUT or the TO and VIA statements in an OUTPUT as described in subclause 10.4. ***The significance of the orientation of SDL symbols should be clearly explained in the text introducing each process diagram***.

7 Structuring behaviour descriptions

The behaviour of an SDL system is mainly described in process diagrams which represent the topmost layer of the behaviour specification. Partial behaviour descriptions can also be given in procedure and operator diagrams. For readability it is important that the behaviour specification is organized and presented in such a way that each reader can easily find information of particular interest. It is important to bear in mind that a standard is often read in different contexts at different times. For

example, at one time it may be used to gain an overall understanding of the specification while at another time it may be read in order to extract some specific details.

7.1 Basic structuring principles

The key structure within a protocol or service behaviour description is the relationship between a process state, the events that trigger some form of process reaction, the actions that are taken and the resulting state. Process graph should be structured in such a way that these relationships are easy to see. ***A state, input and the associated transition to the next state should be contained within a single SDL page***.

7.2 Structuring using procedures and operators

Within a standard, the most important actions taken between process states are the generation of output signals. If the flow of control leading from one process state through input and outputs to the next state cannot be contained within a single SDL page, procedures and operators should be used to hide some of the other detail, as described in clause 8.

7.3 Emphasizing the difference between normal and exceptional behaviour flows

Within their textual descriptions, standards often make the distinction between normal and exceptional cases. This distinction can also be used effectively in the SDL. Figure 13 shows an example where the analysis result splits the flow into normal behaviour which is specified on the same page and error handling which is specified on another page. This allows the reader to concentrate on the normal behaviour and to look at the various error handling situations if and when that is required.

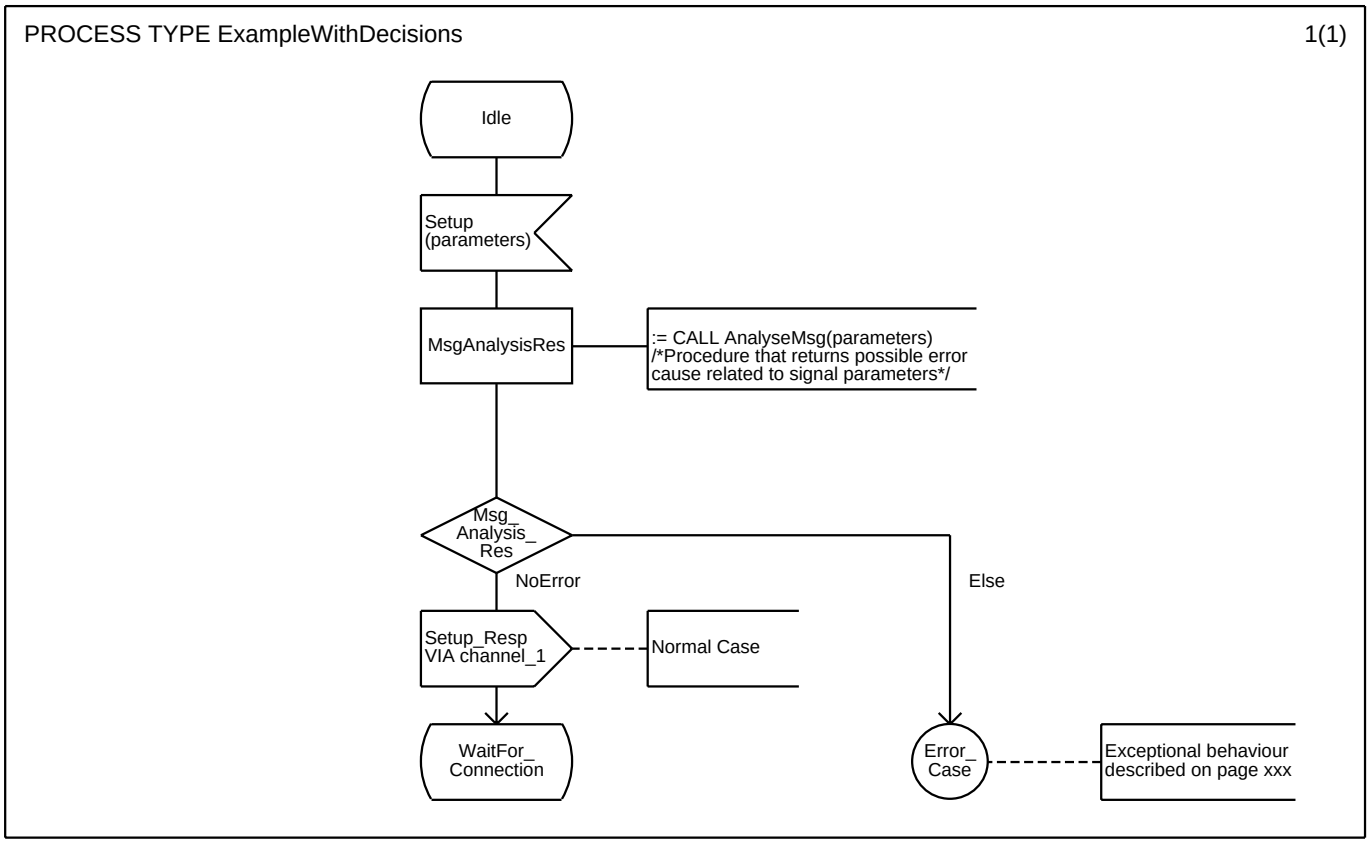


Figure 13: Part of process diagram showing only normal behaviour flow

The separation of normal and exceptional behaviour may also bring benefits to the standard development process, so that specification of normal behaviour becomes stable before error handling issues are addressed. Wherever it is appropriate and convenient, ***process diagrams should segregate normal behaviour from exceptional behaviour.***

8 Using procedures and operators

8.1 Procedures

In common with most programming languages, SDL procedures provide a mechanism for the modular specification of behaviour that can be used in different contexts.

An important aspect from the point of view of a standards specification is that procedures can be used to hide distracting detail. By moving detail to procedures, the reader is presented with a clear and concise overview of the required behaviour even though the detail can be viewed if it is required. ***The use of procedures to modularise specifications and to 'hide' detail is strongly recommended***.

As an example, there may be a requirement in a standard that the contents of an incoming message are analysed and that subsequent processing be based on the results of the analysis. The method of analysis is not an issue for the standard and, as can be seen in Figure 14,such detail can distract from the main purpose of the process. If, as is shown in Figure 15, the detail is removed to a procedure, the reader is left with the information that the message is to be analysed but without the distraction of how the analysis is undertaken.

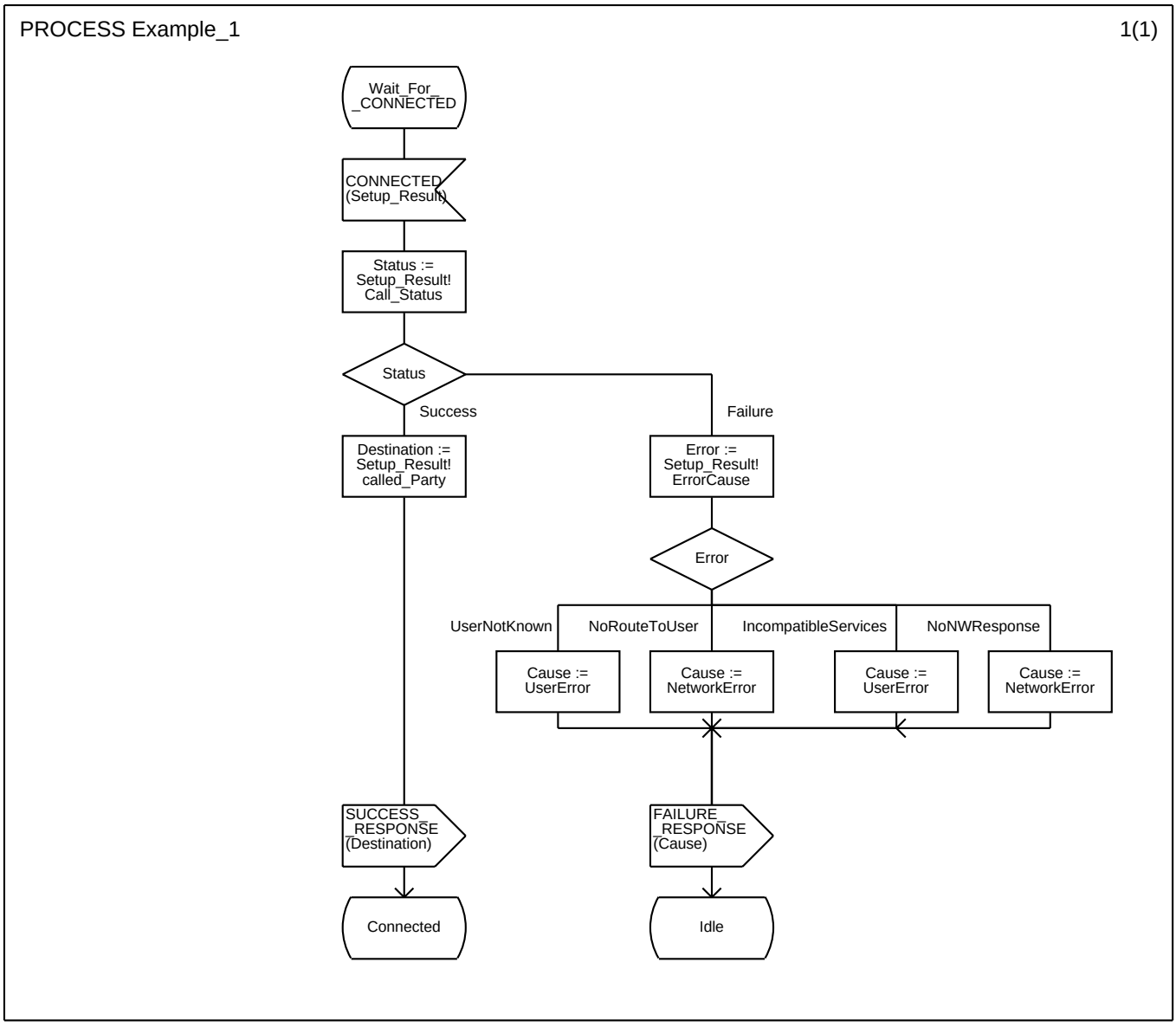


Figure 14: Message analysis example without the use of a procedure

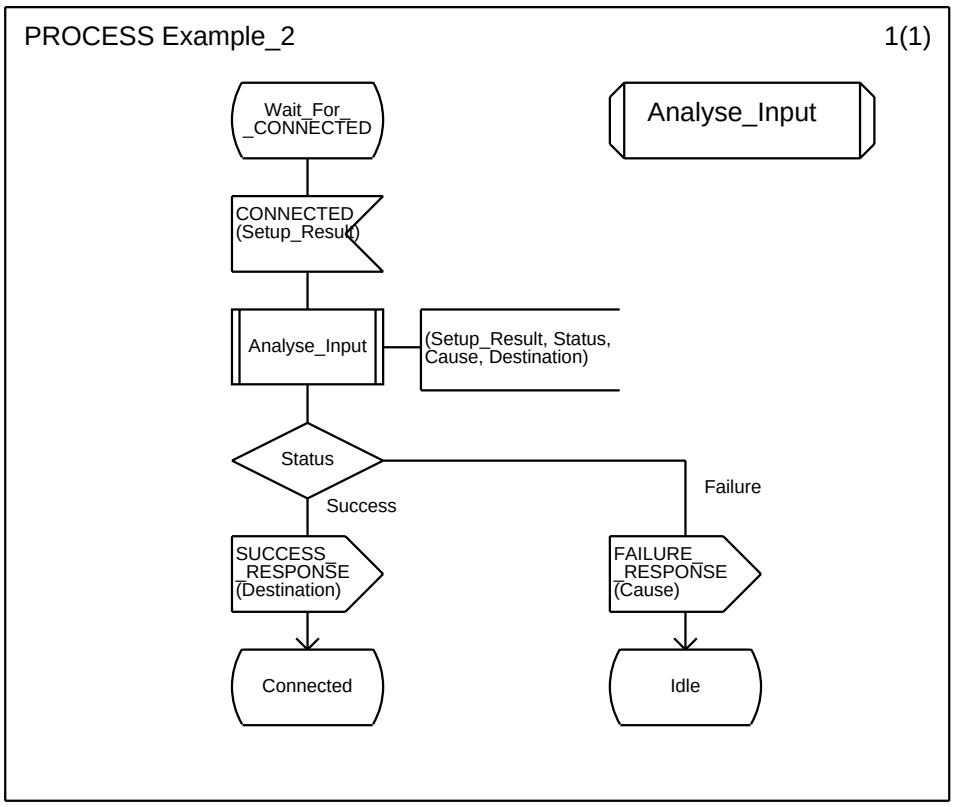


Figure 15: Message analysis example using a procedure

8.1.1 Procedure interface (parameters and return values)

A procedure interface specification identifies a set of parameters for the procedure and defines how these parameters are passed to and from the procedure. ***All data relevant to the real behaviour of a procedure should be specified in the parameter list and return value (if any).*** This means that an interface is specified which allows the contents of the procedure to be updated at a later stage without affecting the other parts of the specification.

The interface specification for a procedure can include

- data items that are to be passed to the procedure;
- data items that are to be passed back to the calling process. These returned parameters can be specified as:
  - a list of one or more items which appear in the calling statement. An example of a call to such a procedure is as follows:

```
Get_Position(identifier, X_Coord, Y_Coord);
             ^^^^^^^^^^  ^^^^^^^  ^^^^^^^
             |           |        |
             |           |        Output parameters
             |           Output parameters
             Input parameter

FPAR
IN          identifier,
IN/OUT      X_Coord,
IN/OUT      Y_Coord;
```

  - A single value associated with the procedure name itself. The following is an example of a call to a procedure of this type:

```
Calling_Party := CALL Extract_Originator(Setup_Data);
                      ^^^^^^^^^^^^^^^^^  ^^^^^^^^^^
                      |                  |
                      |                  Input parameter
                      "Output" parameter

FPAR
IN          Setup_Data;
RETURNS     Party_Number;
```

Figure 16 and Figure 17 show simple examples of each of these procedure types while Figure 18 shows how the procedures could be called.

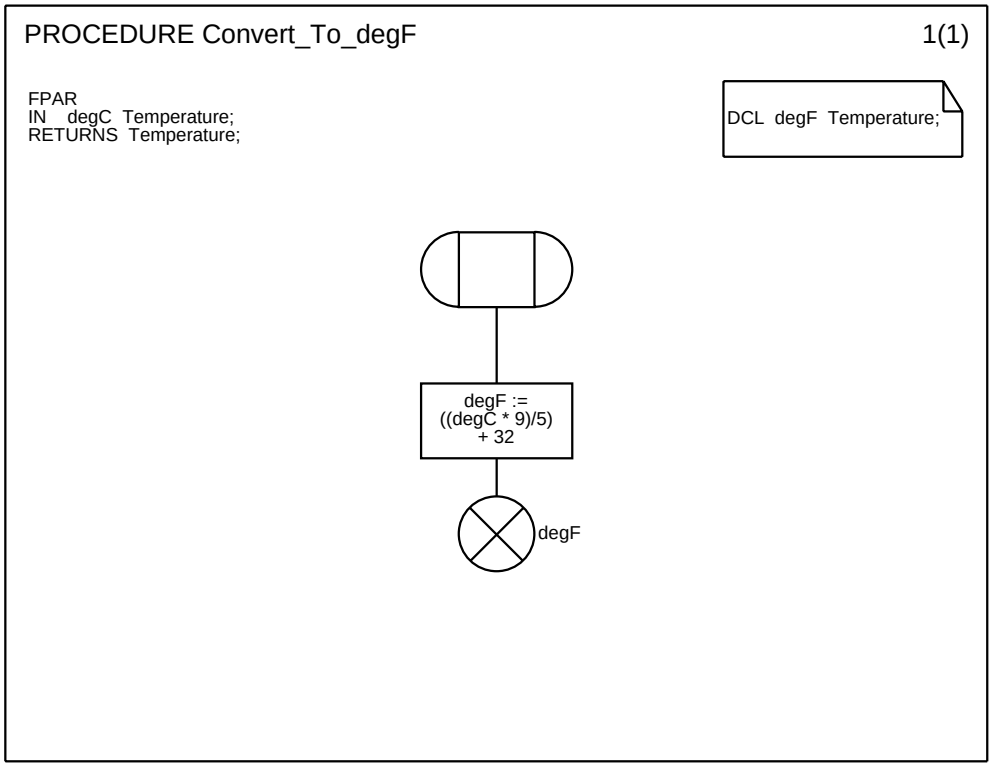


Figure 16: Example of a value-returning procedure

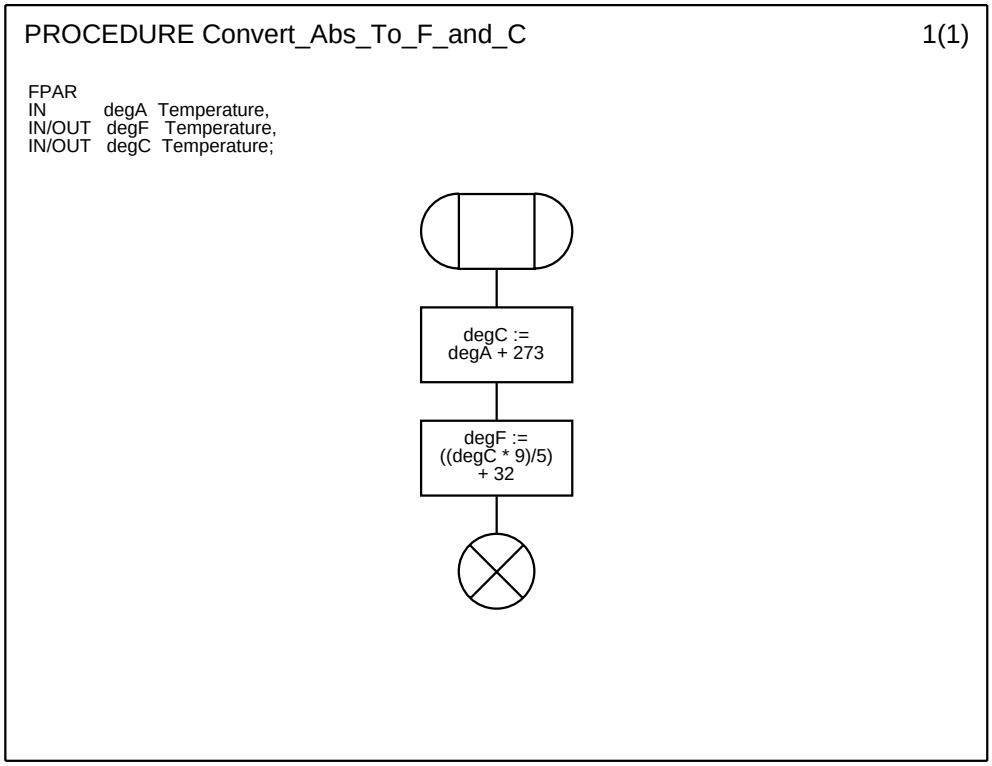


Figure 17: Example of a procedure returning values in the parameter list

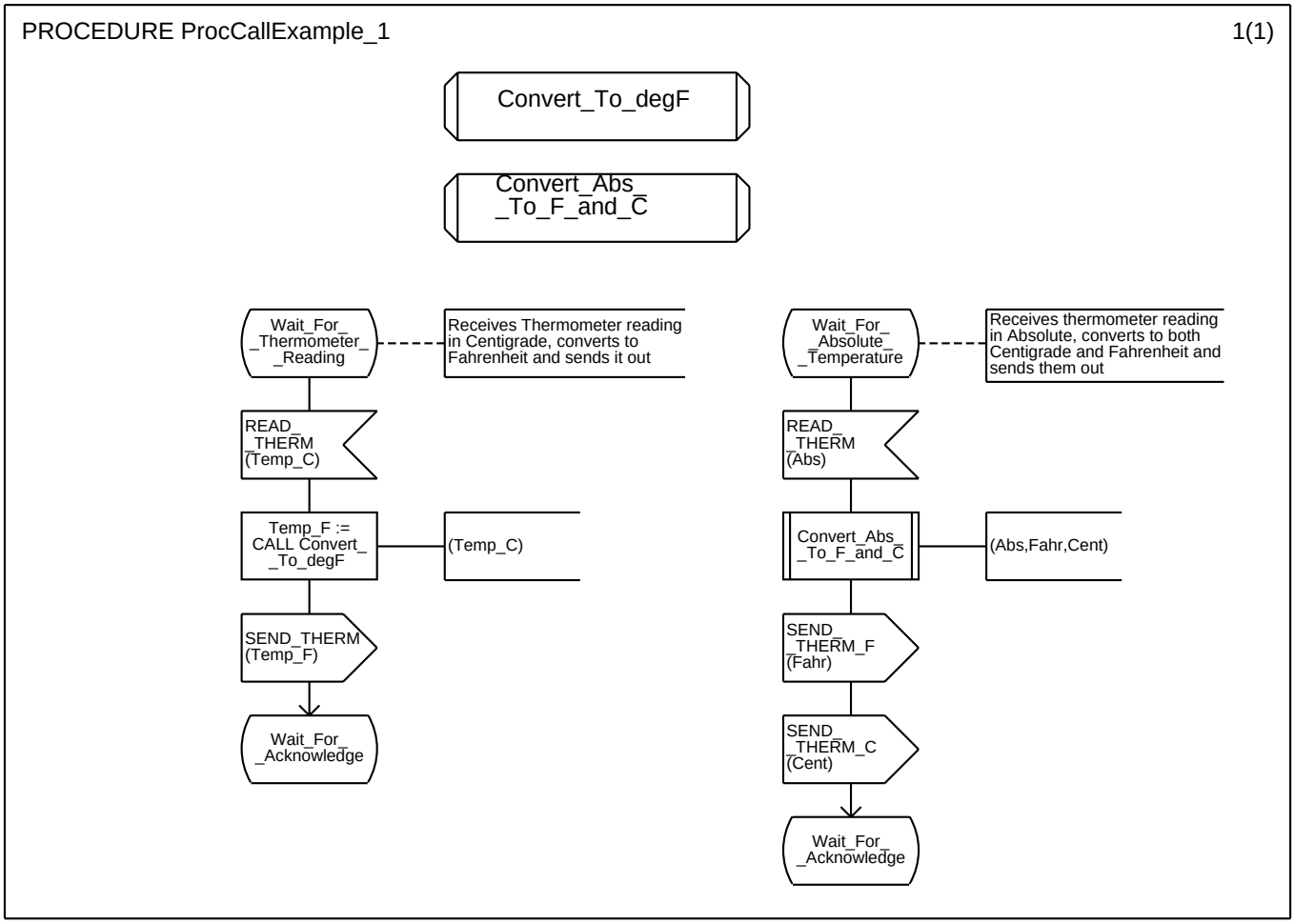


Figure 18: Examples of procedure calls

Procedures which return a value associated with the procedure call itself (Figure 16) can be used in place of variables in decisions, assignments, and output parameter lists to hide some of the detailed processing which is not essential to the understanding of a standard. However, ***in most cases it is preferable to use Operators instead of Value-Returning Procedures.***

In existing standards, it is common to see informal text included in an SDL task box as an item of useful, and often normative, information. For example:

This notation is very easy to understand but it is not possible simulate or validate the action in the symbol. According to the strict definition of SDL, the text, "Stimulate the release of the basic call", is interpreted as a name at the start of an incomplete, and therefore incorrect, assignment statement. To make such expressions formal and executable, ***convert informal text descriptions of actions into procedure calls and replace the task symbols with a procedure symbols***, thus:

Note that in converting such text into a procedure call it may be necessary to add parameters to fully formalize the interface to the procedure.

Procedures defined within the scope of the process calling them can access the variables belonging to that process. Accessing data in this way, particularly writing to a process variable from within a procedure, can result in a confusing specification. In order to avoid the possibility of this confusion and any other unexpected side-effects ***procedures should only read and write to variables that are passed to the procedure in the parameter list or are declared within the procedure itself***.

8.1.2 Procedure body

The behaviour specified within a procedure can be more or less complex depending on the need. In the above example the message analysis procedure could simply make a reference to the relevant text part of the standard where the message analysis is described in detail. This is a reasonable approach if the standard is to be validated by visual inspection or walk-through. If, however, automatic tools are used to check the syntax and semantics of the SDL, such a specification would be considered incomplete. In order to complete the specification, the following methods can be used:

1. provide a "dummy" procedure that does nothing (Figure 19);

   - this is adequate where the detailed behaviour of the procedure is not considered to be normative even though the overall function of the procedure may be. Figure 15 above is an example of this. It is important to include the dummy procedure in the standard as its "FPAR" and "RETURNS" statements serves to define what, in a full implementation, the interface should be between the calling process and the function expected of the procedure.

2. provide a procedure that uses the ANY function to arbitrarily return one of the possible values without actually specifying how the value is determined (Figure 20);

   - this is an ideal approach if the SDL model is to be validated by an automatic tool as it ensures that all possible return values are evaluated during the validation process.

3. provide a procedure that specifies in detail the behaviour expected (Figure 21).

- this is the best approach in cases where the procedure has been used to hide complex behaviour but that behaviour is considered an important and normative part of the standard. It would also be advisable to use this approach when simulating the full behaviour of the model.

Whichever method is chosen, ***procedures should specify a level of detail that is suitable for the particular purpose of the standard***. At a minimum, the procedure should express the requirements it is modelling, even if this is simply a comment or a reference.

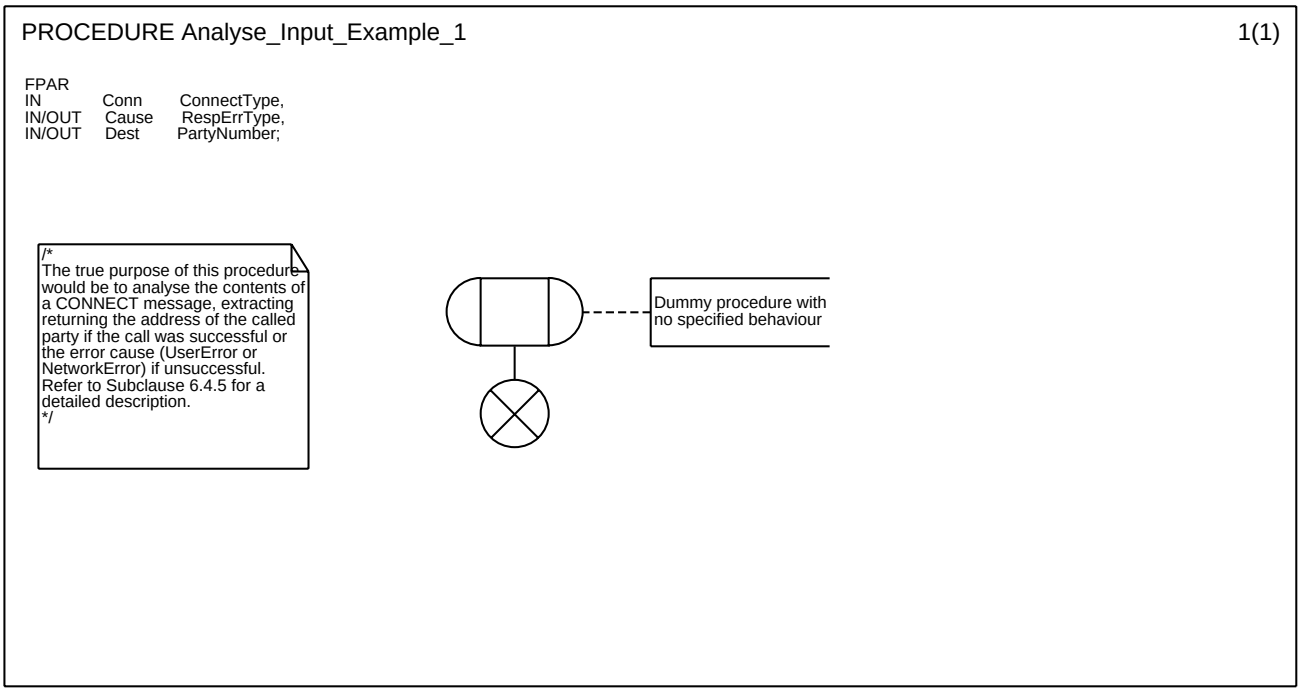


Figure 19: Example "Dummy" procedure

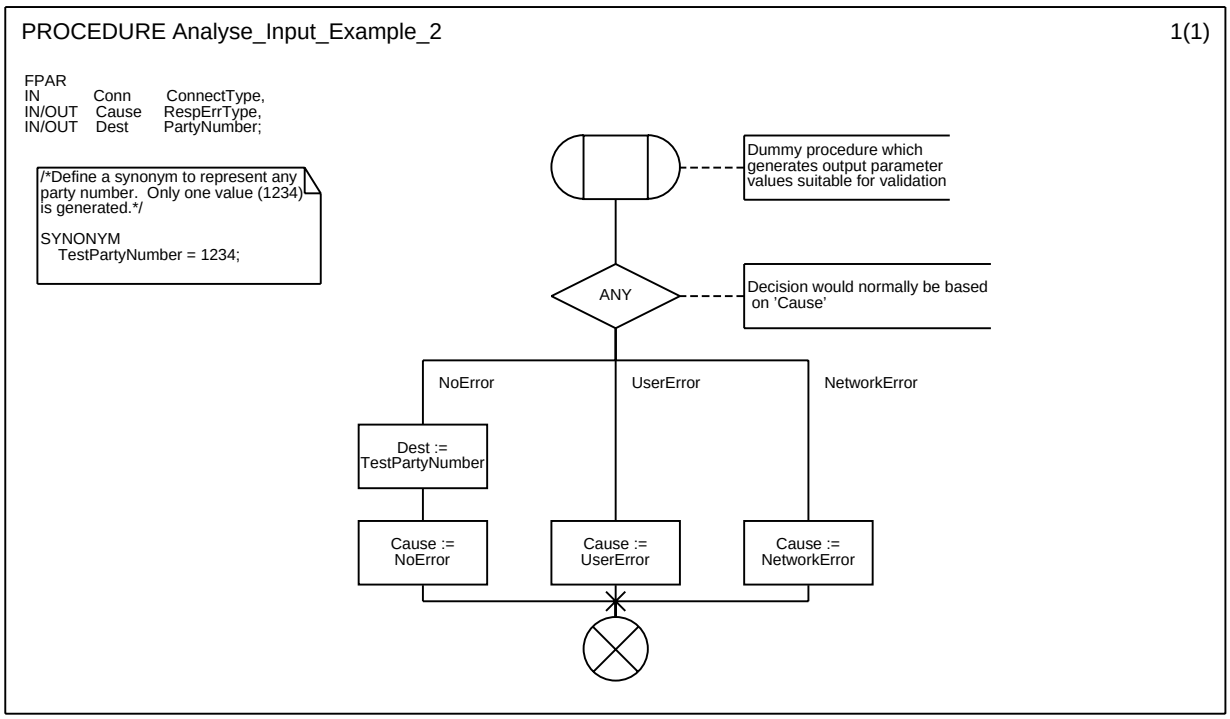


Figure 20: Example of a simple procedure suitable for validation purposes

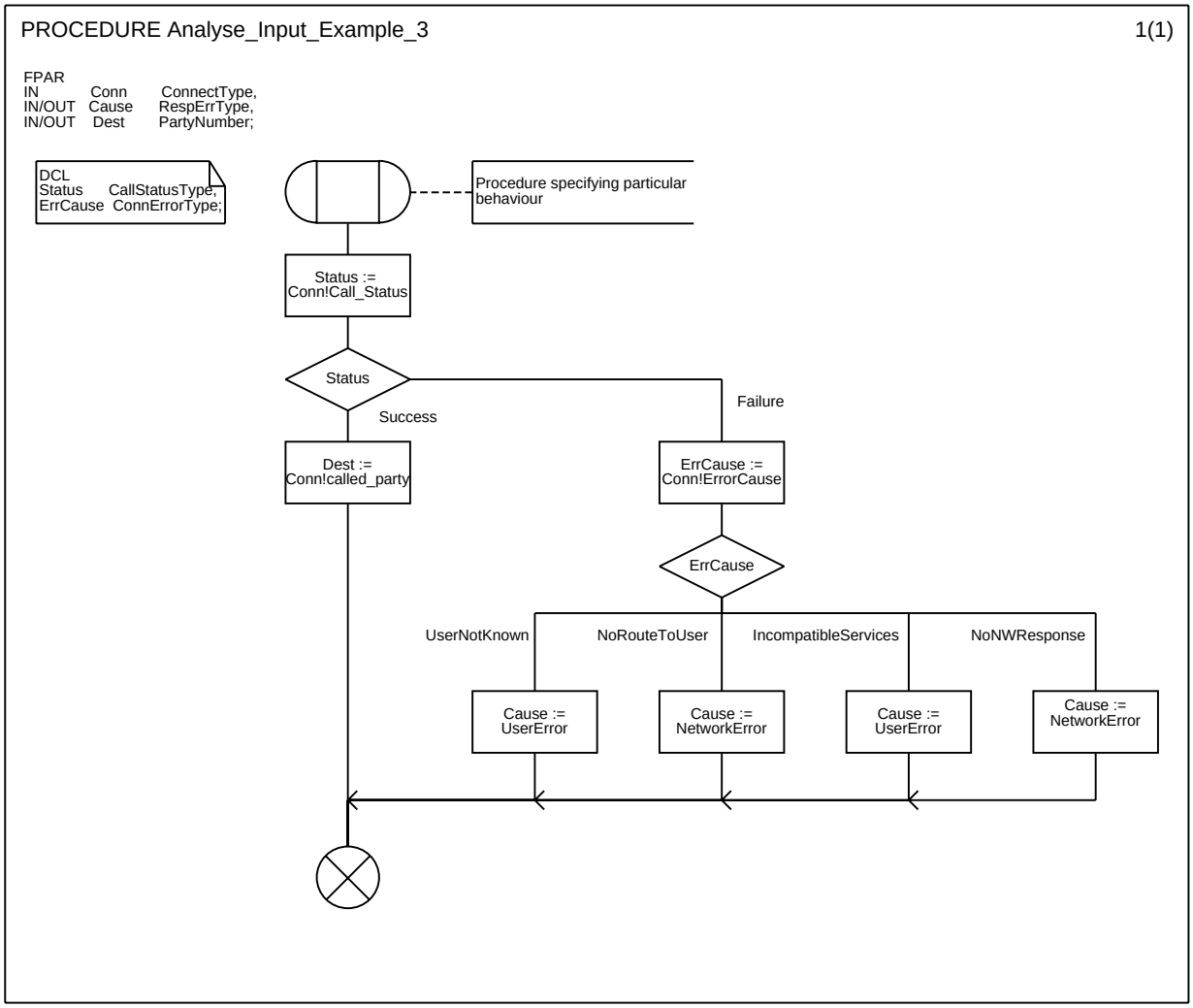


Figure 21: Example detailed procedure

8.1.3 Avoiding side-effects

Each procedure should have a limited and clearly identifiable purpose which should fall into one of the following two categories:

1. Procedures that either analyse something or calculate something from input parameters and return a value that represents the result of the activity. Some programming languages refer to this use of a procedure as a *function*. ***A functional procedure should fulfil its specified role and do nothing that could be considered to be a side-effect***. For example, a procedure that analyses the parameters received with a message should return a value that determines the future behaviour of the calling process. That behaviour may include sending of signals. ***The processing of signals is one of the most important activities shown in the SDL of a protocol standard and should normally be visible in the calling process rather than the called procedure***. Equally so, if the purpose of a procedure is to calculate something, it should do that and nothing else.

2. Procedures that generally do not return any value but have a limited sequence of actions to perform. These actions are worth putting in the procedure provided that the same sequence of actions is repeated in many situations. In this case it may be appropriate that one or more related signals is sent from within a procedure. However, ***it is important that procedures that specify a limited sequence of actions should be given names that reflect as fully as possible the activity performed by a procedure***.

In either case, ***behaviour that could be considered a side-effect to its defined purposes, should not be specified in a procedure***.

The specification of states within procedures obscures the processing of inputs and the overall synchronization of the calling process. Although not generally recommended, it is reasonable in some exceptional cases for a procedure to include the specification of states. Such situations are rare but an example would be a procedure which starts a 500ms timer and excludes all other processing until the timer expires. In this case, a state is necessary in order to receive the timer expiry

In the exceptional case that a procedure includes the specification of one or more states, it is important to ensure that all signals which are not directly processed within the procedure are correctly handled for subsequent processing. This can be accomplished in one of the following ways:

- explicitly receiving all possible input signals in all states in the procedure;

- using the "SAVE all inputs" symbol which ensures that all signals that are not explicitly processed in the state are maintained as inputs until the next state is reached (see the example in Figure 22).

A simple example of a procedure containing a state symbol is shown in Figure 22.

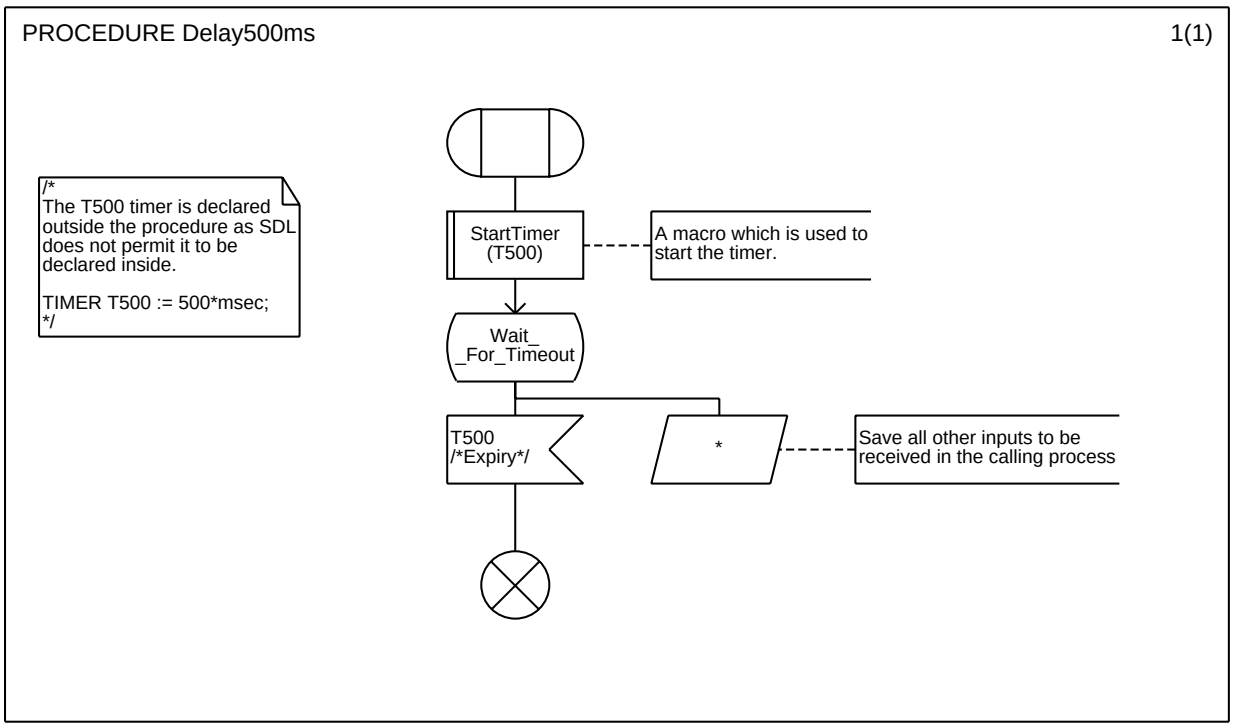


Figure 22: Example of a procedure containing a state

8.1.4 Naming of procedures

Procedure names should follow the naming conventions described in clause 5 and should attempt to clearly reflect the purpose of the procedure without requiring detailed knowledge of the contents of the procedure (e.g., Analyse_SETUP). ***The names of procedures having multiple effects should reflect each intended effect either individually or collectively***. For example, a procedure that builds and then transmits a SETUP message might be named "BuildAndSend_SETUP".

8.2 Operators

In many situations operators represent a viable alternative to procedures. There are, however, some useful differences between them:

- operators can have IN parameters, but not IN/OUT parameters;
- operators are not permitted to have states;
- operators are not permitted to send signals;
- operators are permitted to access only parameters passed into the operator and variables declared inside the operator;
- operator invocations do not have to be preceded with the CALL keyword;
- operators may be used wherever procedures are valid but, unlike procedures, they can also be used in continuous signals.

Thus, operators inherently have many of the desired characteristics of value-returning procedures described in subclause 8.1.3.

One of the simplest but most effective uses of operators is to improve the readability of expressions which contain data elements that need to be extracted from a complex data type. Such an assignment statement may look like this:

```
x := unitdata_ind!called_party_tsi!address!subaddress
```

As well as being long to write, the statement also shows in detail where the particular data element exists in a complex structure and this is probably not relevant in the context of the function of the process.

The example in Figure 23 shows how an operator which will perform the necessary extraction of the data element can be added to an inherited ASN.1 complex data type.

NOTE: Although most data types are specified in protocol standards using the ASN.1 notation defined in ITU-T Recommendation X.680 [7], operators can only be added in an SDL NEWTYPE statement.

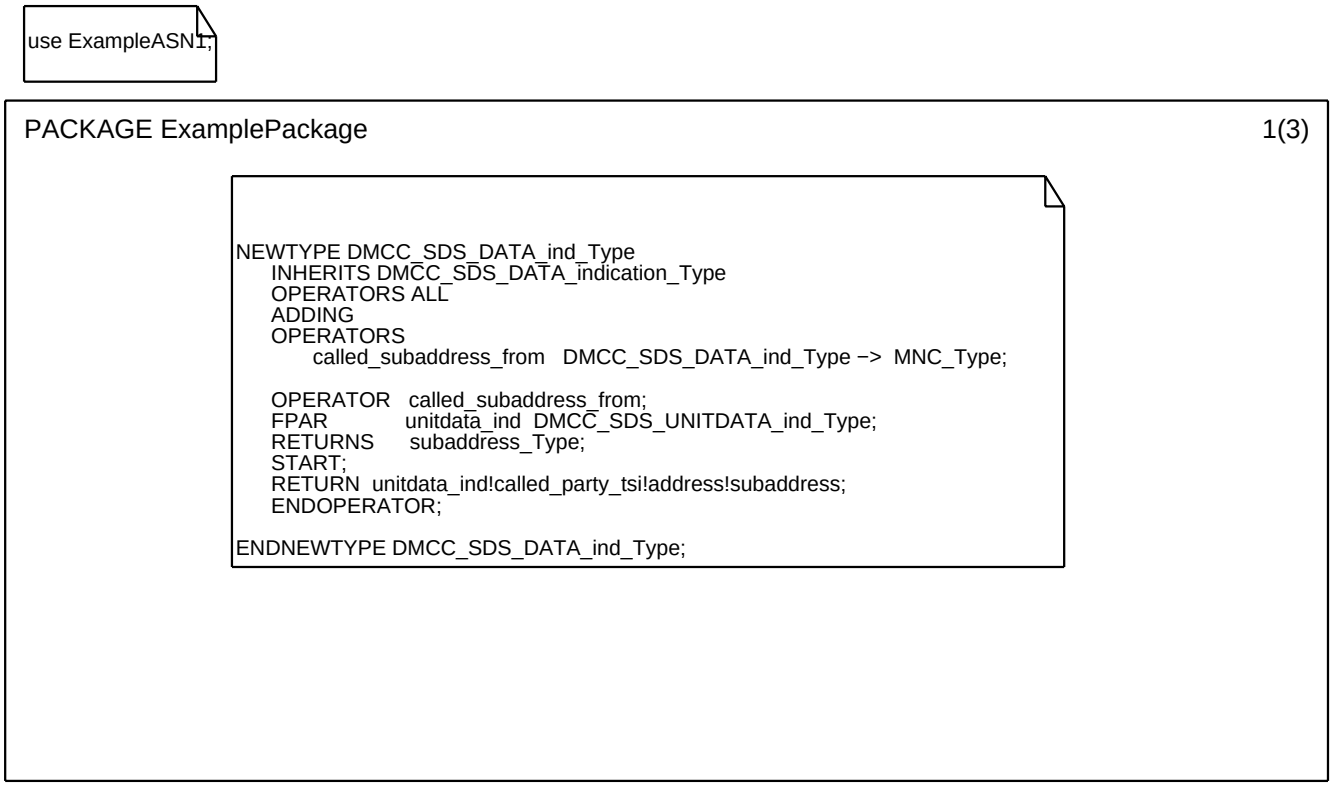


Figure 23: SDL package where new data type containing an operator is specified

An operator is defined as part of the data type to which it belongs and has interface and body specifications similar to those defined for procedures. There is also a signature specification that introduces the operator name and specifies the types of parameters that it receives and returns.

Having defined the operator, the assignment statement can now be re-expressed as:

x := called_subaddress_from(unitdata_ind)

Although this assignment is not much shorter than the original, it now shows the more useful information of what is extracted and where it is extracted from.

The textual syntax of SDL can be used to define simple operators such as the one shown in Figure 23. More ***complex operators should be specified as operator diagrams which are referenced from the relevant data type specification***.

An example of where a complex operator could be very useful is in the management of a circular counter that is permitted to have only a restricted range of values. Each time the value of the counter is incremented, there needs to be a check to determine whether the upper limit has been reached and, if so, counting needs to be restarted from the lowest allowed value. Instead of specifying it repeatedly in process diagrams, an operator can be used for this purpose. Figure 24 shows the necessary data type specification and includes the operator diagram reference. Figure 25 shows the operator diagram itself.

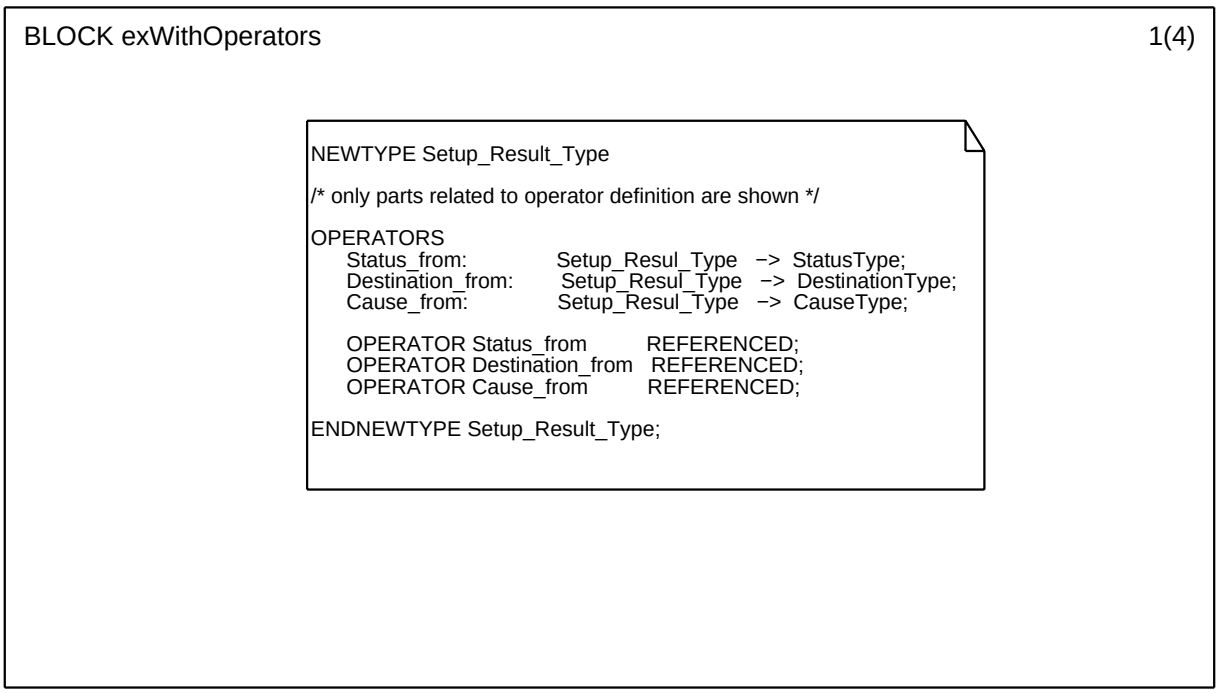


Figure 24: Data type containing signature specification of an increment operator

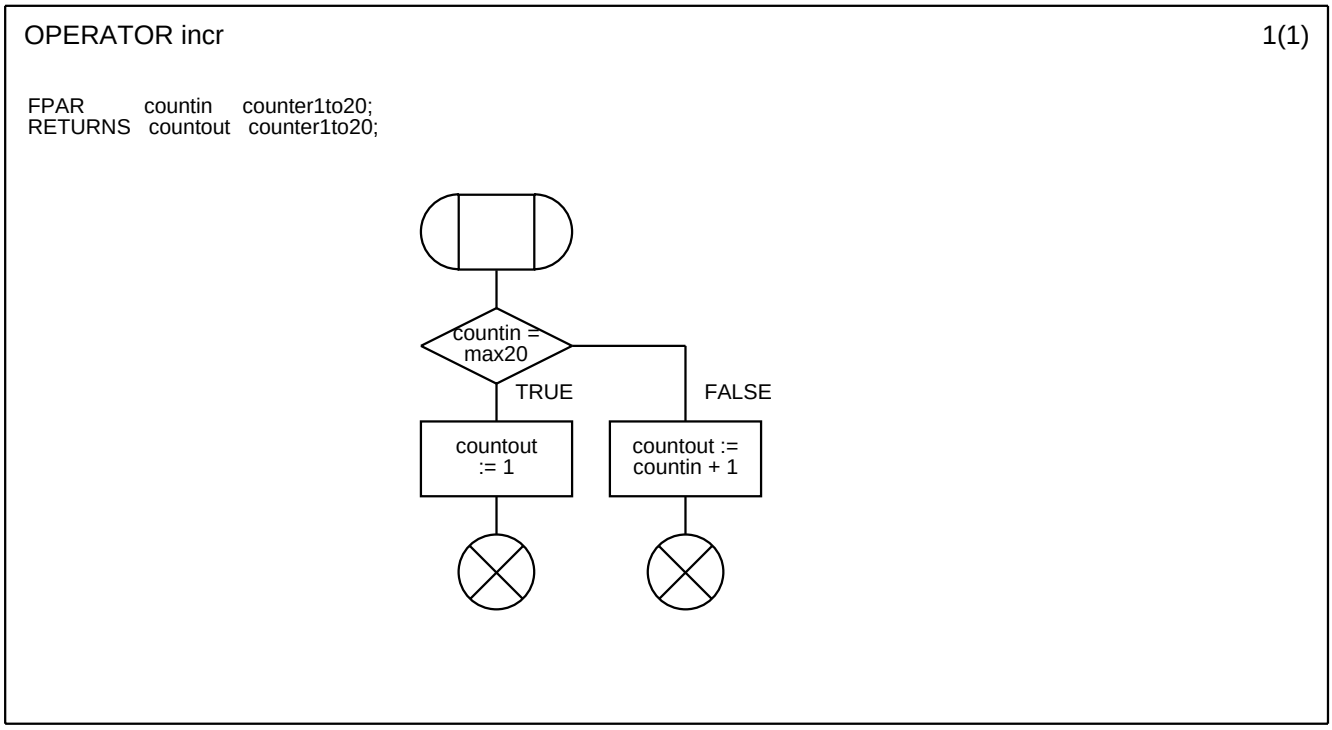


Figure 25: Operator diagram for the increment operator

Figure 26 shows how operators can be used to achieve the same effect as the procedure call shown in Figure 15. Three operators are used to extract status, error cause and destination address information from the *'Setup_Result'* parameter of the 'CONNECTED' message. The intermediate 'Analyse_Input' step is removed and, by choosing names for the operators carefully (*Status_from*, *Cause_from*, and *Destination_from*), the readability of the SDL is improved.

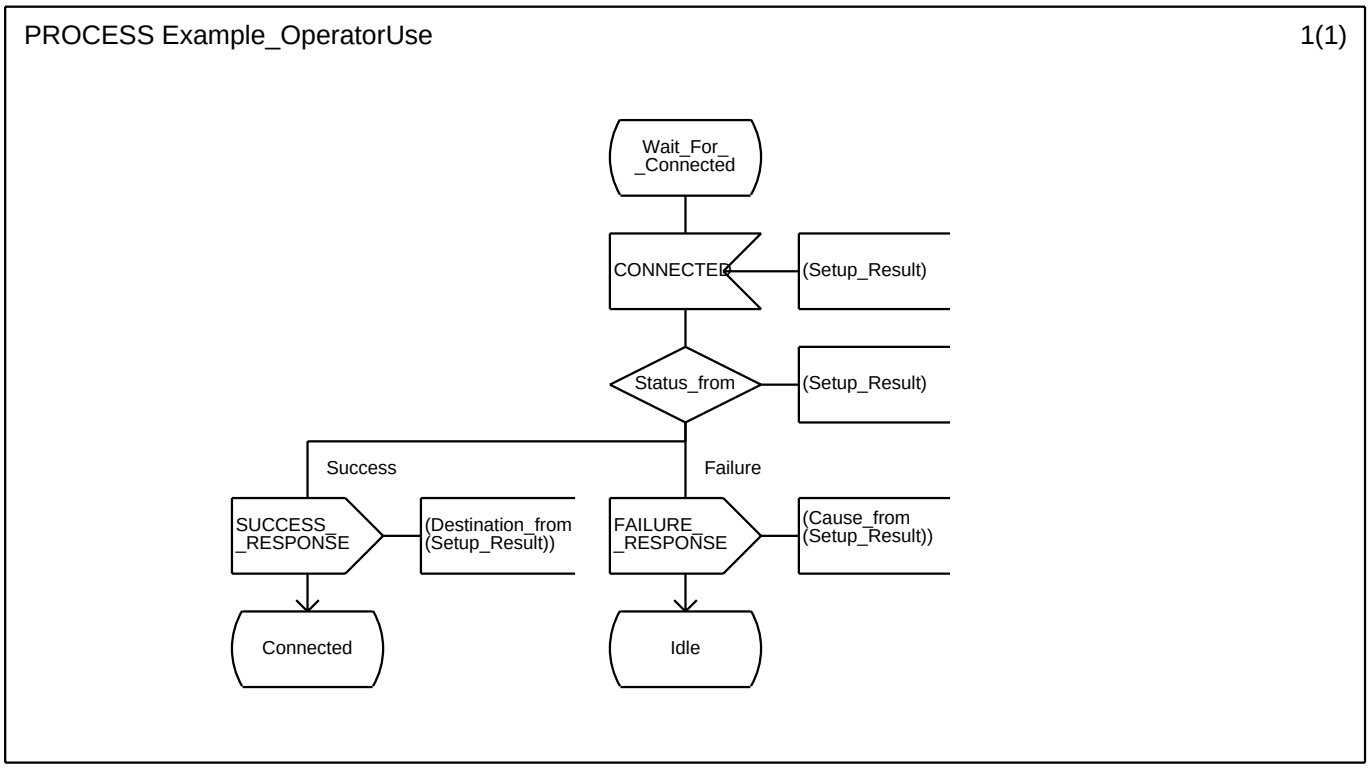


Figure 26: Examples of operator invocation

The operator signature specification, with references to appropriate operator diagrams, for the above example is shown in Figure 27. The operator diagram for *Cause_from* is shown as an example in Figure 28.

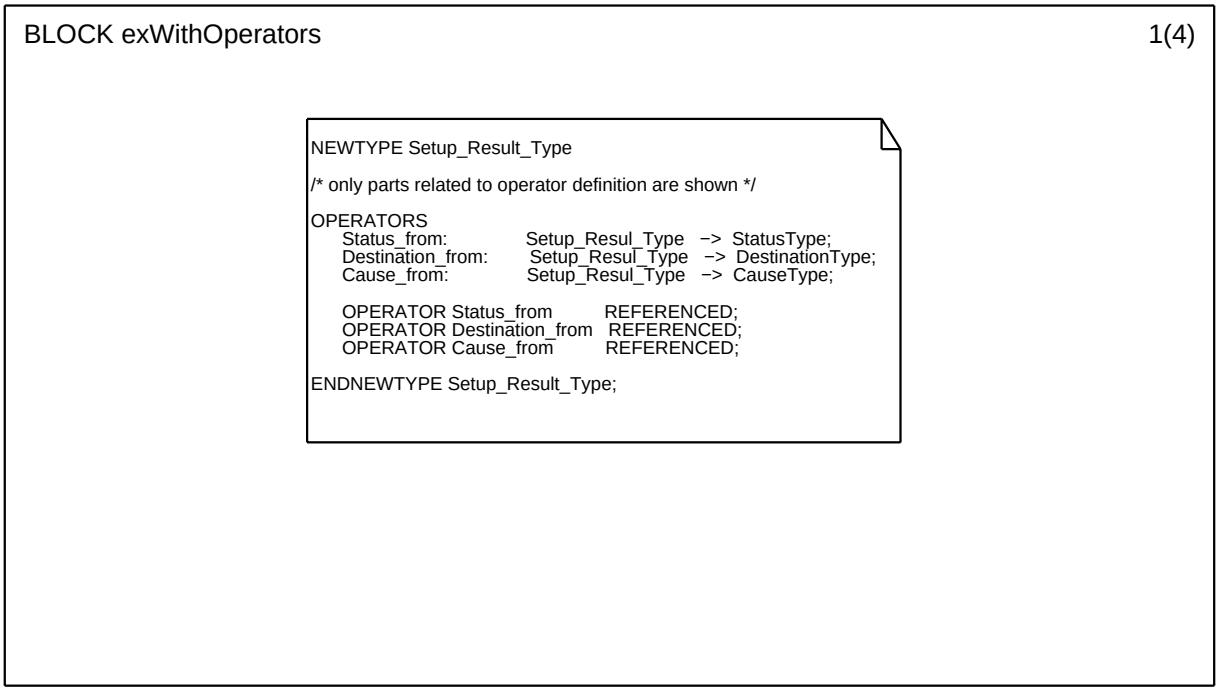


Figure 27: Example of data type definition containing operator signature specification

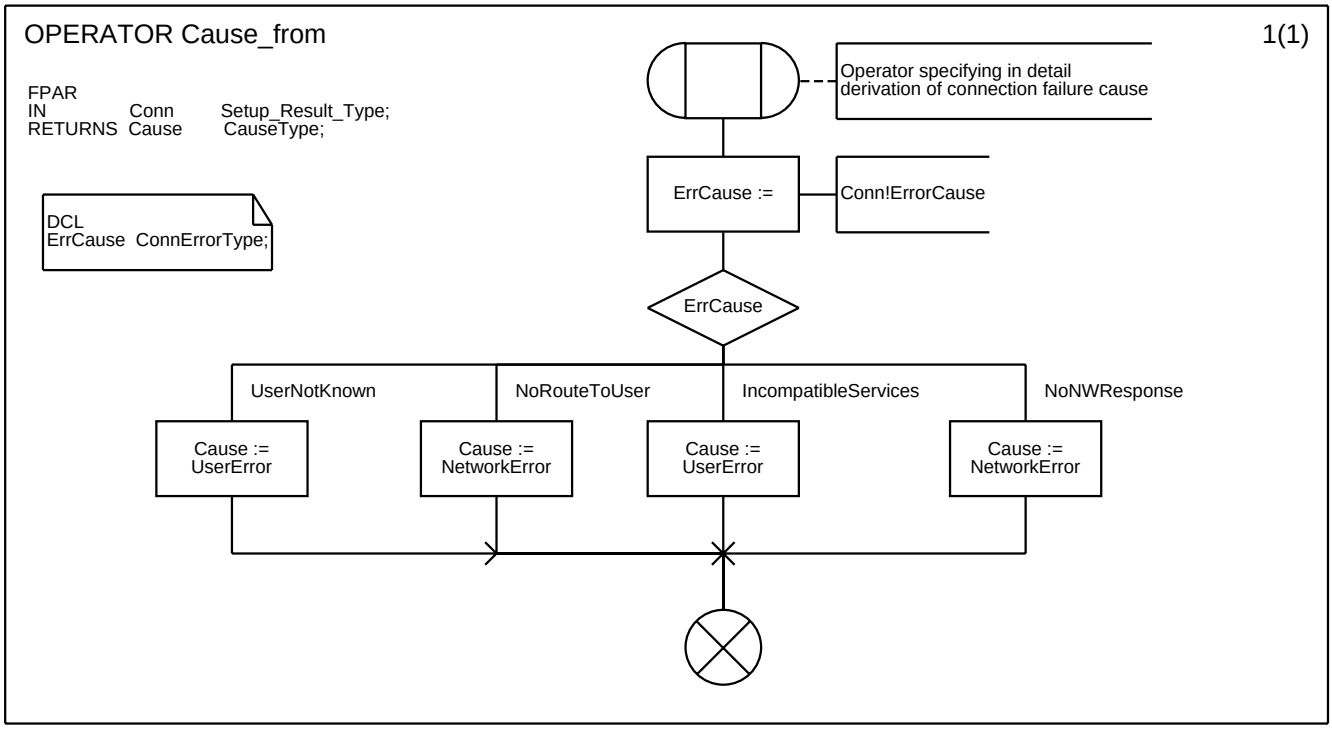


Figure 28: Example of detailed operator diagram

8.3 Using macros

SDL graphical macros can be a very dangerous constructs which, if not used with extreme care, can make a specification difficult to interpret and understand. However, there is one particular circumstance where macros can be used to add clarity and readability to a standard. In most protocol standards, SDL timers are controlled using the informal terms such as "*Start Tn*" and "*Stop Tn*". Unfortunately, SDL uses the keyword SET to start a timer and RESET to stop it. To avoid the use of SET and RESET (which is often misinterpreted to mean "re-start the timer") it is possible to define two macros for this purpose. SDL already uses the keywords START and STOP and so, in Figure 29 and Figure 30 the macros have been named "Start_Timer" and "Stop_Timer".

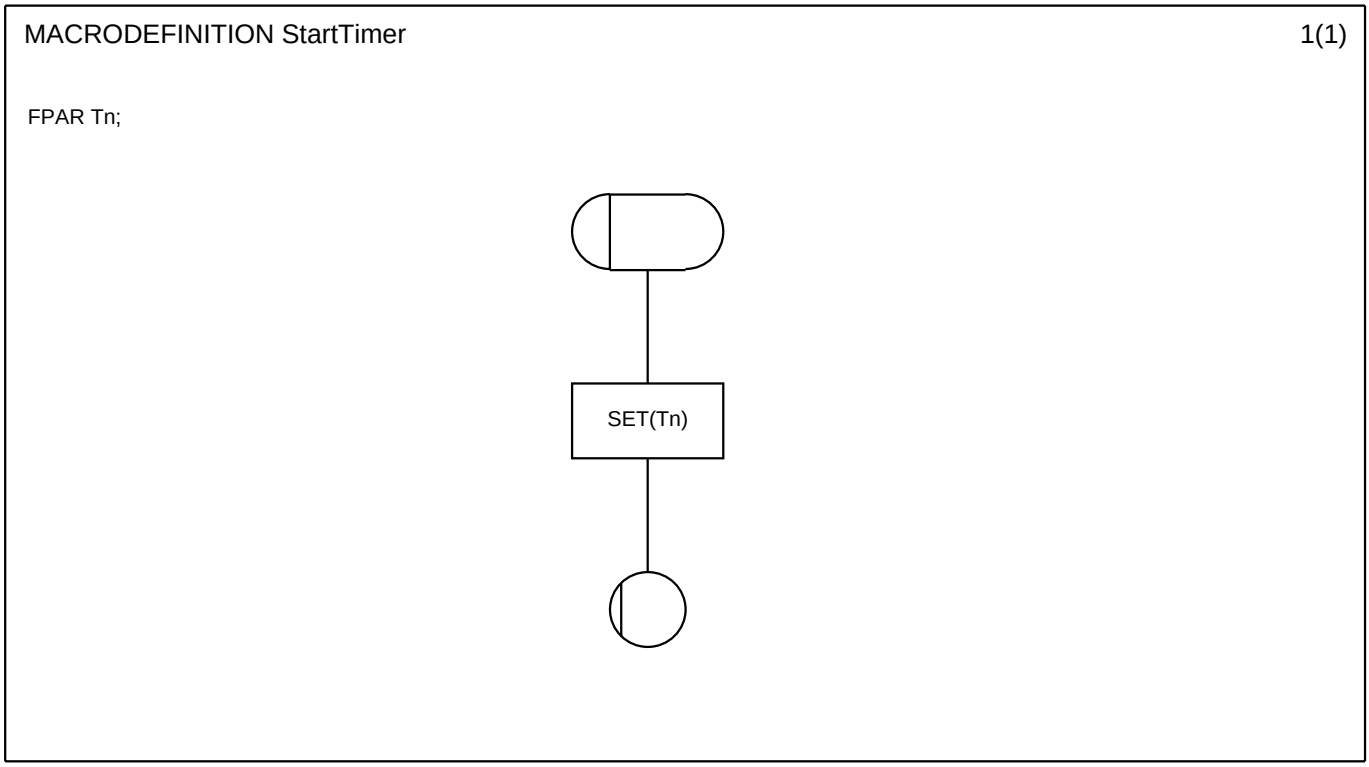


Figure 29: Macro definition for starting a timer

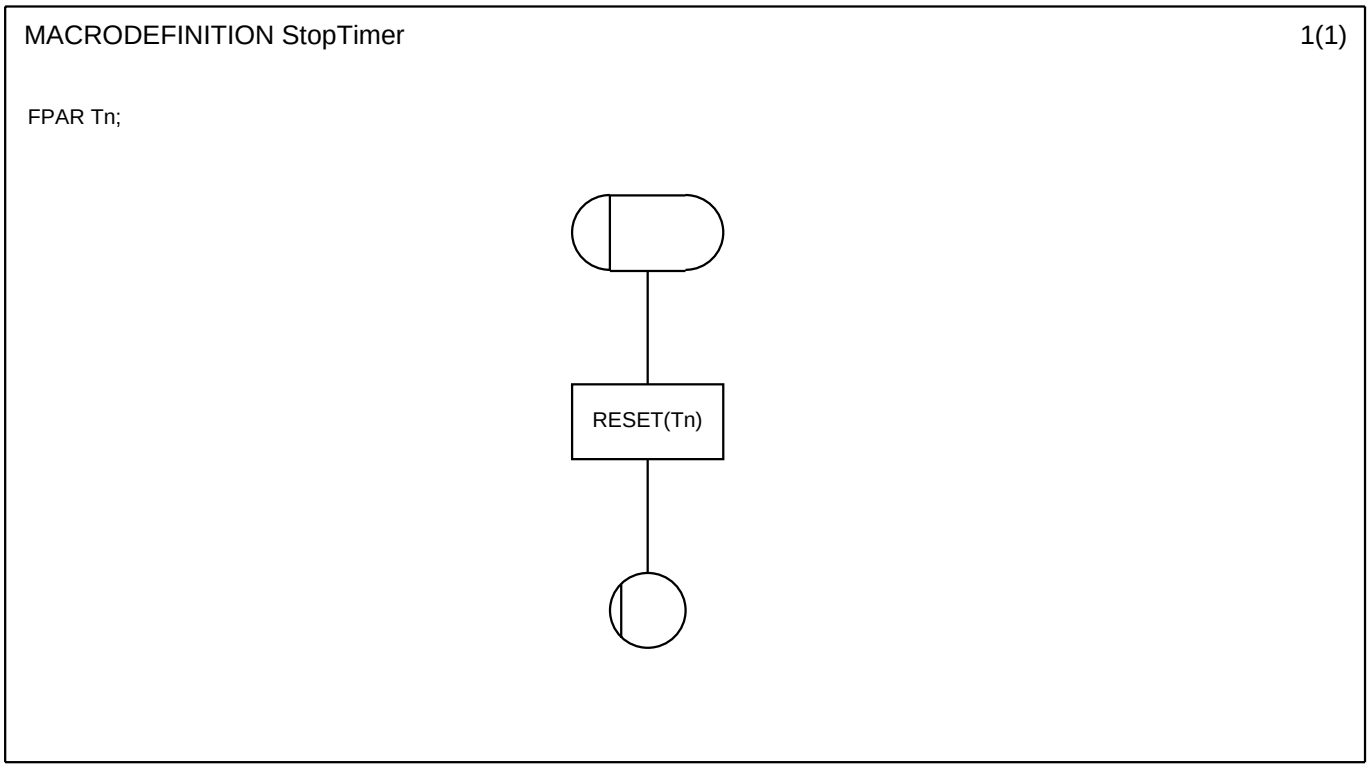


Figure 30: Macro definition for stopping a timer

The example in Figure 31 shows how these macros can be used in practice. Note that the expiry of a timer in SDL is shown as an INPUT symbol simply containing the identifier of the timer so that in this example, the word "Expiry" has been added as a comment.

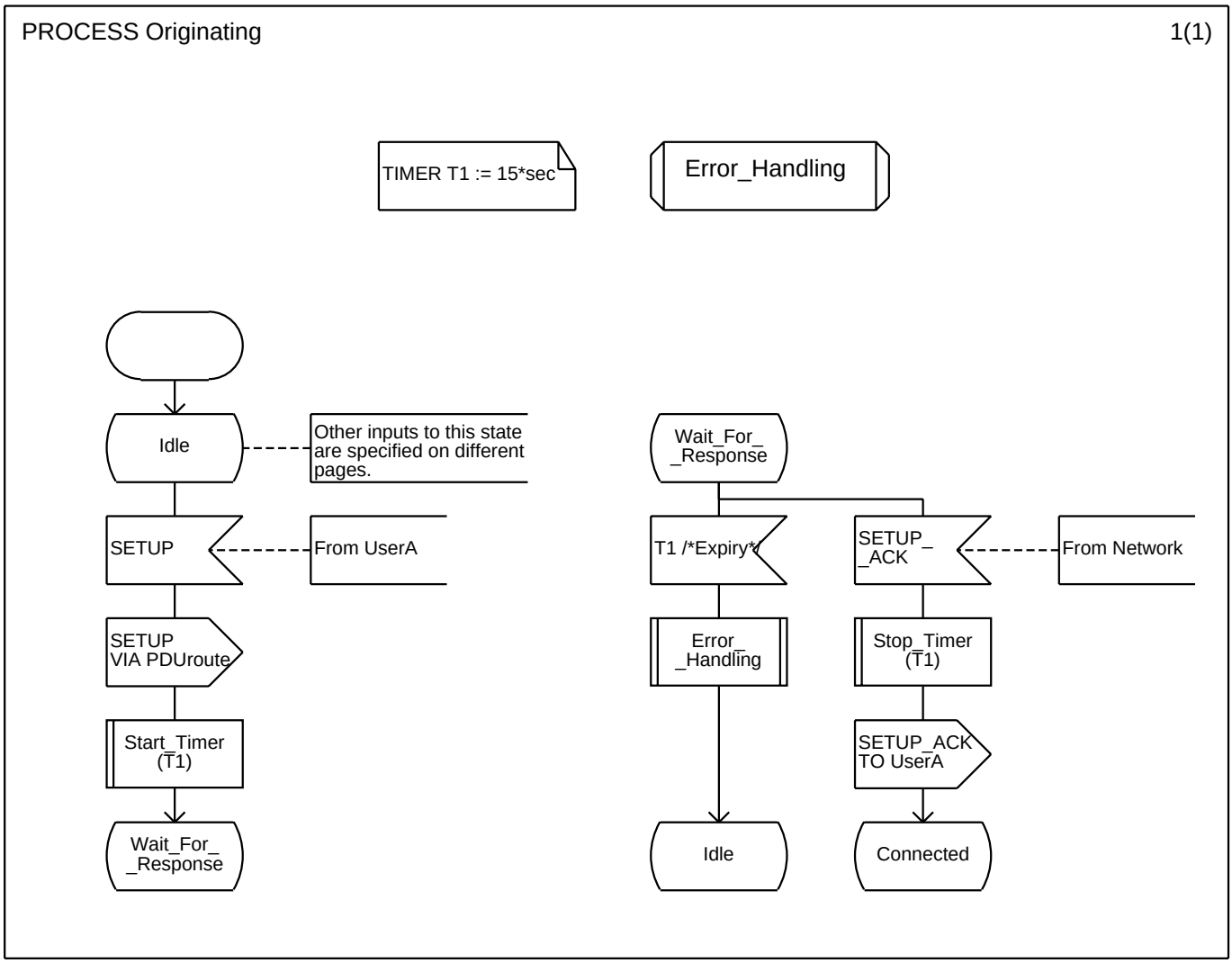


Figure 31: The use of macro definitions

9 Using decisions

Conditional and optional requirements expressed in the textual version of a standard can often be represented in SDL as decisions or options. Decisions are used when the behaviour depends on the actual values or system status at a given time and options are used when the behaviour is fixed by the implementation (or non-implementation) of optional requirements (subclause 9.11).

A decision symbol may contain:

- informal text;
- an expression that evaluates to a value of a certain data type;
- a variable that contains a value of a certain data type;
- a procedure call that returns the value of a certain data type;
- an operator that returns the value of a certain data type.

The use of informal text in decisions is described in subclause 9.3. The remaining cases have the following in common:

- the data type contained in the decision precisely determines the range of values that are acceptable;

- each branch that follows a decision begins with the specification of an answer that determines the range of values for which that particular branch is to be taken. Such values are not permitted to contain expressions that depend on variables or procedure calls.

It is essential that the complete range of values of the data type contained in the decision is covered by ranges of values in the answers without any overlap. In this way it is possible to ensure that a unique execution branch is available for all possible results of a decision. The following errors can occur in the specification of a decision and should be avoided:

- part of the range is not covered by an appropriate answer. This means that there is no path through the decision for such values and so further behaviour is unspecified;

- the ranges of one or more answers overlap. In this case, more than one branch can be taken for a particular value and this would lead to is ambiguity;

- the range of values specified in the answers is larger than the range of values of the data type contained in the decision. As a result, some branches will never be executed. This is likely to be confusing and would hamper readability.

9.1 Naming of identifiers used with decisions

Sensible use of identifiers should ensure that a decision has a clear correspondence to the various alternatives expressed in the text. In addition to following the naming conventions expressed in clause 5 ***identifiers used in decisions should clearly reflect to a reader the 'question' and 'answer' nature of the conditions being expressed.***

9.2 Using decisions to structure a specification

Decisions can be used effectively to divide a specification into separate parts, each dealing with a particular aspect of behaviour. For example, it is quite effective to use a decision to segregate the normal expected behaviour from the exceptional behaviour. This approach can improve the readability of a standard and is illustrated further in 7.3.

It is sometimes the case that a standard needs to specify a complex decision tree based on a number of different parameters. An example of this might be the determination of an error cause based on a message received and the status of some internal data items. In most cases, particularly where the decision process is considered to be normative, it is not possible to simplify the presentation of the decision process by using alternative SDL constructs without losing clarity. Summarizing the decisions in a table before attempting to write the SDL can be helpful. Each decision should then be specified explicitly in the SDL and not hidden in a procedure or operator.

9.3 Use of text strings in decisions

The simplest way of expressing the basis of a decision is to use informal text. This method is often chosen by specifiers for its readability. However, it is prone to errors as it gives no precisely defined relationship between the range of values acceptable in a decision and the range of values expressed in the answers.

In the example shown in Figure 32, the implication is that the question is of Boolean type. Unfortunately, as that is not specified explicitly, other values, such as 'Minor error', could exist as part of the range results. The reader cannot be helped by automatic tools which are unable to detect such problems.

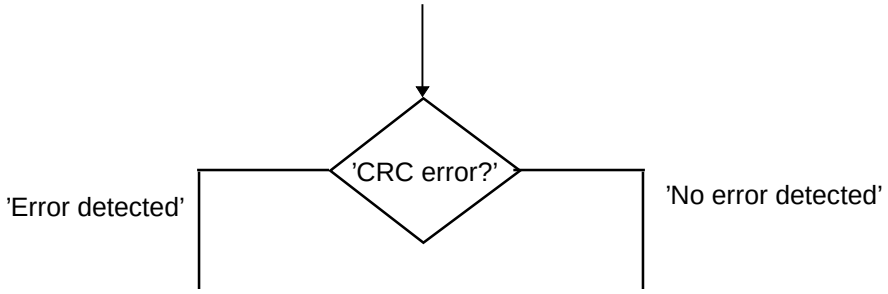


Figure 32: Use of informal text in a decision

NOTE - In a simulation environment the user would be prompted at run-time to choose a particular outcome. While this allows flexibility, it can make simulation cumbersome by requiring excessive interactive input.

It is common in SDL specifications to omit the quotes (") around the text string. This is syntactically incorrect as the quotes should always be present.

Informal text should be used in decision statements with care and should be limited to those cases where the decision is obviously Boolean in nature.

9.4 Use of enumerated types in decisions

The use of enumerated types results in a style which is similar in appearance to the example in Figure 32 but which has the additional and important benefit that a relationship between the question and answers is explicitly and precisely defined. The reader is made aware that there are no more than two possible outcomes. Furthermore, a tool can check that:

- the contents of the decision symbol and the outcomes are compatible;
- the value expressed for each outcome is within the enumerated range;
- that all items in the enumeration have a possible outcome

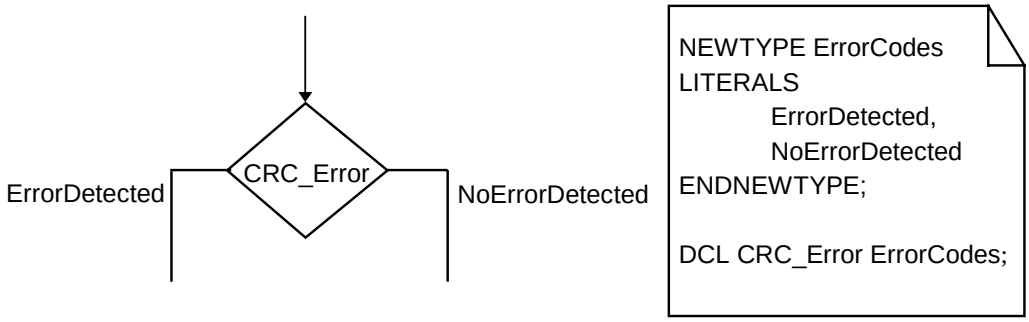


Figure 33: Use of enumerated types in a decision

NOTE: In this example a simulator would take one branch or the other depending on the actual value of *CRC_Error*.

While this approach requires slightly more effort to declare the enumerated types and the associated variables, it produces a specification which is far less prone to error and aids understanding by allowing the grouping of related components such as error codes, service options and status values. ***In most cases, enumerated types rather than text strings should be used to express decisions.***

9.4.1 Use of Else

The use of the SDL built-in value ELSE is useful in completing ranges of outcomes. In the example shown in Figure 34, separate branches are specified for 7200bps and 14400bp while 28800bps and 33600bps are both covered by the ELSE.

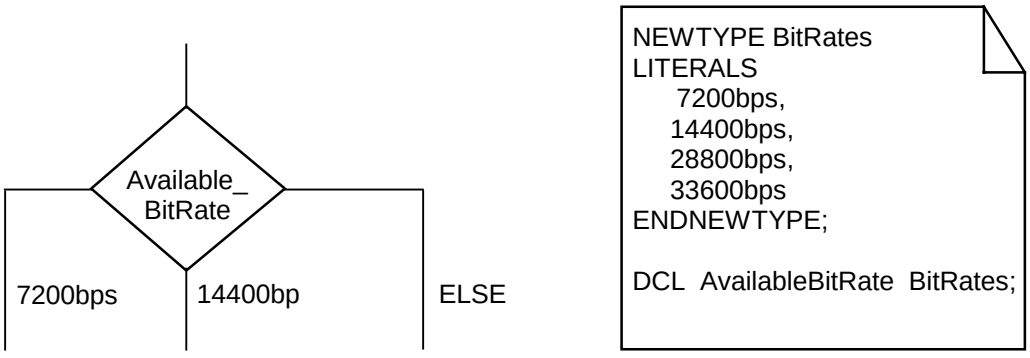


Figure 34: Use of ELSE in a decision

Note that a precise interpretation of the ELSE construct is only possible if the range of values in a decision is defined by a data type (be it enumerated or any other type).

ELSE should be used as a decision outcome value to distinguish between one or more specific outcomes and all other possibilities.

9.5 Using SYNTYPES to limit the range of values in decisions

It is often necessary to limit the range of values a particular data type can have. This is especially important in decisions where ELSE is used since it limits the range of values that lead to an ELSE branch. In most cases, the SDL concept of SYNTYPEs can be used to define a type that is the basically the same as an existing type but which has a limited range. In the following example, the type 'Digit' has all properties of 'Integer' but cannot take values that are less than zero or greater than 9.

SYNTYPE Digit = Integer CONSTANTS 0:9;

Thus, ***SYNTYPE expressions should be used to limit the range of values represented by an ELSE branch in a decision***.

9.6 Use of symbolic names in decision outcomes

In many cases the content of the decision will be a boolean data type, which means that the values of 'True' and 'False' should be given in answers. ***SDL SYNONYMS should be used to define meaningful alternatives to the boolean values of 'True' and 'False' if this aids clarity***. Figure 35 shows examples of the specification of boolean SYNONYMs.

```
SYNONYM  Yes BOOLEAN = TRUE;
SYNONYM  No   BOOLEAN = FALSE;

SYNONYM  Available     BOOLEAN = TRUE;
SYNONYM  NotAvailable BOOLEAN = FALSE;

SYNONYM  Success  BOOLEAN = TRUE;
SYNONYM  Failure     BOOLEAN = FALSE;
```

Figure 35: Examples of the specification of SYNONYMS

9.7 Use of logical expressions in decisions

In some cases, as shown in Figure 36, it is more meaningful to use comparisons to identify the possible outcomes from a decision.

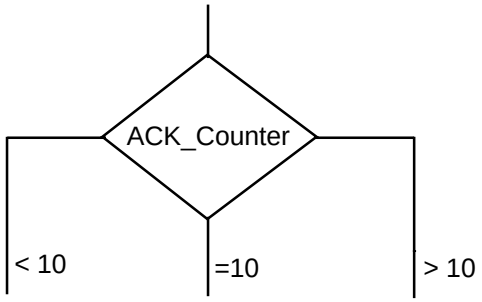


Figure 36: Use of logical expressions in a decision

Although this explicit expression of outcome values is unambiguous, it lacks flexibility. For example, if the value '10' was the maximum value that 'ACK_Counter' should reach and it is used in numerous decisions throughout the specification, it would be very time-consuming to modify all relevant instances of '10' in the event that the requirement for the maximum value of 'ACK_Counter' changes. ***For the purposes of flexibility symbolic names rather than explicit values should be used to express decision outcome conditions***. This approach is shown in Figure 37.

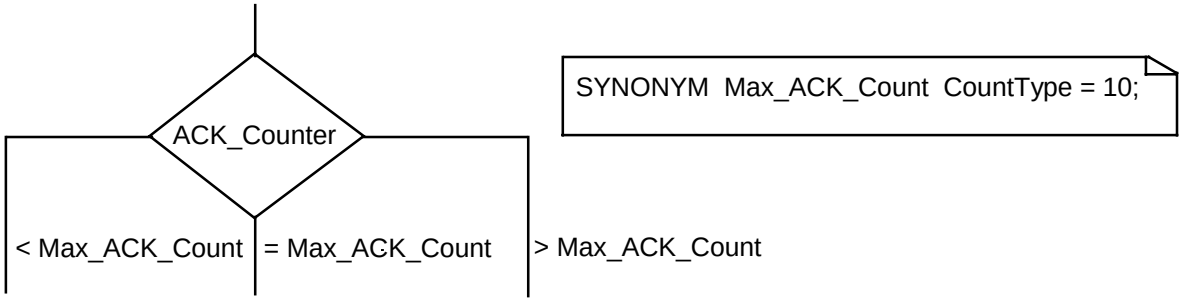


Figure 37: Use of symbolic names rather than explicit values

9.8 Use of Procedures in Decisions

It is possible to use procedures in conjunction with decisions both to simplify the SDL and to improve its syntax without impairing its readability. As an example, the informal description shown in Figure 38 could be re-written in three ways using a procedure with the decision.

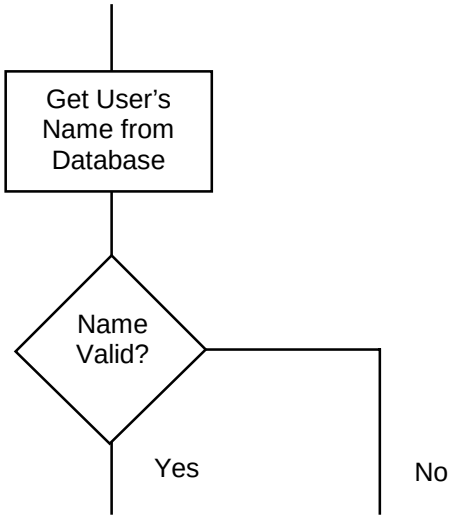


Figure 38: Informal task and decision

The first alternative is to call a value procedure directly from the decision, as in Figure 39. The procedure 'UserName' extracts from the database the user's name associated with 'UserNo'.

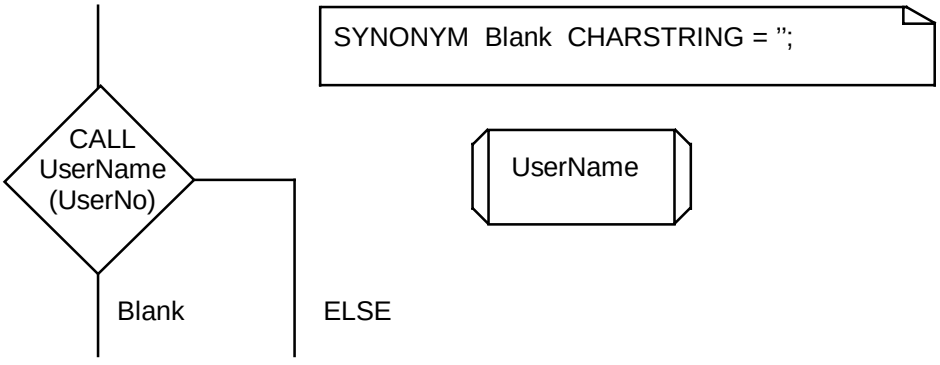


Figure 39: Procedure called from within a decision

The advantages of this method are that it is concise and, in many cases, expresses only those aspects of the specification that are important to the standard.

The disadvantages are that:

- in some instances it is too concise and hides normative requirements in the procedure;
- the CALL keyword is a distraction within the decision symbol;
- the diagrammatic structure is quite different from the original informal SDL.

The second alternative is to assign the result of the value procedure to a variable before making the decision based on the contents of the variable as shown in Figure 40.

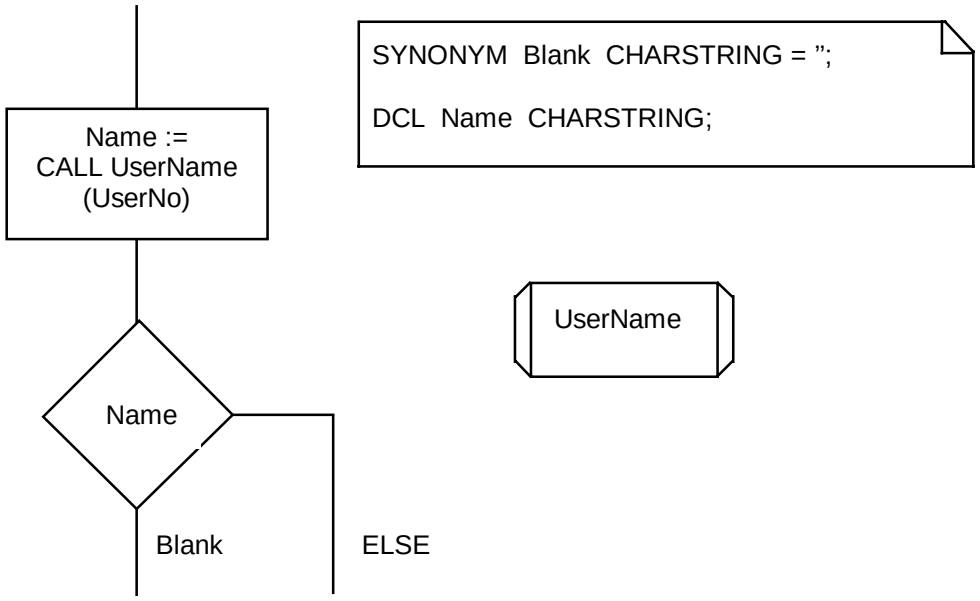


Figure 40: Decision based on a variable assigned from a value procedure

The advantages of this method are that it maintains a similar structure to the original SDL and can make clearer the individual steps involved.

The disadvantages are that an additional variable ('Name') must be specified and the assignment statement is less descriptive than the informal text.

The final alternative is to call a procedure which returns a value as a parameter which is then used as the basis for the subsequent decision, as shown in Figure 41.

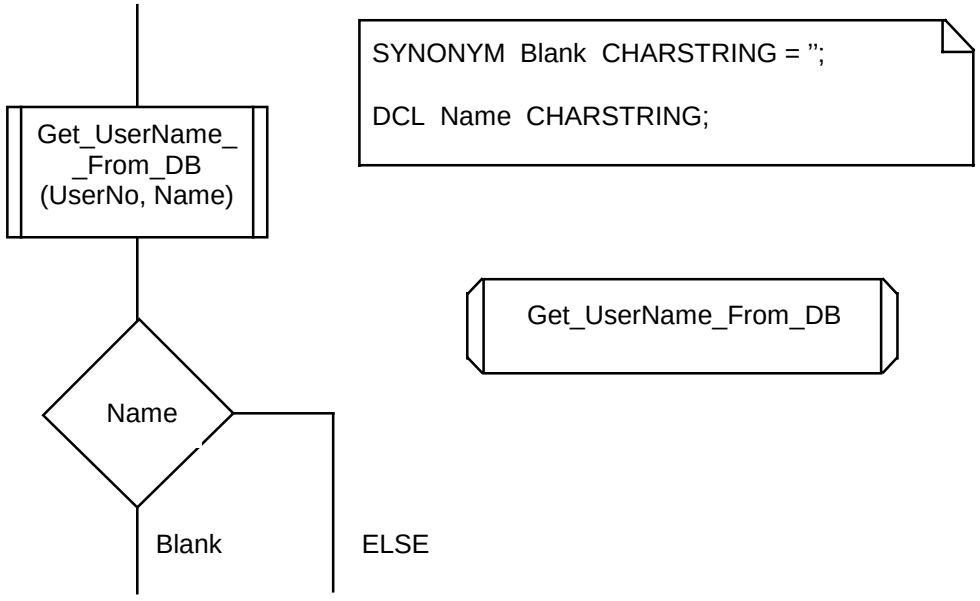


Figure 41: Decision based on a procedure's return parameter

The main advantage of this method is that it maintains almost exactly the appearance of the original informal text.

The disadvantages are the same as those for the second alternative with the additional factor that returning the decision variable in a parameter can mask errors in the specification. As an example, if the procedure 'Get_UserName_From_DB' did not determine and return a value in the 'Name' parameter, this would not be detected by automatic tools and the decision would be based on whatever value had previously been assigned to 'Name'.

All of the three alternatives above are valid methods and it is a matter for the rapporteur to decide which is the most appropriate on a case-by-case basis. Whichever one is selected, ***procedure calls should be used in conjunction with decisions to eliminate the use of informal text***.

9.9 Use of Operators in decisions

In the first two alternatives in subclause 9.8, SDL Operators could have been used in exactly the same way as the procedures except that the distraction of the CALL keyword would have been omitted. ***When a decision is based on a data item that is not directly available to the process, SDL operators should be used rather than value procedures***.

9.10 Use of ANY in decisions

For validation purposes, it may be necessary to re-specify decisions using the non-deterministic ANY symbol. However ***the ANY symbol should not appear in the SDL specifications in standards except where it is included to show the behaviour of an entity (such as a user) that is not the subject of the standard***.

9.11 Use of options rather than decisions

The dynamic nature of decisions is not well suited to the expression of the implementation options which are often to be found in protocol standards. Fortunately, SDL includes a symbol (Figure 42) specifically for this purpose.

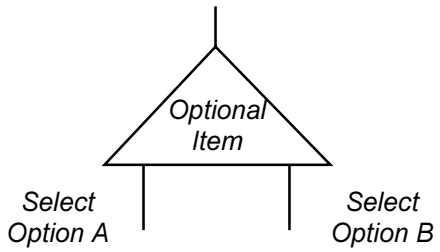


Figure 42: SDL Option symbol

Where mutually exclusive implementation options are to be expressed, the option symbol should be used rather than a decision.

The most effective way of labelling the paths leading from an option symbol is to define appropriate synonyms.

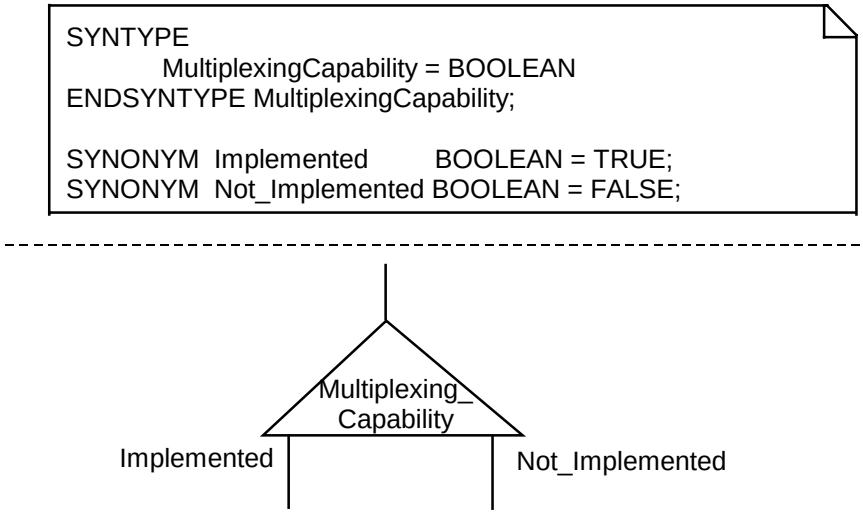


Figure 43: Use of SYNONYMs with options

NOTE: A practical problem can occur with a specification model that has many options and which is to be used for validation purposes. In such cases the 'hardwired' nature of SDL options makes this cumbersome as each new combination will require a new compilation of the executable model. Decisions together with some form of parameterisation would provide a more flexible approach.

10 System Structure, Communication and Addressing

One of the principle aims when using SDL in a descriptive manner is to provide a readable specification that concentrates on describing what the system is supposed to do (requirements) rather than on the detail of how the system is to be implemented. A useful technique for hiding detail at various levels of complexity is the *layering of information* (sometimes called *data hiding*) where components in the model are specified at several layers, with each layer successively containing more detail.

NOTE - The structure and readability of an SDL specification with respect to its graphical layout is considered in clause 7 and the use of data for signals in clause 11.

10.1 System structure

SDL allows the layered specification of systems such as protocols or services in a hierarchical manner through the use of *system*, *blocks* and *processes*. The system and blocks define the static architecture of the system. The processes contained in a block define its dynamic behaviour.

The SDL system and block structuring give an unambiguous description of the system architecture. It is usual that architectural aspects are described elsewhere in a standard (or even in other standards) often using non-SDL figures. If this is the case then ***the SDL version of the architecture of a protocol or service should be consistent with and complementary to other (non-SDL) descriptive diagrams***. This is particularly important in relation to naming, which facilitates the easy identification of system components. In addition, ***comments should be used to convey to the reader the relationship of the SDL architecture to the relevant non-SDL parts of the standard***. If the structure of the system is specified in SDL, ***informal drawings that duplicate structural information given by the SDL diagrams should not be used***. This may mean including SDL system diagrams in the parts of the document where structure and architecture are discussed.

The major advantage of SDL structure diagrams is that their meaning is well defined, so that the document does not rely on intuitive understanding of an informal drawing or introduce an explanation of the notation used. Of course, many issues (such as physical arrangements) cannot be described in SDL, and other well defined notations may also be used.

An SDL specification is incomplete if it includes behaviour descriptions in process diagrams but does not include the associated system and block structures. Even in the case of a simple protocol or service, ***the SDL specification within a standard should comprise one system composed of at least one block which in turn is composed of at least one process***. This is not simply a case of 'getting the SDL right' for the sake of it. The SDL architecture provides useful information for the reader such as what entities and communication paths exist within the system. The communication paths have an important role

in the addressing of messages from one behavioural part to another. ***SDL should be used to show the structure of a system as well as its behaviour***.

NOTE SDL blocks and processes define the functional partitioning of the system. Using SDL does not imply that a real system need implement a standard exactly as defined by the SDL, only that the implementation should exhibit external behaviour over the normative interfaces that is equivalent to the behaviour defined by the SDL model.

In a complex standard it is possible that the SDL description only covers part of the system. It may also be necessary to include sub-structuring that is only implied in the text but which is needed to give a coherent and complete SDL model. It is not possible to give strict guidelines on how to structure a specification as this will depend on the subject matter of the standard. However, although the careful use of sub-structures can make a complex specification easier to understand, the overuse of sub-structures can render them unreadable. ***SDL sub-structuring should be used to simplify complex SDL models but should not be used excessively.***

10.2 Minimising the SDL model

The example in Figure 44 shows a situation where there are a large number of identical user terminals communicating with one of several identical local concentrators which are all connected to a single common network.

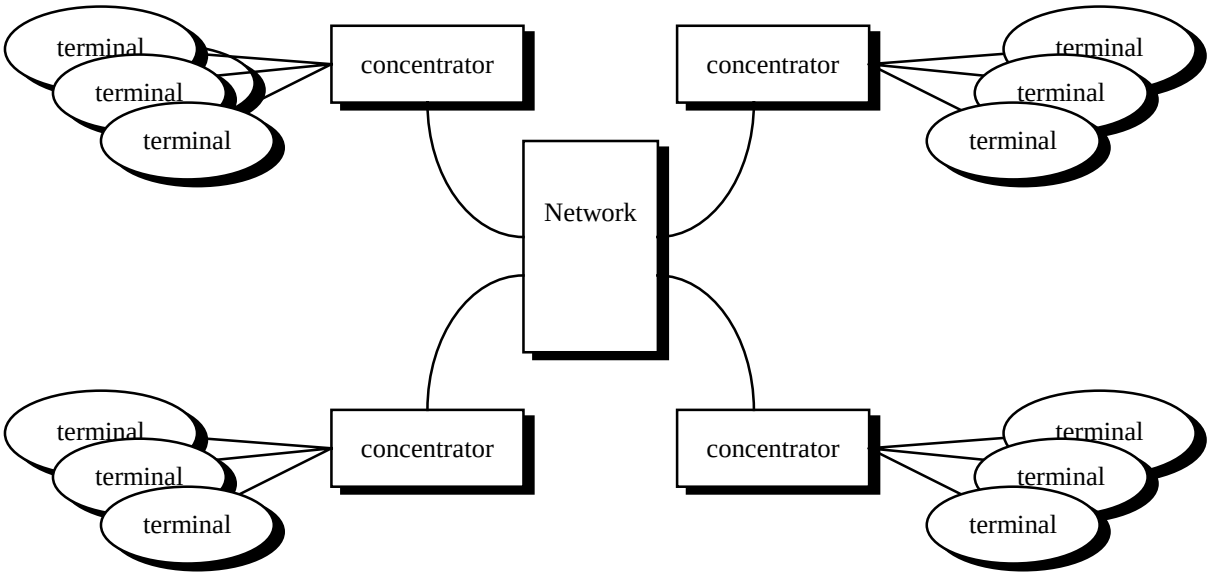


Figure 44: A hypothetical network

Since the terminals all have the same behaviour, it would be possible to describe the system by providing a single description for a terminal that is replicated several times. Similarly the concentrators could be replicated and the corresponding SDL model for an implementation might be as shown in Figure 45. In general, this approach is perfectly acceptable for the specification of an operational system but is unnecessarily complex for describing protocols and services in standards. What needs to be captured in a standard is the minimum that implementations should conform to, and a standard needs to make clear the role of each entity involved.

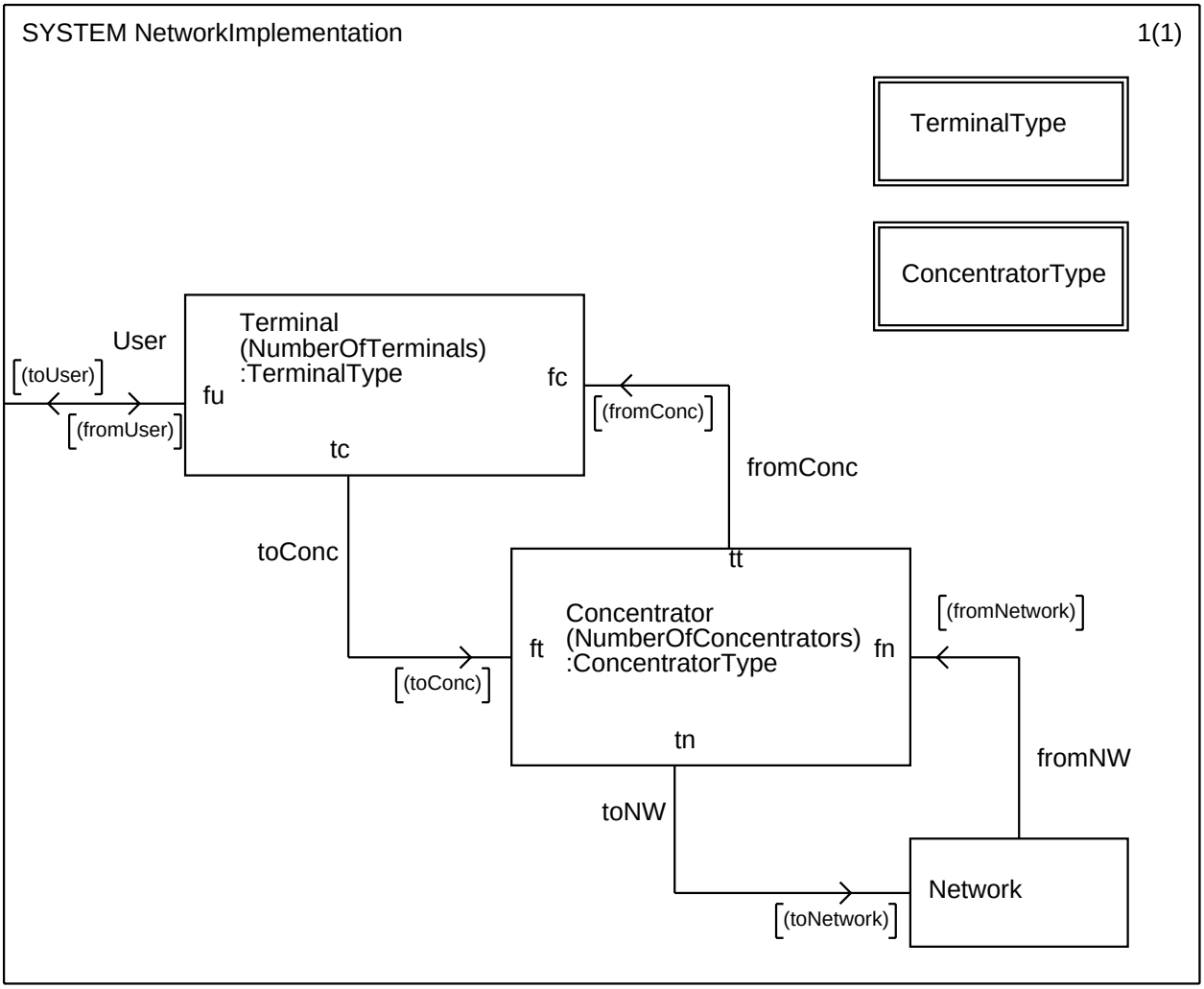


Figure 45: An SDL system model appropriate for implementation of the network in Figure 44

In the example, it would be sufficient to describe the protocol in terms of an origination terminal, a destination terminal, an origination concentrator, a destination concentrator and the network as shown in Figure 46. Each block represents a particular role and the unnecessary complexity of multiple instances shown in Figure 45 is removed. ***Multiple instances of SDL blocks and processes should be avoided if possible***.

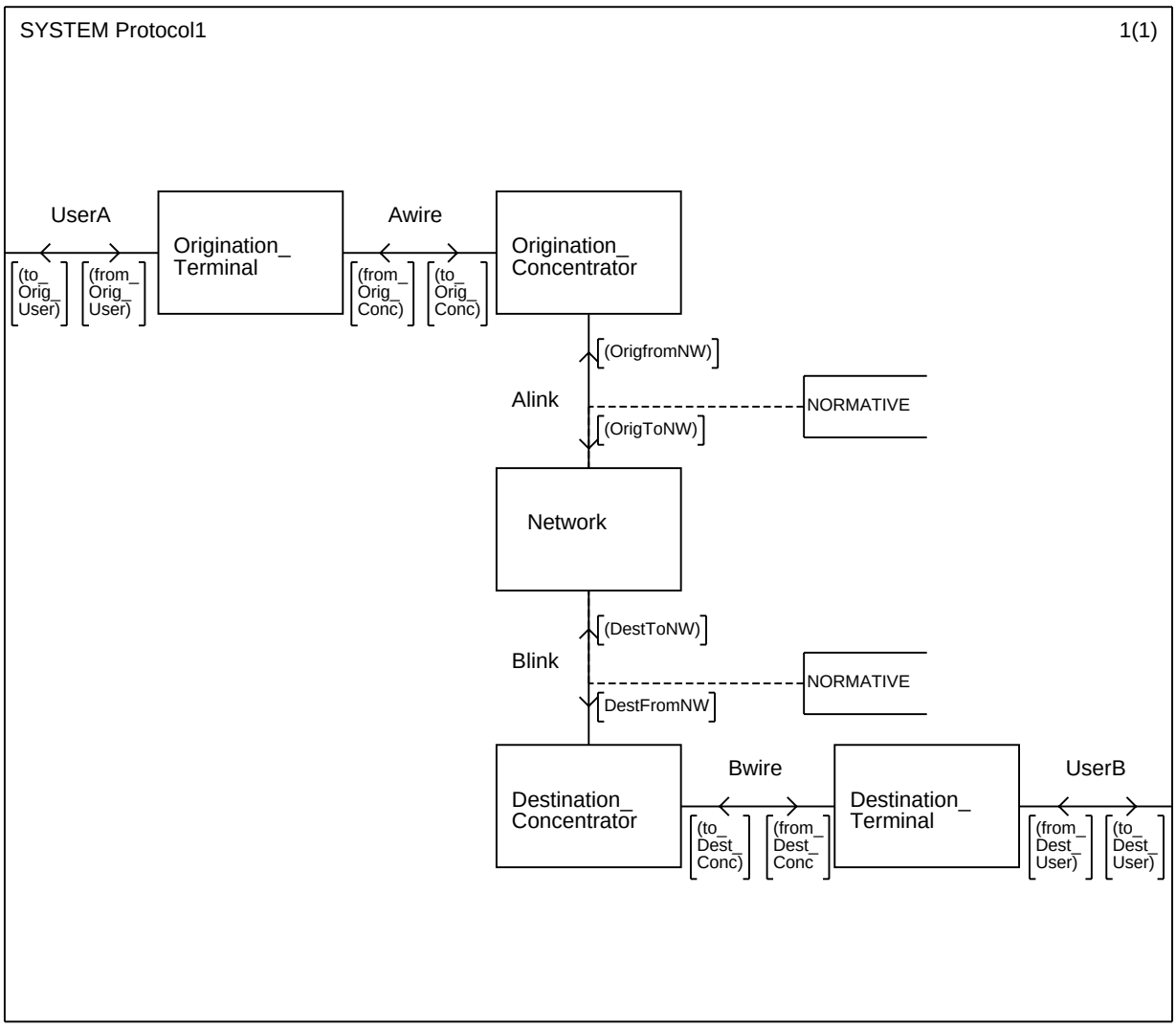


Figure 46: A simplified SDL system model for the network in Figure 44

Sometimes informative blocks and processes (such as terminal in Figure 46) are needed to aid the understanding of a standard, and to describe the behaviour of entities surrounding the functions which are the subject of the standard. If the terminal and network behaviour is not needed for the concentrator-to-concentrator example, an SDL system such as Figure 47 with only the different end functions can be used. ***Informative blocks or processes that are not needed to aid understanding should be omitted***, because such detail will obscure the minimum requirements expressed by the standard.

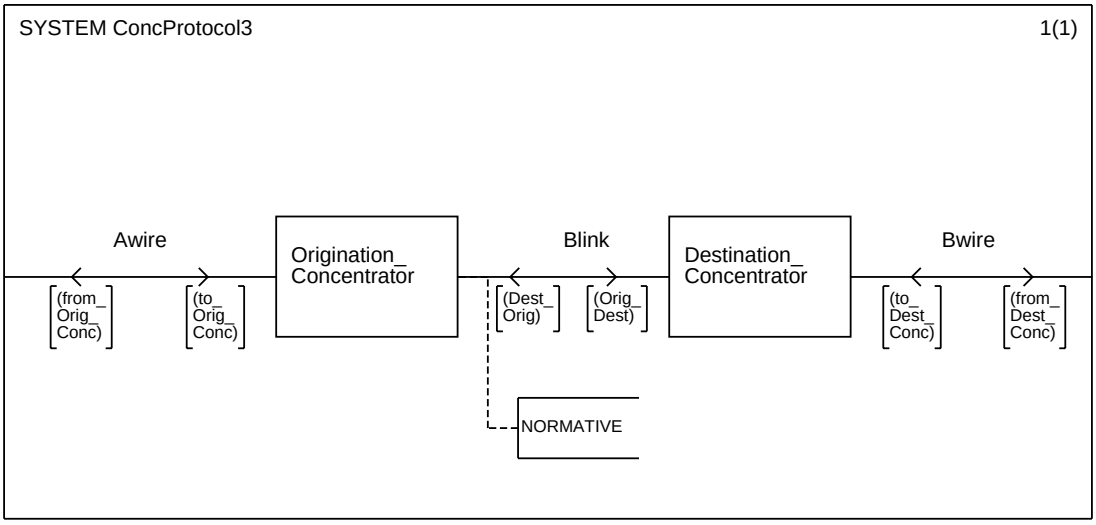


Figure 47: A minimal SDL model for the concentrator protocol standard example (distinct ends)

Protocols can be modelled effectively by showing the functionality of the ends separately as shown in Figure 47. This has the advantage that the description can be simplified so that only the functionality essential to the protocol is defined.

10.3 Avoiding repetition by using SDL types

In some specifications, there may be structure and behaviour that is replicated in more than one block or process. To avoid repetition, ***if the same block or process is required at more than one place within an SDL specification, a BLOCK TYPE or PROCESS TYPE should be defined from which instances can be derived.***

10.3.1 Defining the same behaviour at both ends of a protocol

The use of SDL types is particularly useful for standards that specify the behaviour of both ends (such as origination and destination) of a protocol communication as a single, multi-purpose entity as in Figure 48. With this approach, the function of each end of the protocol is not so distinctly separated but actual functional behaviour is specified only once (in the BLOCK TYPE Concentrator in the example).

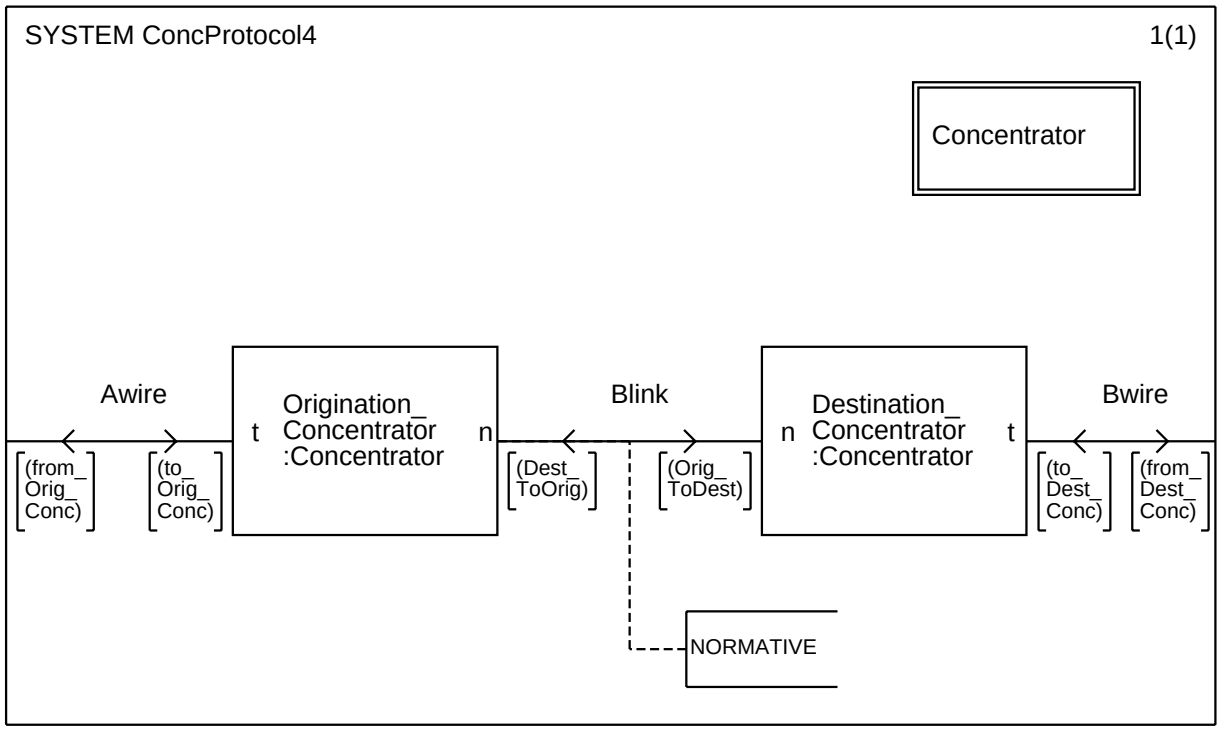


Figure 48: A minimal SDL model for the example where the same function is used at each end

10.3.2 Static instances to represent repeated functionality

In some cases, a standard may suggest that process instances need to be dynamically created. Dynamic creation of entities usually adds unnecessary complexity in the addressing of entities and should only be used in the (rare) occasions that it is essential. If, for example, there is a multi-link concentrator standard then one origination concentrator and two destination, as shown in Figure 49, may be sufficient. In this case, it is appropriate to use the BLOCK TYPE DestConc because the two destination concentrators have the same functionality. ***Wherever possible, a minimal number of static instances should be used instead of dynamically created SDL processes.***

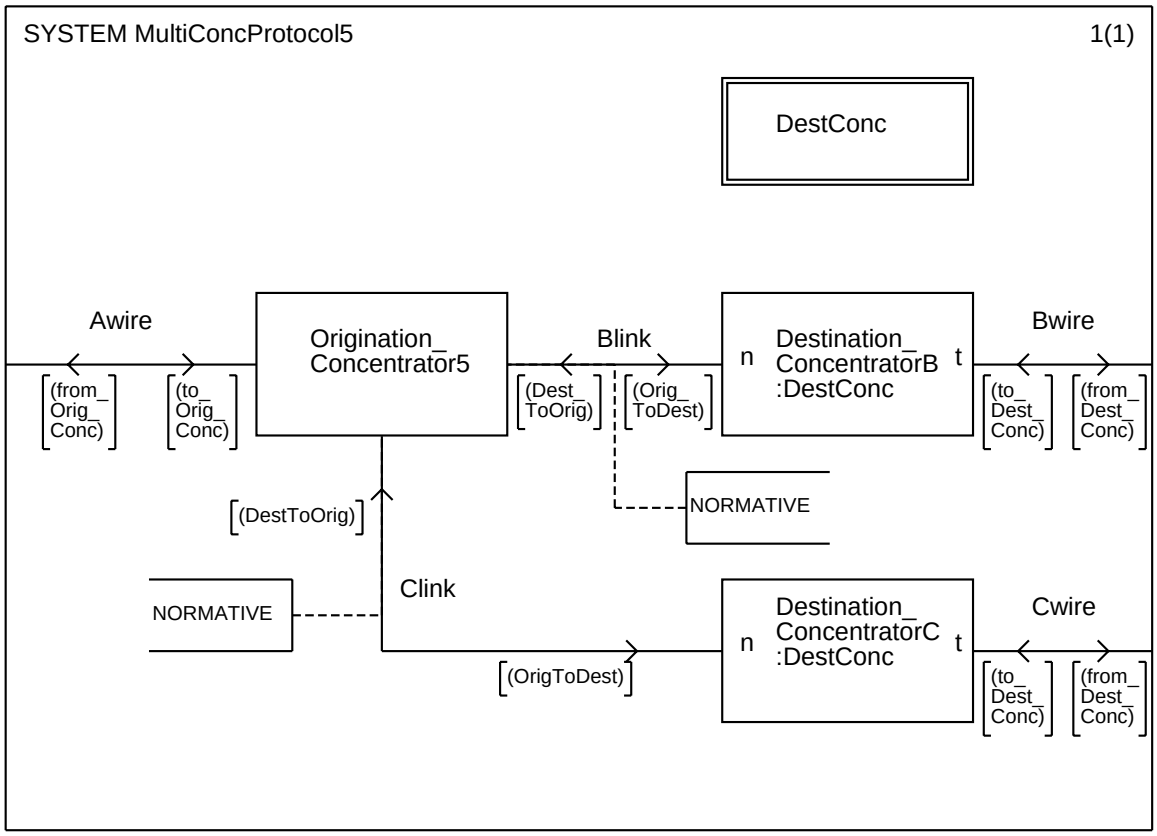


Figure 49: Static SDL model for a multi-link scenario.

10.4 Communication and Addressing

Communication between blocks and with the environment in an SDL system is effected by using *signals* on *channels.* Within a block, communication should be by *signals* on *signal routes.* A communication path is either a channel or a signal route. In simple models there will only be one process instance that a signal can reach and no further addressing is needed.

At least one channel should represent the normative interface(s) of the system being specified and ***all normative channels (interfaces) should be clearly marked as being normative (using a comments box)***, with the assumption that channels not so marked are informative and that they have been introduced into the SDL for clarity and completeness only.

SDL processes are concurrent so it is possible that signals from different processes on the same communication path could be interleaved. If there are two different paths from a sending process to a receiving process, it is possible for messages to arrive in a different order from the order in which they were sent. To avoid this, ***there should be no more than one communication path specified in each direction between one entity and another.*** This makes the communication clearer, and also avoids the possibility of a signal sent on one path overtaking a signal sent on another path.

Although SDL supports other forms of communication (remote procedures and import/export of data), it is better to use these only in exceptional cases, for example where complex internal signal interchanges may be reduced by using remote procedures. These constructs imply that the calling process waits and passes control to the called process. Such a mechanism cannot be supported easily across a normative interface. ***Remote procedures and import/export to exchange information between blocks and processes should not be used***.

10.4.1 Indicating the use of signals in inputs and outputs

A signal instance sent directly from one SDL process to another must have the same name at both ends of the communication. To indicate the different use of signals in inputs and outputs (for example a setup considered as a request at the sending side, and as an indication at the receiving side), the following approaches may be used:

1) giving the signal a composite name (for example, SetupReqInd, SetupRespConf);

2) a context dependent suffix attached to the signal name as a comment (see Figure 50).;

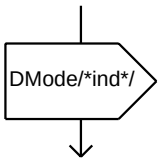


Figure 50: A comment used as a signal name suffix.

10.4.2 Use of SIGNALLIST

Usually there are too many signal names for them all to be listed with the channel or route, so it is better to group related signals together into a named signallist. This also has advantages that the same list can be used in several places. For example, in Figure 49 the list *OrigtoDest* is used twice. This is defined at the system level as:

```
SIGNALLIST OrigtoDest = setupreq, releasereq, datareq, (failures);
```

where *failures* is another signallist - denoted by the parentheses around the name.

To further aid readability, ***signallists should be used to logically group signals on a particular channel*** rather than listing all signals explicitly.

Communication paths show the links between sending and receiving entities. The list of signals conveyed in each direction is associated with the direction arrow on the path. In current SDL these lists are optional if they can be derived from other information but, for clarity, ***all communication paths (SDL channels and signal routes) should be shown with the associated signals or signalists.*** This provides the information where it is needed by the reader.

10.4.3 Directing messages to the right process

SDL allows the specification of a communication path or recipient process to be part of an output. Although there is often no ambiguity as the signal can only take one path to one process, adding this information can make it easier to understand the system (see examples in Figure 51). The TO construct can also be used in some cases to identify a process but, in the example, a comment has been used to clarify that route Conc is connected to a concentrator. The TO construct cannot be used in this particular case because neither is the PId value known, nor is the process name visible. When there is more than one possible recipient of an output, TO or VIA *must* be used in order to be unambiguous . ***TO or VIA should be used in an output symbol to indicate the recipient clearly***.

When a process sends a signal that it can also receive as an input, it is essential to use TO or VIA to avoid the possibility (unless intentional) that the sending process receives the signal. This situation is common for signals that are "passed on" to another process.

Figure 51: Examples of the use of TO and VIA.

SDL also provides a method for directing reply signals using the TO construct and the PId value of the sender. If the reply is generated before any other signal is received, TO SENDER can be attached to the output statement. If, however, the reply has to be sent after receiving subsequent signals, then the SENDER value needs to be stored in a PId variable so that it can be used

later in an output. It is always safer to use this approach rather than TO SENDER because some SDL constructs (such as remote procedures) implicitly change the SENDER value. Thus for Figure 49, an origination concentrator can reply to either of the destination concentrators by an output such as in Figure 52.

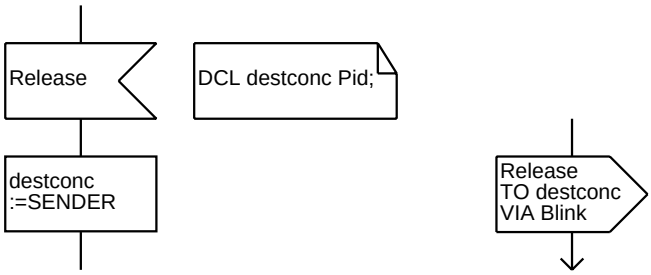


Figure 52: Replying to a sender.

Where communication is with the environment, any differentiation between entities in the environment should be handled by the identity or content of signals, or the identity of channels, rather than use of the TO mechanism.

10.4.4 Differentiating messages

The only way that one message can be distinguished from another before it is received in an input is by its signal name. It is not possible to selectively receive a signal according to its content or the sender or the communication path. When a process reaches a state waiting for a stimulus (a signal or timer), those stimuli which can trigger a transition and those which are saved are distinguished by name only.

NOTE: If a specific signal can be received from several processes, it is not possible to selectively receive it from one source. The sending process identity can be determined by examining the SENDER value, but this does not enable the name of the sending process or block definition to be (easily) determined.

To determine the SDL behaviour for each stimulus, it is necessary to define a signal for each distinct event that can lead to a different transition in the SDL. If it is required to distinguish the same stimulus from different sources, then different signal names should be used. ***A different signal (with a self descriptive name) should be defined for each distinct message event.***

Although it is possible to determine the source of a signal from the communication paths leading to the receiving process, ***the source of the signal in an input should be indicated either by its name or by a comment*** with the INPUT of the signal because it makes it much easier to understand the description. Figure 53 shows alternative methods for indicating the source of a signal.

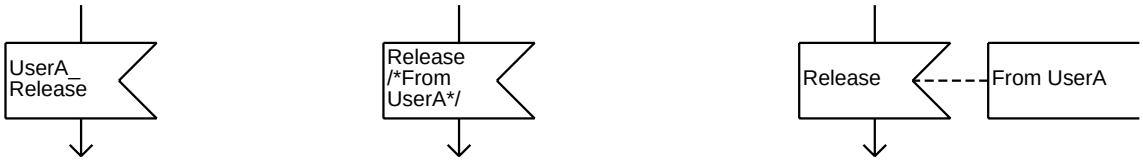


Figure 53: Identifying the source of an input signal

It is possible for messages, particularly those coming from the environment, to be defined in a generic form such that it is necessary to examine the message contents to determine what event it represents. In these cases, a process can be used to translate the generic message into a signals that have a different name for each event.

10.4.5 Multiple outputs

Multiple messages output from a single process are sent in the order that the outputs are interpreted. A single output containing several signals is equivalent to outputting each signal in turn as listed (left to right, top to bottom) in the text of the output. ***There should be only one signal in each output symbol.*** This makes the description easier to read and clarifies the actual order of the outputs.

10.4.6 Transitions triggered by a set of signals

It is sometimes necessary for a process to trigger a transition only when it has received a set of more than one signal (perhaps from the same entity or perhaps from different entities) although the order in which the signals are received is not important. SDL does not have a built-in mechanism for achieving this but the behaviour can be modelled by saving signals and treating each one in turn.

In the example in Figure 54, the process is waiting for two messages (*UserData*, *DataModeReq*) before entering the DataMode state. The *DataModeReq* signal is saved so, if it arrives before *UserData*, it is not lost and can be processed later. Other signals that can be received are treated in the same way regardless of whether *UserData* has been received or not.

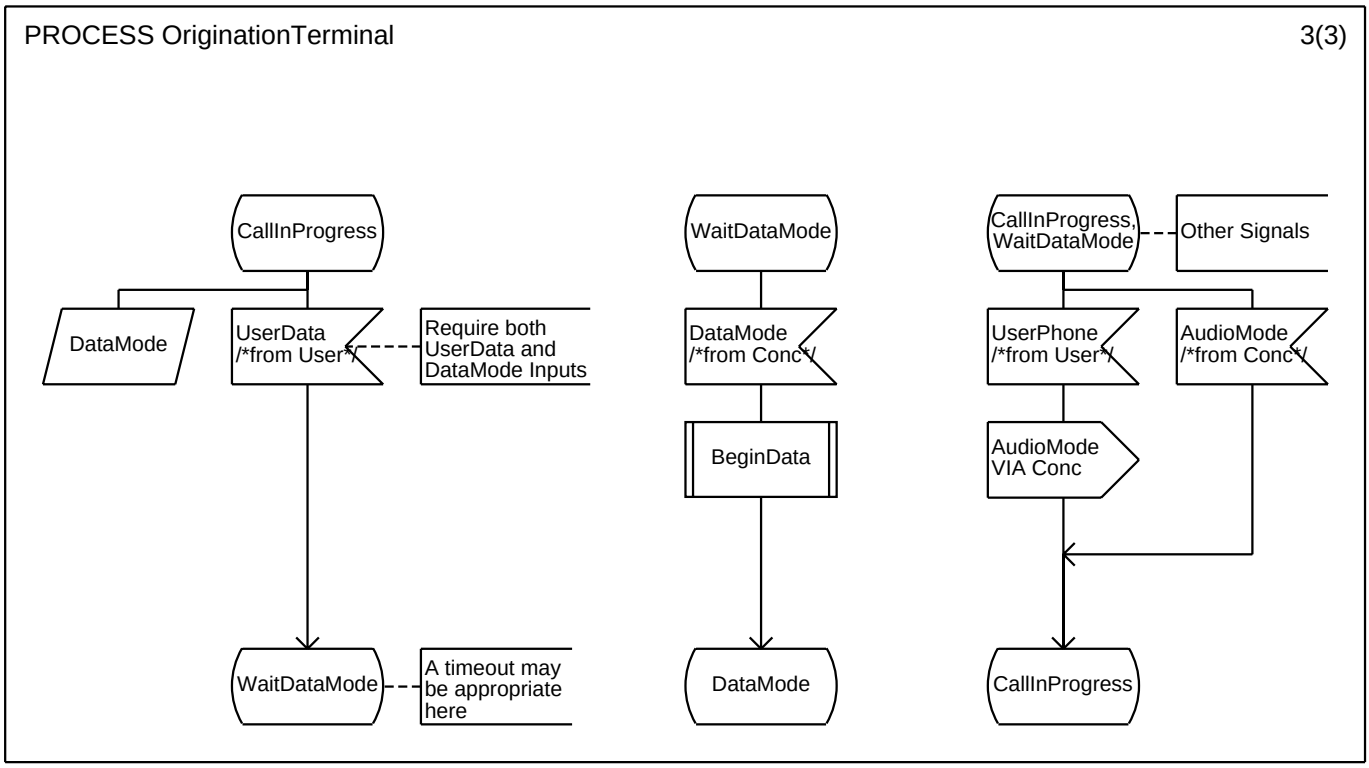


Figure 54: Waiting for multiple messages.

11 Specification and use of data

A very important part of any protocol or service standard is the specification of data. SDL has its own built-in data types and mechanisms to create new data types. However, the standardized data type notation, ASN.1 (see ITU-T Recommendation X.680 [7], is increasingly used in modern telecommunications standards to specify messages and other data. ASN.1 is now an integral part of SDL and may be used as an alternative to SDL data types (see ITU-T Recommendation Z.105 [5]).

NOTE: Strictly speaking SDL data types are called 'Sorts'. However, in this document for the sake of simplicity the term 'data type' is used both in the context of using ASN.1 and SDL Sorts.

ASN.1 should be used in ETSI standards to specify data and the ASN.1 data definitions should be made common to both the SDL specification and the non-SDL parts of a standard.

This approach of common data has the significant advantage of reducing the possibility of confusion and mistakes which can be introduced if there are separate data descriptions of the same or similar data structures.

11.1 Specifying messages

One of the main purposes of using SDL in an ETSI standard is to provide an unambiguous description of the exchange of messages over normative interfaces. ***SDL signals should be used to represent normative messages with ASN.1 describing the***

parameters carried by the messages. The SDL process diagrams (state transitions) describe dynamic mechanisms that control the sending and receiving of these messages.

NOTE: The details of these dynamic mechanisms are not usually normative and the SDL that describes them should be regarded as just one description of any number of possible alternative descriptions. What is normative is the behaviour that the SDL 'machine' exhibits over the normative interfaces with regard to message interactions.

Even though an ASN.1 module will specify the complete set of messages and message parameters relevant to a standard, it is unlikely that all the message parameters will be directly relevant to the SDL model. Note that even if the ASN.1 data type definitions are complex, only those parameters relevant to the dynamic requirements of the standard need actually be used in the SDL behaviour descriptions. In this way, the complexity of the data type definitions does not adversely affect the readability of the SDL specification

11.1.1 Structuring messages

Except in the very simplest of cases, ***the top-level parameters of messages should be contained in a single structured type (e.g., ASN.1 SEQUENCE or SET) rather than specified as a list of simple types***.

For example, the signal specification in case a), below, is equivalent to the longer but considerably more meaningful specification in b).

a)

```
SIGNAL SETUP (BIT STRING, BIT STRING, BIT STRING, BIT STRING)
```

b)

```
..
SIGNAL SETUP(SETUP_Type)


SETUP_Type ::= SEQUENCE
    {   header          Header,         -- Note that these examples follow the ASN.1 convention of
        identifier      Identifier,     -- starting identifiers with lower case letters and starting
        extension_block Extension_Block -- type references with upper case letters.
    }

Header ::= BITSTRING (SIZE 8..32)

Identifier ::= BITSTRING (SIZE 8..8)

Extension_Block ::= SEQUENCE
    {   call_reference Call_Reference,
        party_reference Party_Reference
    }

Call_Reference ::= BIT STRING (SIZE 4..8)

Party_Reference ::= BIT STRING (SIZE 4..8)
```

The ASN.1 in item b) has the added benefit of being able to give explicit names to message parameters in the SDL signal. It also allows the use of SIZE restrictions in a readable manner. Finally, an advantage of using ASN.1 in this example is that it has the pre-defined data type BIT STRING, whereas SDL does not.

The use of structures has the added benefit of allowing the easy capture and manipulation of the entire contents of messages rather than on a parameter-by-parameter basis. Figure 55 shows how the contents of an incoming message can be simply output on another channel. In this example, *Details* has been declared as a variable of a structured type.

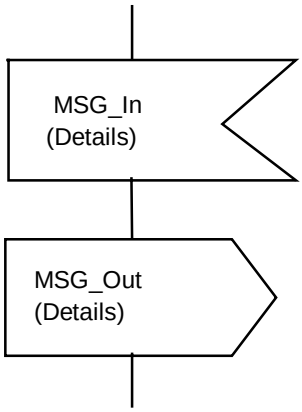


Figure 55: Transferring message contents from Input to Output

A minor drawback of using complex structures is that references to the parameters in the SDL description may be longer. For example, if two variables, 'setup_extension' and 'setup_out' were declared as follows:

```
DCL setup_extension_block Extension_Block;
DCL setup_out SETUP_Type;
```

then an assignment in the SDL to call_reference would be

```
setup_out!setup_extension_block!call_reference:='00001111'B
```

Subclause 8.2 gives details of how operators can be used to hide long references. It also shows how operators may be added to ASN.1 types.

11.1.2 Ordering message parameters

Protocol messages are most easily specified using the ASN.1 constructors: SEQUENCE or SET. ***If the parameters in a message must appear in a fixed order, then the ASN.1 constructor SEQUENCE should be used to specify the message contents***, as in the following:

```
SETUP_Type ::= SEQUENCE
    {   header          Header,
        identifier      Identifier,
        extension_block Extension_Block
    }
```

However, it is common that a protocol specification will allow elements to appear in any order. ***If the parameters of a message may appear in any order, then the ASN.1 constructor SET should be used to specify the message contents.*** For example, in the extension block of the previous example it could be required that it is possible to receive the call_reference and the party_reference in either order, in which case this would be specified as follows:

```
Extension_Block ::= SET
    {   call_reference  Call_Reference,
        party_reference Party_Reference
    }
```

Another useful concept in ASN.1 that is difficult to model using SDL data types is the ability to specify parameters as OPTIONAL. In the following example the party_reference may be omitted

```
Extension_Block ::= SET
    {   call_reference  [1] Call_Reference,
        party_reference [2] Party_Reference OPTIONAL
    }
```

NOTE: Tags ([1] and [2]) have been introduced to enable encoders to differentiate between the two parameters.

Finally, ASN.1, unlike SDL, allows the specification of unions through the CHOICE construct, for example:

```
Any_Message := CHOICE
    {   setup       SETUP_Type,
        release     RELEASE_Type,
        acknowledge ACKNOWLEDGE_Type
    }
```

NOTE: Z.105 [5] states that SET and SEQUENCE are treated in the same way, and that SDL requires parameters to be in a specific order.

11.1.3 Specifying data that is internal to the SDL model

Data that is internal to the SDL model may be specified using either SDL types or ASN.1. In most cases it will be simpler to use SDL data types (though bear in mind the additional capabilities of ASN.1 mentioned in the previous clauses).

Subclauses 9.4 and 9.6 recommend that synonyms or enumerated types should be used to specify symbolic names which can be used as decision labels and which convey meaningful information to the user.

11.1.3.1 Using NEWTYPE and SYNTYPE

Both NEWTYPE and SYNTYPE are SDL constructs which can be used to specify new data types. SYNTYPE is particularly suitable for renaming existing types, for example:

```
SYNTYPE
    Int =Integer;
ENDSYNTYPE;
```

NEWTYPE, on the other hand, is more suitable for specifying new data types that are needed in a specification and which are not included in the existing types. In general ***NEWTYPE should be used to define a new data type in a specification while SYNTYPE should by used to rename existing types***.

It is worth noting that the following ASN.1 specification:

```
B_Type::= A_Type
```

is equivalent to the SDL:

```
NEWTYPE B_Type
    INHERITS A_Type
    OPERATORS ALL;
ENDNEWTYPE B_Type;
```

11.2 Transposing other message formats

In many lower-layer protocol standards, messages are specified using a tabular format. These tables will have to be transposed to ASN.1 or SDL data types in order to be used in an SDL specification. In these cases it will probably be adequate to specify a simplified form of the messages (e.g., by omitting various message parameters). ***When mapping messages described in another format (e.g., tables) to a simplified form as ASN.1 or SDL data types, the structure of the simplified messages should be kept as close to the structure of the original messages as possible and the names of messages and their associated parameters should be preserved.*** The important point is that messages should be reduced to a simpler format in a consistent manner and that the mapping from the real messages to the simplified ones in the SDL is well documented and obvious. Conversely, parameters that are not specified in the full description of the messages should not be introduced in the transposed formal specification.

12 Using Message Sequence Charts (MSC)

12.1 Basic Message Sequence Charts

The syntax and semantics of Message Sequence Charts are described in ITU-T Recommendation Z.120 [6]. A Basic Message Sequence Chart is formed by a finite collection of instances of entities and the messages which describe the communication behaviour between them. An instance of an entity is an object which has the property of the entity. On an instance, message outputs, message inputs, local actions and timer events (timer starting, stopping and expiration) may be specified. In relation to SDL, an entity should be a system, block or process. A message output should correspond to the sending of a signal and a message input to its consumption. Timer expiration (Time-out) should correspond to the consumption of a timer signal.

12.1.1 Instances

There are two graphical forms of instances which may be used together within one MSC but which must not be mixed within one instance: The first form is a single vertical axis (line form) and the second is the so-called column-form (See Figure 56). In both cases, the description of the instance starts with the instance head symbol and ends with the instance end symbol.

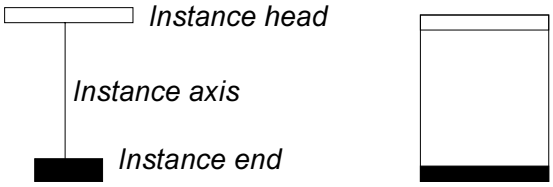


Figure 56: Instance symbols: line form and column form

The column form has the advantage that local actions, such as comments or action identifiers, can be placed inside the column. The instance line form is useful for saving space within the complete MSC.

Every instance has a name associated with it. The optional entity name, e.g. process name, may be specified in addition to the instance name. In relation to SDL, the entity name is preceded by the keyword SYSTEM, BLOCK, PROCESS OR SERVICE. ***Each MSC entity name should correspond to the name of the equivalent entity in the associated SDL***.

The instance name (together with the optional entity name) may be placed above or inside the instance head. If the entity name is present both names may be split such that the instance name is placed inside the instance head symbol and the entity name is placed above. (as shown in Figure 57). It is recommended to use this splitting of instance name and entity name. Otherwise both names have to be separated by a colon symbol.

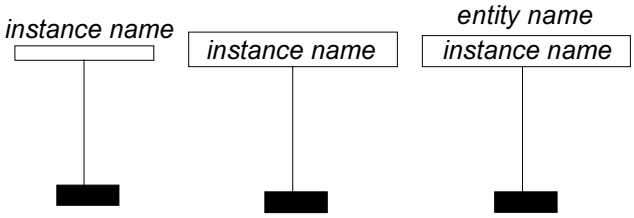


Figure 57: Placement of instance name and entity name

12.1.2 Message communication

A message between two instances is represented by a labelled arrow which starts at the sending instance and ends at the receiving instance (Figure 58).

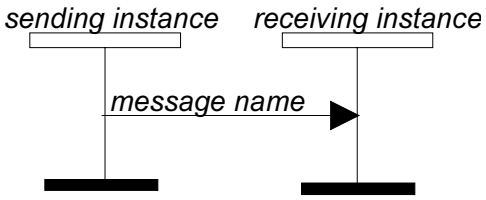


Figure 58: Message symbol

An MSC message name should be placed close to the message with which it is associated.

Messages may cross instances which are placed between the sender and receiver instance and this can be seen in the example in Figure 59.

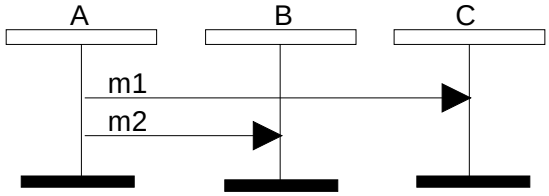


Figure 59: Message crossing an instance

With a better arrangement of the chart, this crossing of instances could be avoided. Figure 60 shows the same message sequence but with a rearrangement to avoid messages crossing the instances.

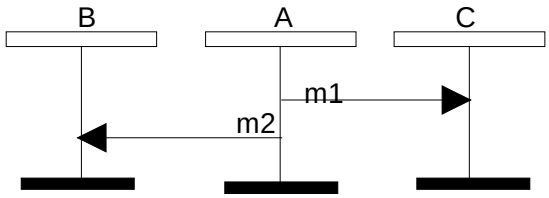


Figure 60: Equivalent message flow without instance crossing

The crossing of MSC instances by messages should be minimised by placing communicating instances close to each other wherever possible (See the example in Figure 61). However, the natural and logical ordering of entities should be considered to be more important than strict adherence to this guideline.

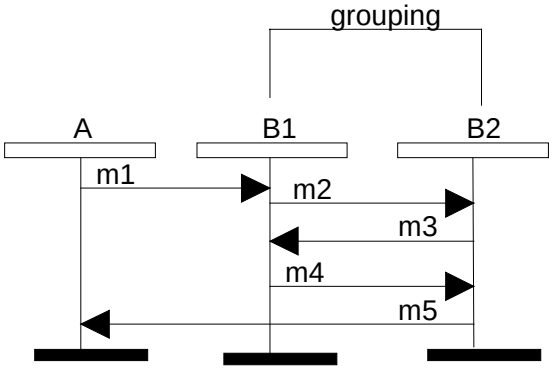


Figure 61: Suitable grouping and placement of strongly coupled instances

A message arrow may be horizontal or with a downward slope. Both forms are equivalent but the downward slope is often used to indicate the passage of time. These forms should not be mixed within one diagram in order to avoid misunderstandings.

A message arrow may also be bent and may even terminate in the sending instance. As shown in Figure 62 and Figure 63, messages with downward slopes and bent messages are very useful for describing the overtaking or crossing of messages. However, ***the unnecessary crossing of message arrows should be avoided since it can obscure the true meaning of an MSC***.

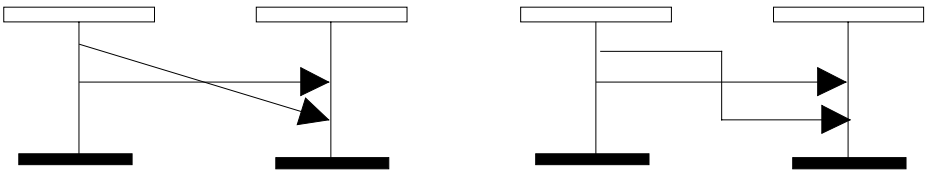


Figure 62: Message overtaking

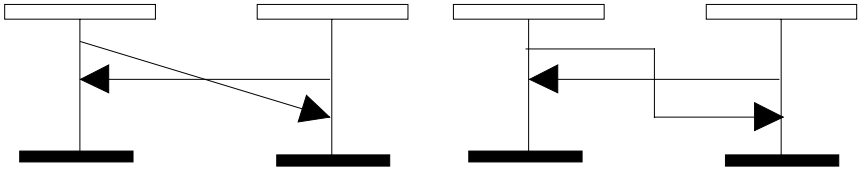


Figure 63: Message crossing

In general, two or more events may not be attached to the same point or at the same level on an instance axis. There is one exception to this rule. An incoming event and an outgoing event may be attached to the same point or at the same height, as shown in Figure 64. This is interpreted as if the incoming event is drawn above the outgoing event (see Figure 65)

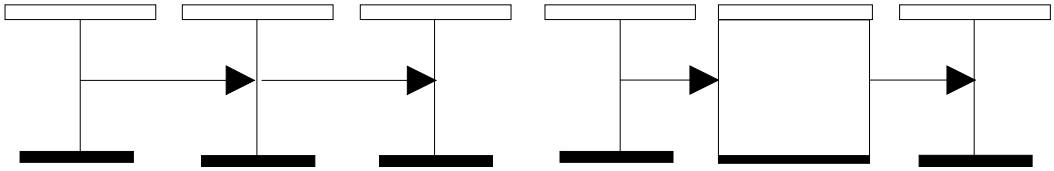


Figure 64: Message representation open to misunderstandings

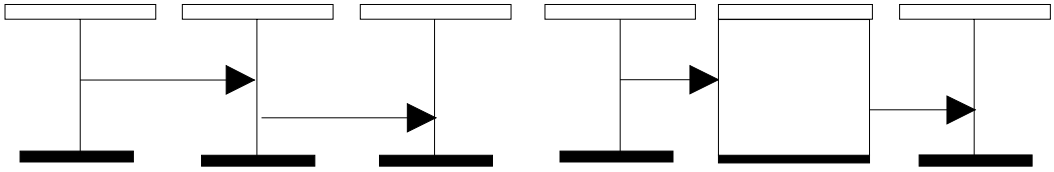


Figure 65: Evident message flow representation

Although both representations are equivalent, ***an MSC within a standard should show an outgoing event below the incoming event that preceded it*** as this presentation gives a clearer description of the ordering relationships.

12.1.3 Lost messages

Besides the specification of successful transmission of messages, lost messages can be described in MSC-96. A lost message is a message which is sent but will never be received by the other party in the communication. Lost messages may be used to describe the reaction of a system in error cases such as. in case of an unreliable transmitter (See Figure 73 in 12.1.6).

Graphically a lost message is indicated by a lost message symbol, i.e. a line from an instance axis to a black dot ("black hole") as shown in Figure 66. In SDL, unsuccessful signal transmission can only be described in an indirect manner.

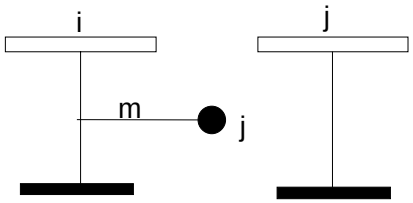


Figure 66: Lost message represented by a “black hole”

12.1.4 Environment

In general, one MSC is only able to describe the possible behaviour of a small section of the system. The rest of the world which is not included in the MSC is called the "environment". Instances may send messages to and receive messages from the environment. Graphically the environment is represented by a frame in the form of a rectangle. Communication with the environment is provided by message arrows connected to the rectangle frame (See Figure 67).

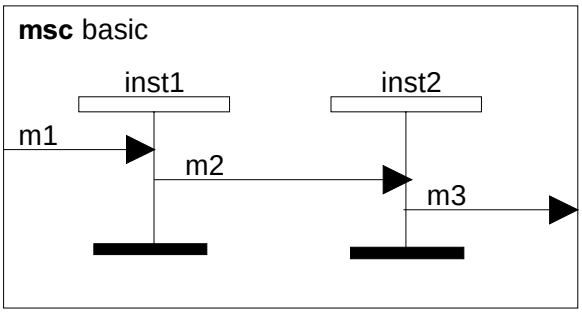


Figure 67: Complete MSC surrounded by an environment frame

As an alternative to the environment frame, instances may be used to describe the interaction of the system with the environment. Such instances are usually specified by means of the instance name *environment.* Unlike the environment frame, these instances allow a concrete behaviour description of external entities which interact with the system under consideration (See Figure 68).

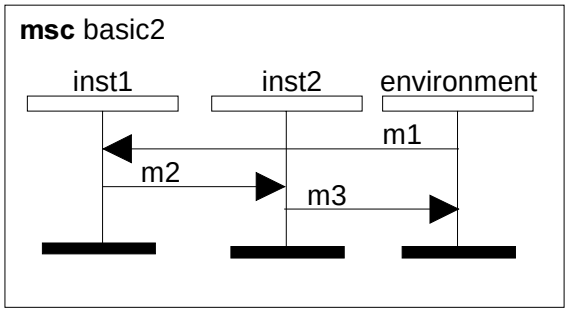


Figure 68: MSC containing an 'environmental' instance

12.1.5 Action

A local action is denoted by an action symbol (rectangle) on an instance with an informal text description of this internal activity placed in it (See Figure 69)

Figure 69: Placement of local actions on line form and column form instances

Although they can be very useful in annotating an instance, ***local actions should not be used in an MSC as a substitute for SDL to describe behaviour requirements***.

12.1.6 Timer handling

MSC contains the following timer events:

- starting a timer;
- stopping a timer;
- expiration of a timer (time-out).

A timer start event is denoted by an hourglass symbol attached to the instance axis by means of a horizontal or bent line (See Figure 70).

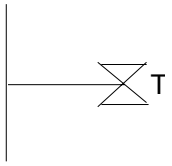


Figure 70: Starting of a timer

A timer stop event is denoted by a cross which is attached to the instance axis by means of a horizontal line (See Figure 71).

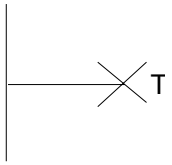


Figure 71: Stopping of a timer

A time-out is represented by an hourglass symbol which is attached to the instance axis by means of an horizontal or bent arrow from the hourglass symbol to the instance axis (See Figure 72).

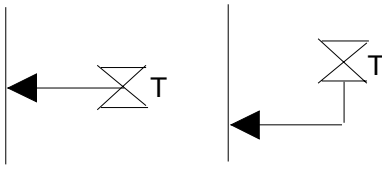


Figure 72: Expiration of a timer (time-out)

If corresponding timer events are used in combination they are connected by a vertical line. A typical use of combined timer events (time supervision) containing incomplete messages is shown in Figure 73. In MSC 'Connection' the repetition of a connection set up which fails twice is shown. The connection set up is supervised by a timer. This example also demonstrates the use of incomplete messages described in 12.1.3.

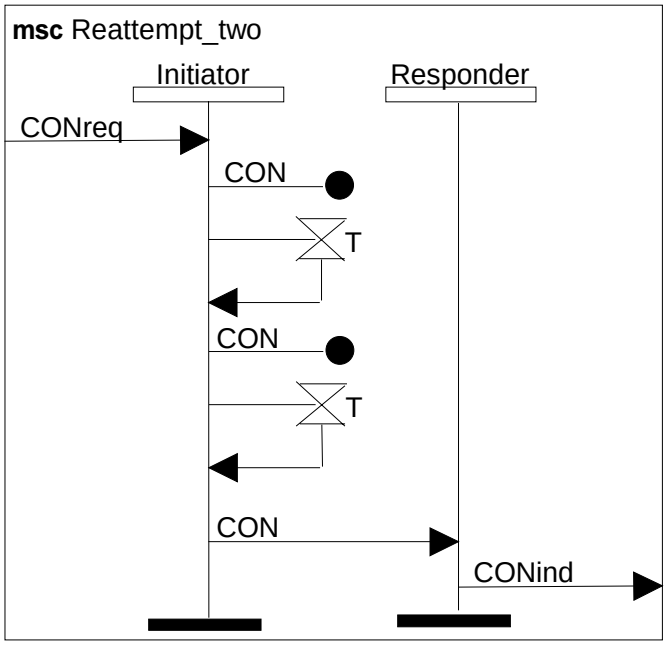


Figure 73: Time supervision

A timer event is local to an instance. It is not allowed to specify a timer start and a subsequent timeout or timer stop on different instances.

Timer events are treated in the same way as message events (See 12.1.2). Within an MSC, timer events and message events should not appear on the same point of the axis.

12.1.7 Coregion

If not otherwise specified, the events along an instance are assumed to be totally ordered in the direction from top to bottom. To enable the specification of unordered events on an instance the coregion is introduced. A coregion is a part of the instance axis in which the events are assumed to be unordered in time. Within a coregion, only message and timer events, actions and process creates may be specified.

Graphically a coregion is indicated by dashing a part of the instance axis as shown in Figure 74.

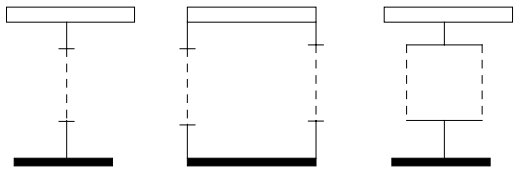


Figure 74: Several forms of coregion

If a timer is set in a coregion and connected to its corresponding reset or timeout then the ordering between these events is preserved.

A coregion is a very convenient and intuitive way of describing the causal independence of events, in particular of message events. However, ***the use of MSC coregions should be restricted to simple cases such as the arrival of two independent messages on the same instance***. The specification of message communication between coregions on different instances in some cases can lead to MSCs which are lacking intuitiveness. In such cases, the employment of composition mechanisms, as described in 12.2, might be more useful.

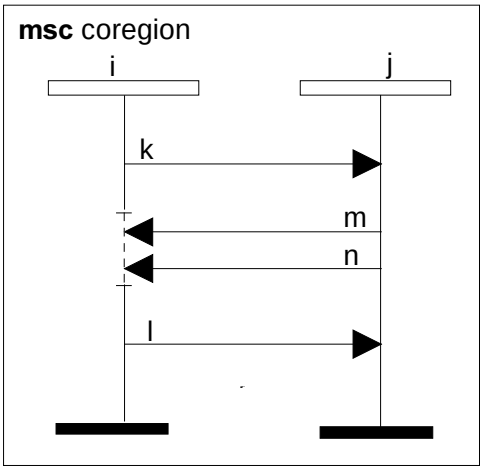


Figure 75: MSCs containing coregions

As an example, within the MSC 'coregion' in Figure 75, message 'm' is sent on instance 'j' before message 'n' but it may be received on instance 'i' either before or after message 'n'.

12.1.8 Conditions

MSC conditions should be used to indicate system states corresponding to states in SDL. Graphically, a condition is represented by a hexagon containing the condition name.

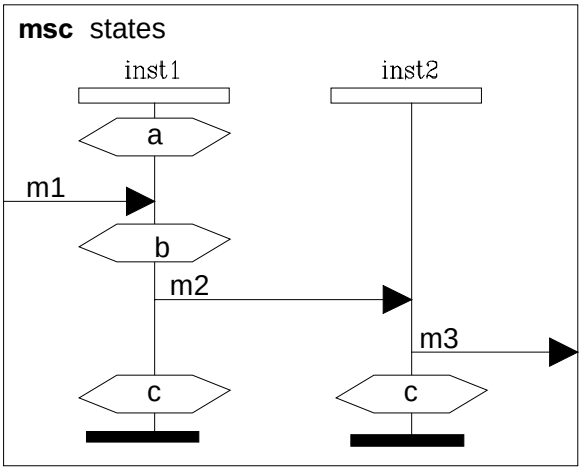


Figure 76: MSC containing local conditions

The conditions shown in Figure 76 are all local, i.e. each one is attached to a single instance. In practice, the use of several local conditions may lead to a loss of transparency. ***An MSC should concentrate on the description of the message flows and should not be obscured by too many other symbols***.

Global conditions provide a more compact presentation in cases where the condition applies to a complete system. They are attached to all instances contained in an MSC and denote global system states (See Figure 77).

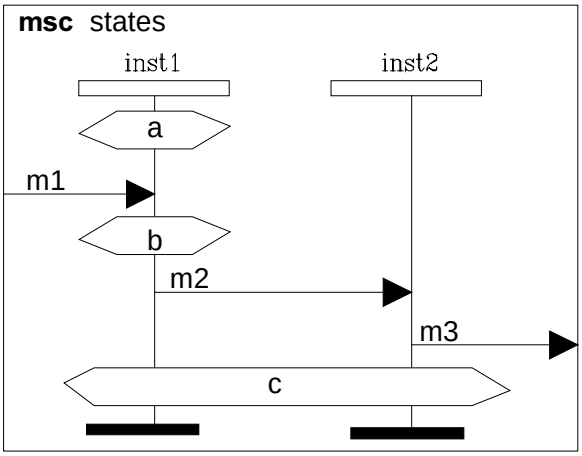


Figure 77: MSC with a global final condition

The real importance of global conditions is in their use as connection points between different MSCs within a set of MSCs. Without the specification of global initial and final conditions, such a set of MSCs appears rather disconnected and is difficult to maintain. An example of an MSC with a global initial condition and a global final condition is shown in Figure 78. Initial and final conditions are normally used in conjunction with a High-Level MSC (HMSC) to provide a useful overview of a set of MSCs (See 12.2.2).

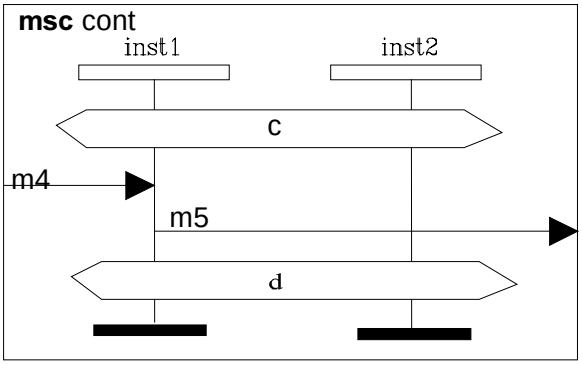


Figure 78: MSC with global initial and final conditions

12.2 Composition

12.2.1 Using MSC references

An MSC reference can be used to refer to other MSCs by means of their MSC name. MSC references may be inserted within plain MSCs or in High-Level MSCs (HMSC) which are described in 12.2.2.

In general, only a small number of MSC references should be used within a plain MSC. HMSCs should be used to illustrate more complex cases. HMSC references may be included in Plain MSCs but referring to "overview" charts from detailed sequence specifications can be confusing. Therefore, ***Plain MSCs should not include HMSC references***. MSC references in plain MSCs should be used as a structuring means and for the reuse of behaviour patterns. Figure 79 shows an example of MSC references used in the specification of a test purpose preamble and postamble. As such, the MSC reference plays a similar role to that of a procedure in SDL. If the same behaviour pattern appears in several MSCs of an MSC document, it should be specified in the form of an MSC reference.

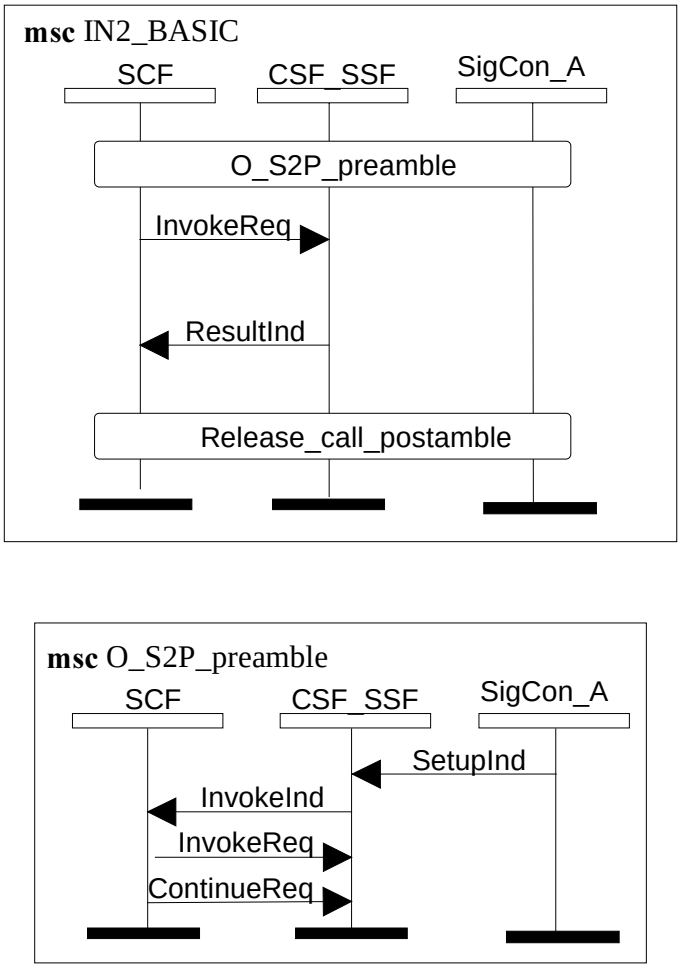


Figure 79: Typical usage of MSC references

12.2.2 Using HMSC

The main purpose of global conditions is to indicate possible continuations of Message Sequence Charts by means of condition name identification. For example, if MSC *A* ends with a final global condition and MSC *B* starts with an initial global condition with the same name then MSC *B* can be regarded as a possible continuation of MSC *A*. The actual composition of MSCs is specified by means of an High Level MSC (HMSC), also informally known as a 'roadmap'. HMSCs provide an attractive graphical way of describing the combination of Message Sequence Charts. For the purposes of transparency and maintenance, all MSCs should use global initial and final conditions to describe the continuation between charts..

The meaning and usage of an HMSC may be illustrated by the following example of the Call Completion to a Busy Subscriber (CCBS) service. All possible combinations (continuations) of the four MSCs, 'Request', 'Reject', 'Activation' and 'Release' shown in Figure 80 can be described by the HMSC 'CCBS' in Figure 81 in a very intuitive manner.

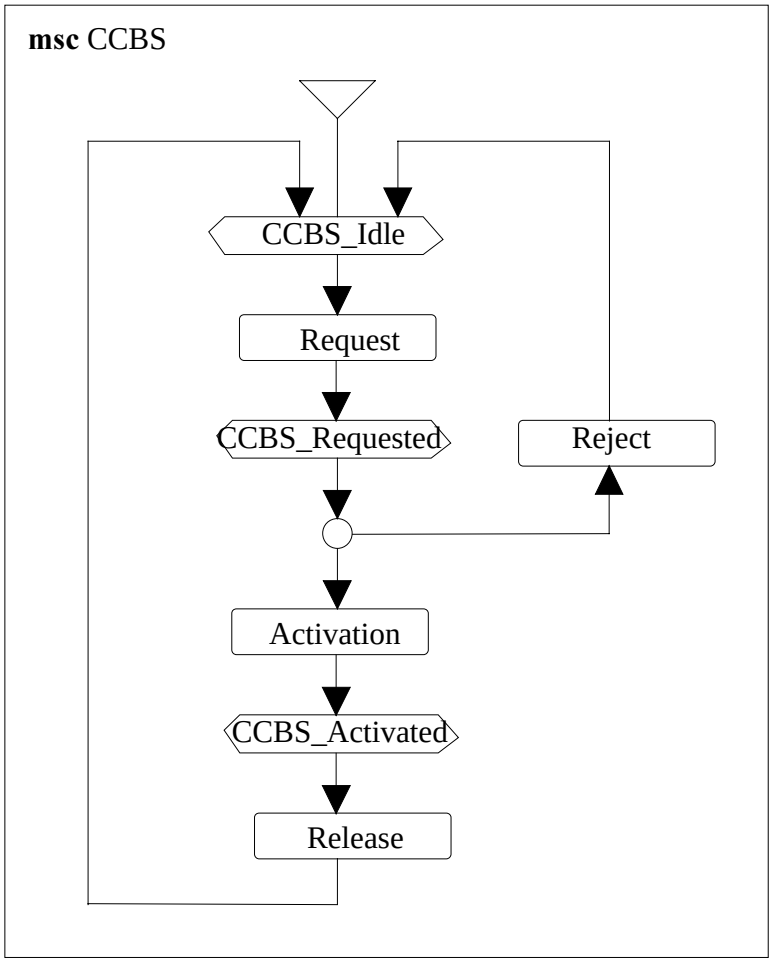


Figure 80: HMSCs showing the combination (composition) of MSCs

Unlike plain MSCs, instances and messages are not shown within an HMSC. In this way, HMSCs can focus completely on the composition aspects.

HMSCs are hierarchical in the sense that an MSC reference may refer to an HMSC. This feature supports a top down design very well. In order to keep HMSCs sufficiently transparent and manageable ***references to other HMSCs should be used to ensure that the number of symbols within one HMSC is kept sufficiently small***.

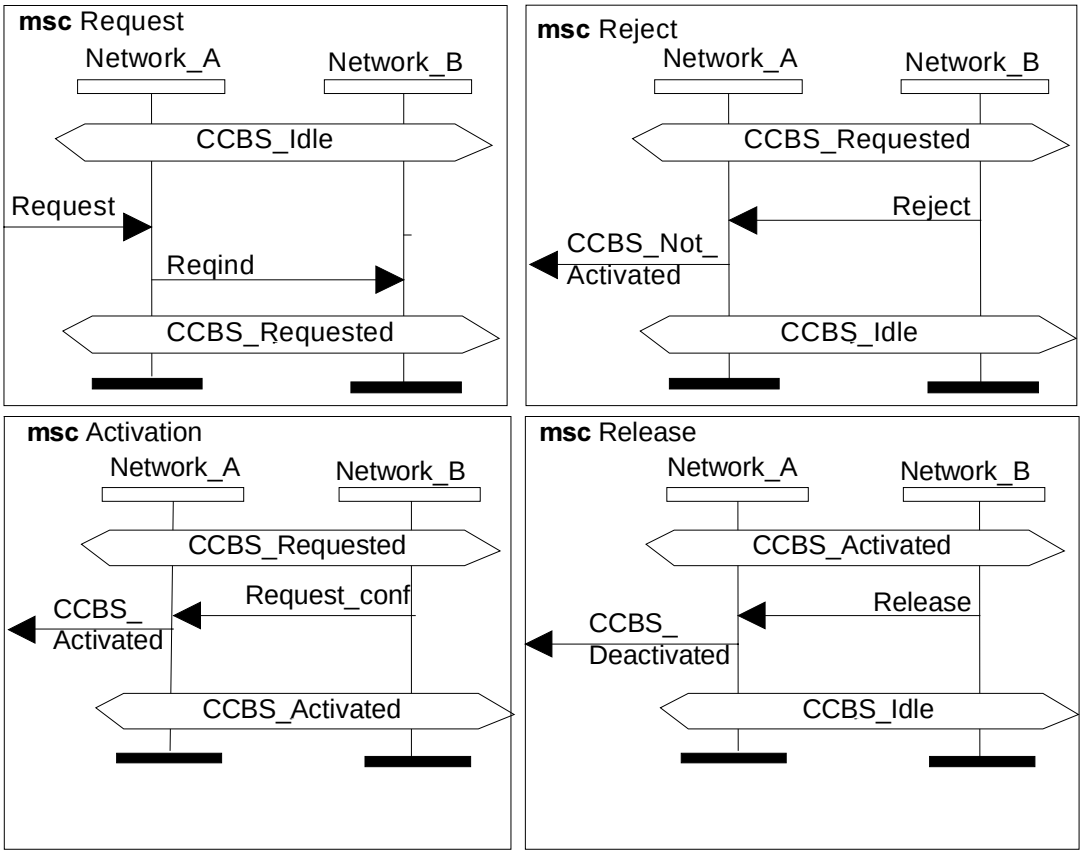


Figure 81: Plain MSCs with possible continuation by initial and final conditions

Annex A Reserved words

The following words are keywords in SDL and cannot be used as names or words separated by spaces within names.

ACTIVE	ADDING	ALL	ALTERNATIVE
AND	ANY	AS	ATLEAST
AXIOMS	BLOCK	CALL	CHANNEL
COMMENT	CONNECT	CONNECTION	CONSTANT
CONSTANTS	CREATE	DCL	DECISION
DEFAULT	ELSE	ENDALTERNATIVE	ENDBLOCK
ENDCHANNEL	ENDCONNECTION	ENDDECISION	ENDGENERATOR
ENDMACRO	ENDNEWTYPE	ENDOPERATOR	ENDPACKAGE
ENDPROCEDURE	ENDPROCESS	ENDREFINEMENT	ENDSELECT
ENDSERVICE	ENDSTATE	ENDSUBSTRUCTURE	ENDSYNTYPE
ENDSYSTEM	ENV	ERROR	EXPORT
EXPORTED	EXTERNAL	FI	FINALIZED
FOR	FPAR	FROM	GATE
GENERATOR	IF	IMPORT	IMPORTED
IN	INHERITS	INPUT	INTERFACE
JOIN	LITERAL	LITERALS	MACRO
MACRODEFINITION	MACROID	MAP	MOD
NAMECLASS	NEWTYPE	NEXTSTATE	NODELAY
NOEQUALITY	NONE	NOT	NOW
OFFSPRING	OPERATOR	OPERATORS	OR
ORDERING	OUT	OUTPUT	PACKAGE
PARENT	PRIORITY	PROCEDURE	PROCESS
PROVIDED	REDEFINED	REFERENCED	REFINEMENT
REM	REMOTE	RESET	RETURN
RETURNS	REVEALED	REVERSE	SAVE
SELECT	SELF	SENDER	SERVICE
SET	SIGNAL	SIGNALLIST	SIGNALROUTE
SIGNALSET	SPELLING	START	STATE
STOP	STRUCT	SUBSTRUCTURE	SYNONYM
SYNTYPE	SYSTEM	TASK	THEN
THIS	TIMER	TO	TYPE
USE	VIA	VIEW	VIEWED
VIRTUAL	WITH	XOR	

Annex B - Summary of guidelines

Identifier	Guideline
NAMING CONVENTIONS	
1	Names of less than 6 characters may be too cryptic and names of more than 30 characters may be too difficult to read and assimilate.
2	Readability is improved if the same convention for separating words within names is used throughout a specification
3	In most cases an underscore character between each word removes any possibility of misinterpretation and this is the approach that is recommended
4	In more complex models where each block is made up of a number of processes and where there are many data items, the use of a single name for multiple entities is likely to cause confusion and should be avoided.
5	The name of an ASN.1 type (i.e., a type reference) should start with an upper case letter, and names of values should start with a lower case letter
6	By giving processes names that represent the overall role that they play within the system, it is possible to distinguish process names from procedure names. If carefully chosen, they can help to link the SDL back to the corresponding subclauses in the text description
7	The names chosen for procedures should indicate the specific action taken by the procedure
8	If possible, it is advisable to leave at least one significant word in the name unabbreviated as this can help to provide the context for interpreting the remaining abbreviations
9	Signal list names can be chosen to indicate the origin and the destinations of the associated signals
10	A state name should clearly and concisely reflect the status of the process while in that state
11	If it is important to number states then this should be done in conjunction with meaningful names
12	The name chosen for a variable should indicate in general terms what it should be used for
13	Names used to identify constants can be more specific by indicating the actual value assigned to the constant
PRESENTATION AND LAYOUT OF PROCESS DIAGRAMS	
14	The flow of SDL process diagrams should be from the top of the page towards the bottom
15	The flow on a page of an SDL process should terminate in a state
16	States that are entered from NEXTSTATE symbols on other pages should always be placed at the top of the page.
17	Where transitions are short and simple they can be arranged side-by-side on a single page
18	When two or more transitions are shown on one page, there should be sufficient space between them to make their separation clear to the reader
19	Connector symbols should generally only be used to provide a connection from the bottom of one page to the top of another
20	All reference symbols and text boxes containing common declarations should be collected together at a single point within the process chart.
21	A new text box symbol should be used for each different type of declaration
22	When the text associated with a task symbol overflows its symbol boundaries, a text extension should be used to carry the additional information
23	Symbols that terminate the processing on a particular page should be aligned horizontally
24	In simple systems where each process communicates with only one or two other processes, the orientation of INPUT and OUTPUT symbols can be used to improve the readability of the SDL
25	The significance of the orientation of SDL symbols should be clearly explained in the text introducing each process diagram
STRUCTURING BEHAVIOUR DESCRIPTIONS	
26	A state, input and the associated transition to the next state should be contained within a single SDL page
27	Process diagrams should segregate normal behaviour from exceptional behaviour.
USING PROCEDURES AND OPERATORS	
28	The use of procedures to modularise specifications and to 'hide' detail is strongly recommended
29	All data relevant to the real behaviour of a procedure should be specified in the parameter list and return value (if any).
30	In most cases it is preferable to use Operators instead of Value-Returning Procedures.
31	Convert informal text descriptions of actions into procedure calls and replace the task symbols with a procedure symbols
32	Procedures should only read and write to variables that are passed to the procedure in the parameter list or are declared within the procedure itself
33	Procedures should specify a level of detail that is suitable for the particular purpose of the standard
34	A functional procedure should fulfil its specified role and do nothing that could be considered to be a side-effect
35	The processing of signals is one of the most important activities shown in the SDL of a protocol standard and should normally be visible in the calling process rather than the called procedure
36	It is important that procedures that specify a limited sequence of actions should be given names that reflect as fully as possible the activity performed by a procedure
37	Behaviour that could be considered a side-effect to its defined purposes, should not be specified in a procedure

Identifier	Guideline
38	In the exceptional case that a procedure includes the specification of one or more states, it is important to ensure that all signals which are not directly processed within the procedure are correctly handled for subsequent processing
39	The names of procedures having multiple effects should reflect each intended effect either individually or collectively
40	The textual syntax of SDL can be used to define simple operators
41	Complex operators should be specified as operator diagrams which are referenced from the relevant data type specification
USING DECISIONS	
42	It is essential that the complete range of values of the data type contained in the decision is covered by ranges of values in the answers without any overlap
43	Identifiers used in decisions should clearly reflect to a reader the 'question' and 'answer' nature of the conditions being expressed.
44	Informal text should be used in decision statements with care and should be limited to those cases where the decision is obviously Boolean in nature
45	In most cases, enumerated types rather than text strings should be used to express decisions.
46	ELSE should be used as a decision outcome value to distinguish between one or more specific outcomes and all other possibilities
47	SYNTYPE expressions should be used to limit the range of values represented by an ELSE branch in a decision
48	SDL SYNONYMS should be used to define meaningful alternatives to the boolean values of 'True' and 'False' if this aids clarity
49	For the purposes of flexibility symbolic names rather than explicit values should be used to express decision outcome conditions
50	Procedure calls should be used in conjunction with decisions to eliminate the use of informal text
51	When a decision is based on a data item that is not directly available to the process, SDL operators should be used rather than value procedures
52	The ANY symbol should not appear in the SDL specifications in standards except where it is included to show the behaviour of an entity (such as a user) that is not the subject of the standard
53	Where mutually exclusive implementation options are to be expressed, the option symbol should be used rather than a decision
SYSTEM STRUCTURE, COMMUMICATIONS AND ADDRESSING	
54	The SDL version of the architecture of a protocol or service should be consistent with and complementary to other (non-SDL) descriptive diagrams
55	Comments should be used to convey to the reader the relationship of the SDL architecture to the relevant non-SDL parts of the standard
56	Informal drawings that duplicate structural information given by the SDL diagrams should not be used
57	The SDL specification within a standard should comprise one system composed of at least one block which in turn is composed of at least one process
58	SDL should be used to show the structure of a system as well as its behaviour
59	SDL sub-structuring should be used to simplify complex SDL models but should not be used excessively.
60	Multiple instances of SDL blocks and processes should be avoided if possible
61	Informative blocks or processes that are not needed to aid understanding should be omitted
62	If the same block or process is required at more than one place within an SDL specification, a BLOCK TYPE or PROCESS TYPE should be defined from which instances can be derived.
63	Wherever possible, a minimal number of static instances should be used instead of dynamically created SDL processes.
64	All normative channels (interfaces) should be clearly marked as being normative (using a comments box)
65	There should be no more than one communication path specified in each direction between one entity and another.
66	Remote procedures and import/export to exchange information between blocks and processes should not be used
67	Signallists should be used to logically group signals on a particular channel
68	All communication paths (SDL channels and signal routes) should be shown with the associated signals or signalists.
69	TO or VIA should be used in an output symbol to indicate the recipient clearly
70	A different signal (with a self descriptive name) should be defined for each distinct message event.
71	The source of the signal in an input should be indicated either by its name or by a comment
72	There should be only one signal in each output symbol.
SPECIFICATION AND USE OF DATA	
73	ASN.1 should be used in ETSI standards to specify data and the ASN.1 data definitions should be made common to both the SDL specification and the non-SDL parts of a standard
74	SDL signals should be used to represent normative messages with ASN.1 describing the parameters carried by the messages.
75	The top-level parameters of messages should be contained in a single structured type (e.g., ASN.1 SEQUENCE or SET) rather than specified as a list of simple types

Identifier	Guideline
76	If the parameters in a message must appear in a fixed order, then the ASN.1 constructor SEQUENCE should be used to specify the message contents
77	If the parameters of a message may appear in any order, then the ASN.1 constructor SET should be used to specify the message contents.
78	NEWTYPE should be used to define a new data type in a specification while SYNTYPE should by used to rename existing types
79	When mapping messages described in another format (e.g., tables) to a simplified form as ASN.1 or SDL data types, the structure of the simplified messages should be kept as close to the structure of the original messages as possible and the names of messages and their associated parameters should be preserved.
USING MESSAGE SEQUENCE CHARTS	
80	Each MSC entity name should correspond to the name of the equivalent entity in the associated SDL
81	An MSC message name should be placed close to the message with which it is associated
82	The crossing of MSC instances by messages should be minimised by placing communicating instances close to each other wherever possible
83	The unnecessary crossing of message arrows should be avoided since it can obscure the true meaning of an MSC
84	An MSC within a standard should show an outgoing event below the incoming event that preceded it
85	Local actions should not be used in an MSC as a substitute for SDL to describe behaviour requirements
86	The use of MSC coregions should be restricted to simple cases such as the arrival of two independent messages on the same instance
87	An MSC should concentrate on the description of the message flows and should not be obscured by too many other symbols
88	Plain MSCs should not include HMSC references
89	References to other HMSCs should be used to ensure that the number of symbols within one HMSC is kept sufficiently small

Annex C - Additional MSC Features

MSC-96 contains several features beyond the described language constructs. They are presented only briefly within these guidelines since they seem to be of less importance for the development of ETSI standards at present. However, their usage may become more established in the future.

C.1 MSC reference expressions

An MSC reference expression is a generalisation of the MSC reference introduced in 12.2.1. An MSC reference may contain an operator expression instead of a reference name. This operator expression is a textual formula containing references to other MSCs in the document via their MSC name. Operators for composing MSCs are **alt**, **seq**, **par**, **loop, opt, exc, subst** operators and parentheses for grouping sub expressions. MSC reference expressions are useful for a compact representation, in particular of several alternatives.

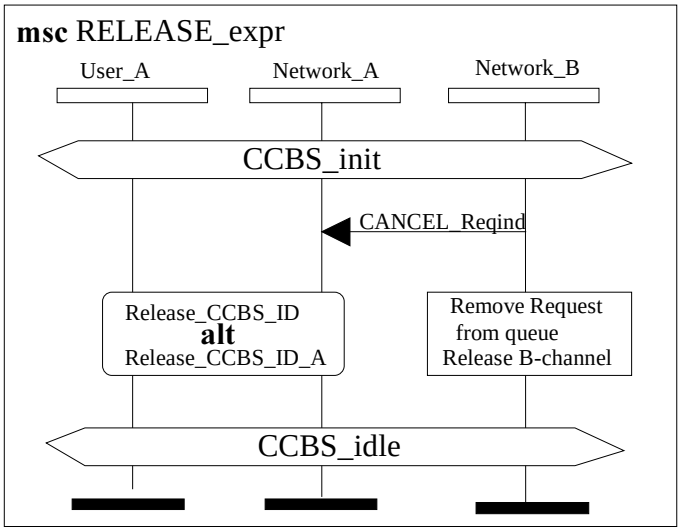


Figure 82: Typical usage of MSC reference expression

C.2 MSC inline expressions

Inline expressions can be looked at as an expanded form of MSC reference expressions. They are ideally suited for the compact description of several small variants. Typically, they cover only a small section of the complete MSC that means the inline expression should contain only few events (as an example see Figure 30).

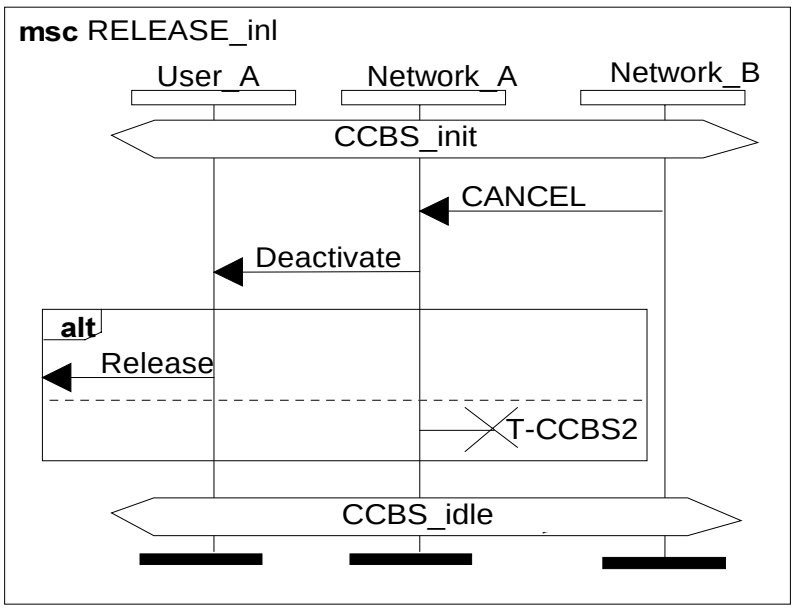


Figure 83: Typical usage of an inline expression

Graphically an inline expression is described by a rectangle with dashed horizontal lines as separators. The operator (**alt**, **par**, **loop**, **opt**, **exc**) is placed in the left upper corner. Guards for the alternatives have to be indicated in form of comments since no formal data description is provided in MSC-96 yet.

C.3 Gates

The transition points of messages through the environment frame are denoted as gates. The gates may be named explicitly. The message gates are used when references to one MSC are put in another MSC.

C.4 Instance decomposition

Instances in MSC may refer to entities of different level of abstraction, in particular, in connection with SDL which is indicated already by the keywords **system**, **block**, **service**, **process**. Instance decomposition defines the transition between these different levels of abstraction. By means of the keyword **decomposed**, a refining MSC may be attached to an instance whereby a formal mapping between decomposed instance and refining MSC is provided for messages.

C.5 Generalised ordering

MSC'96 contains another abstraction mechanism in form of generalised ordering constructs. On an early stage of requirement specification, one often abstracts from the internal message exchange while specifying the external behaviour only. On this level of abstraction, synchronisation constructs are demanded similarly to Time Sequence Diagrams (TSDs) which impose a time ordering between events attached to different instances. This kind of generalised ordering in MSC'96 is provided by means of connection lines.

History

Document history		
V 1.1.1	July, 97	First Draft for comment
V 1.1.2	July 97	Improvements to "Naming" following comments from RR
V 1.1.3	April 98	Addition of Procedures and Decisions
V 1.1.4	June 98	Addition of all other clauses
V.1.1.5	September 98	Consolidation of comments from MTS