SSW

Description
P30308-X5875-A000-06-7618

SICAT
Developers' Environment
for SDL and MSC
Methodology Guidelines

Issued by the Telecommunication
Networks Group, ÖN
Hofmannstraße 51, D-81359 Munich

Copyright © Siemens AG 2005 All rights reserved.

SIEMENS AKTIENGESELLSCHAFT

Author: Dr. Georg Zeis WKA1
Translation: ?????
Notification: A30148- -M70- -69
In addition to the author(s) mentioned on the cover sheet, the following people also worked on his document:

Birgit Storath
Gerd Höfner
Sabine Mittrach
Dieter Kolb

This document contains a total of 103 pages. The status of all pages is 06. The document is based on template MANUALG.DOT, code number P30305-W5299-A001-02-0035. This edition was created on 11.01.2001 16:34. The document was edited using MS WinWord Version 8.0b. The path name of the master document was F:\Sdlmhb_e.doc.
0 GENERAL INFORMATION .............................................. 0-5
  0.1 Issue status ........................................................................................................ 0-5
  0.2 History .................................................................................................................. 0-5
  0.3 References ........................................................................................................... 0-5
  0.4 Terms and abbreviations ...................................................................................... 0-6
  0.5 Keyword / descriptor ......................................................................................... 0-7
  0.6 List of figures and tables ...................................................................................... 0-7

1 Introduction ............................................................................................................. 1-1
  1.1 Object and content of the manual ..................................................................... 1-2
  1.2 Development model .......................................................................................... 1-3
  1.3 Modeling views ................................................................................................... 1-4
  1.4 Development environment for ÖN .................................................................... 1-5
  1.5 SDL and MSC description tools ....................................................................... 1-5

2 General design principles ...................................................................................... 2-1
  2.1 The 7±2 principle (Miller’s Law) ..................................................................... 2-1
  2.2 High cohesion ..................................................................................................... 2-1
  2.3 Loose coupling ................................................................................................... 2-1
  2.4 Modularization ................................................................................................... 2-2
  2.5 Establishment of a hierarchy ............................................................................ 2-2
  2.6 Abstraction ......................................................................................................... 2-3
  2.7 Maximum possible FAN IN ............................................................................. 2-3
  2.8 Acceptable FAN OUT ....................................................................................... 2-3
  2.9 Information Hiding ............................................................................................ 2-4
  2.10 Proximity .......................................................................................................... 2-4
  2.11 Uniformity ........................................................................................................ 2-4

3 SDL and MSC concepts ......................................................................................... 3-5
  3.1 System and communication structure ............................................................. 3-6
  3.2 Dynamic process creation ................................................................................. 3-8
  3.3 Message Sequence Charts ............................................................................... 3-10
     3.3.1 MSC base elements ....................................................................................... 3-11
     3.3.2 Structuring concepts in MSC ....................................................................... 3-13
     3.3.3 SICAT differences and additions in comparison with Z.120 ....................... 3-16
  3.4 Process behavior ............................................................................................... 3-18
     3.4.1 SDL process as a finite state machine ........................................................... 3-18
     3.4.2 SDL process diagram .................................................................................... 3-19
     3.4.3 Procedures and macros .................................................................................... 3-21
     3.4.4 Input behavior of SDL processes .................................................................... 3-23
     3.4.5 Addressing SDL processes .......................................................................... 3-25
     3.4.6 SDL timer concept ........................................................................................ 3-28
     3.4.7 Non-determinism in SDL ............................................................................. 3-30
     3.4.8 SDL wildcards ............................................................................................... 3-31
  3.5 Data definition, structuring and modeling with SDL ...................................... 3-32
  3.6 Criteria for generating SDL and MSC models .............................................. 3-34
### Inhaltsverzeichnis

3.6.1 Design criteria for the system diagram ................................................................. 3-34
3.6.2 Design criteria for substructure diagrams (block interaction diagrams) .......... 3-35
3.6.3 Design criteria for block diagrams (process interaction diagrams) ................... 3-36
3.6.4 Design criteria for process diagrams ................................................................. 3-37
3.6.5 Design criteria for MSCs .................................................................................. 3-39
3.6.6 Design criteria for sub-MSCs .......................................................................... 3-39

4 Using SDL/SICAT during the ÖN TN development process 4-41

4.1 Analysis ........................................................................................................ 4-43
4.2 Design .......................................................................................................... 4-45
  4.2.1 Drawing up the D Spec and process diagram scaffolds .................................. 4-45
  4.2.2 Coding the subsystem interfaces ................................................................. 4-47
  4.2.3 Design validation with the SICAT simulator ............................................... 4-47
  4.2.3.1 The principles and methods of systematic testing ...................................... 4-47
  4.2.3.2 Test specification and preparation of the test runs with the SICAT simulator 4-48
  4.2.3.3 Executing the test using the SICAT simulator ............................................. 4-51
  4.2.3.4 Evaluating the results .................................................................................. 4-51
4.3 Implementation ................................................................................................ 4-52
4.4 Phase transitions/ consistency ........................................................................ 4-54
4.5 Summary of procedure with SDL/MSC ............................................................ 4-55
4.6 Example ........................................................................................................... 4-59
  4.6.1 Description of a system for processing sheet metal ...................................... 4-59
  4.6.1.1 General notes on the components of the production cells ..................... 4-59
  4.6.1.2 Description of production cell 1 ............................................................... 4-60
  4.6.2 SDL specification for the system for processing sheet metal ...................... 4-62
    4.6.2.1 Analysis ........................................................................................................ 4-62
    4.6.2.2 Validating a design using the SICAT simulator ...................................... 4-87

5 CM and the production of SDL CM sources 5-1

6 Appendix 6-2

  6.1 The SDL block diagram symbol set ............................................................... 6-2
  6.2 SDL process diagram symbol set and wildcards ............................................. 6-3
  6.3 The MSC symbol set ...................................................................................... 6-5
  6.4 Differences in SICAT from Z.100 and Z.120 ............................................... 6-6
  6.5 SOLUTION O.N.E .......................................................................................... 6-11
    6.5.1 The client/server principle ................................................................. 6-11
    6.5.2 SOLUTION O.N.E - shell model ......................................................... 6-12
    6.5.3 SOLUTION O.N.E structure units ...................................................... 6-13
    6.5.4 Design time units ...................................................................................... 6-14
    6.5.4.1 Module ................................................................................................. 6-14
    6.5.4.2 Process ................................................................................................. 6-15
    6.5.4.3 Region .................................................................................................. 6-17
    6.5.4.4 SPU and PIF ......................................................................................... 6-17
    6.5.4.4.1 Design influences and rules when creating SPUs ...................... 6-18
    6.5.4.5 Recovery Suite .................................................................................... 6-22
  6.6 Tables and overviews ..................................................................................... 6-25
  6.7 Glossary .......................................................................................................... 6-28
0 GENERAL INFORMATION

0.1 Issue status

This document contains a total of 137 pages. The status of all pages is 06.

0.2 History

<table>
<thead>
<tr>
<th>Status</th>
<th>Date</th>
<th>Reason for change</th>
</tr>
</thead>
<tbody>
<tr>
<td>06</td>
<td>11.02.97</td>
<td>Change to WinWord and adaptation to SICAT V14</td>
</tr>
</tbody>
</table>

*Table 1: History*

0.3 References


[21] Leaflet PPP:D for large systems. TNR A1R1340*.

[22] The Tree and Tabular Combined Notation. Interim Version of ISO DP 9646/3


[29] V9C, CHILL Language and Compiler. P30308-A2737-A000-*-7618

0.4 Terms and abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADT</td>
<td>Abstract Data Type</td>
</tr>
<tr>
<td>ASN.1</td>
<td>Abstract Syntax Notation One</td>
</tr>
<tr>
<td>CCITT</td>
<td>Comité Consultatif International Télégraphique et Téléphonique</td>
</tr>
<tr>
<td>ESP</td>
<td>Engineering Support Process</td>
</tr>
<tr>
<td>ERM</td>
<td>Entity Relationship Modeling</td>
</tr>
<tr>
<td>DMN</td>
<td>Developers’ Manual</td>
</tr>
<tr>
<td>D Spec</td>
<td>Design Specification</td>
</tr>
</tbody>
</table>
0.5 Keyword / descriptor

Not applicable

0.6 List of figures and tables

Figure 1-1: The pillars of system development ................................................................. 1-1
Figure 1-2: SW lifecycle according to Boehm [3] .............................................................. 1-3
Figure 1-3: Extract from the ÖV TN software development process plan [20] .................. 1-4
Figure 1-4: Complete system views .................................................................................. 1-5
Figure 3-1: Typical SDL and MSC hierarchy of a system .............................................. 3-6
Figure 3-2: SDL system diagram ..................................................................................... 3-7
Figure 3-3: SDL substructure diagram ............................................................................. 3-7
Figure 3-4: Tree structure of an SDL system specification ............................................ 3-8
Figure 3-5: SDL process interaction diagram ................................................................. 3-9
Figure 3-6: Example of instantiation .............................................................................. 3-9
Figure 3-7: MSC base elements ...................................................................................... 3-11
Figure 3-8: Dynamic generation of instances within an MSC ....................................... 3-12
Figure 3-9: Using timers ............................................................................................... 3-12
Figure 3-10: Comments in MSC ................................................................................... 3-13
Inhaltsverzeichnis

Figure 4-17: Substructure diagram for cell 1 - division into functional groups ......................... 4-68
Figure 4-18: MSC of a processing cycle for functional groups in cell 1 ................................. 4-69
Figure 4-19: Shared action units of functional units in cell 1 ................................................. 4-70
Figure 4-20: Block diagram for the "Robot" functional group - division into functional units,
   Design phase ................................................................................................................... 4-72
Figure 4-21: Process interaction diagram for the "RO_MANAG" functional unit ..................... 4-77
Figure 4-22: Process interaction diagram for the "RO_UNIT" functional unit ....................... 4-78
Figure 4-23: Message sequence between the processes of the "RO_UNIT" functional unit .... 4-79
Figure 4-24: Process diagram for the RO_MAN process ..................................................... 4-82
Figure 4-25: Process diagram for the "ARM_UN" process .................................................. 4-84
Figure 4-26: Process diagram for the "ZB_GEN" process .................................................... 4-85
Figure 4-27: Process diagram for the "PR_MAN" process ...................................................... 4-86
Figure 4-28: State event table for the "ARM_UN" process ................................................... 4-87
Figure 6-1: Client/server model ............................................................................................. 6-11
Figure 6-2: SPU with one service and SPU with several services ........................................... 6-12
Figure 6-3: Shell model of the SOLUTION O.N.E system architecture .............................. 6-13
Figure 6-4: SOLUTION O.N.E: Extract from lifecycle ......................................................... 6-14
Figure 6-5: Mapping an elementary module in SDL ............................................................. 6-15
Figure 6-6: Modeling aspects of a module ............................................................................ 6-15
Figure 6-7: Mapping the process structure in SDL ................................................................. 6-16
Figure 6-8: Modeling aspects of a process .......................................................................... 6-16
Figure 6-9: Mapping the region structure in SDL ................................................................. 6-17
Figure 6-10: Modeling aspects of a region ......................................................................... 6-17
Figure 6-11: SPU as a visibility border [25] ......................................................................... 6-18
Figure 6-12: Different views in SPU design ......................................................................... 6-20
Figure 6-13: SDL/MSC modeling for SPU and PIF .............................................................. 6-21
Figure 6-14: Modeling aspects of SPU PIF ......................................................................... 6-21
Figure 6-15: SOLUTION O.N.E structural hierarchy ......................................................... 6-23
Figure 6-16: Classification of SOLUTION O.N.E structure units ........................................... 6-24
Figure 6-17: "Conventional Switching Software" quick reference guide ................................. 6-25
Figure 6-18: "SOLUTION O.N.E. Software" quick reference guide ....................................... 6-26
Figure 6-19: "SDL/MSC Methodology" quick reference guide .............................................. 6-27

Table 1: History .................................................................................................................... 0-5
1 Introduction

These methodology guidelines are based on the 1992 CCITT Z.100 Recommendations for SDL (excluding object-oriented concepts) and on the 1993 Z.120 Recommendations for MSC.

ITU/TS is the new name for the CCITT (Comité Consultatif International Télégraphique et Téléphonique).

SDL (Specification and Description Language) has a long tradition in the area of telecommunications. Since 1976, the ITU (previously the CCITT) has published a standardized description of SDL every four years. The latest SDL standard (Z.100 Recommendations) was drawn up in 1992 ("SDL 92") and also takes account of paradigms of object orientation.

The original aim of the CCITT was to define a graphical design language for computer-based switching systems. This is why SDL is specially tailored for distributed and concurrent systems. However, SDL can also be used to describe any event-oriented system based on message exchange, e.g. for modeling production and automation systems as well as designing dialog interfaces.

Message Sequence Charts (MSC) have a very close link to SDL and enable communications scenarios to be modeled.

However, applying a method involves more than simply using it as a (graphical) descriptive tool. A method is actually a rationalized procedure for achieving an objective.

A developer who uses a method “manually” never attains the level of efficiency of one who applies a method using a suitable (case) tool.

An institution involved in the development of systems only gains the full benefit from methods if, in addition to suitable tools, it has at its disposal a defined procedural model that precisely regulates which method is to be applied at which phase of the system generation process (see Figure 1-1).

![Figure 1-1: The pillars of system development](image)

The ICN Group already has tried-and-tested procedural models for hardware and software development with defined milestones and phase results that are binding on developers.

Software development is regulated by the Software Development Manual (SW DMN) [20], while hardware developments are covered by the Hardware Development Manual (HW DMN) [11].
The use of SDL and MSC at ICN is supported by the SICAT tool, which was developed within ICN itself.

This methods manual is supplementary to the SW DMN and is intended to help SDL and MSC users in the development of high quality telecommunications systems. The appendix deals with SOLUTION O.N.E. in detail by way of example.

1.1 Object and content of the manual

Although SDL and MSC are highly suitable for telecommunications applications, this manual shows how they can also be put to general use, i.e. in other applications.

Nonetheless, the producers of telecommunication software remain the prime target group of this manual. These include developers, as well as the Systems Engineering Department which provides the basic design for switching systems.

The manual offers methodical support for the use of the SICAT tool, but does not replace the operating instructions for the tool.

The aim of the methods manual is to describe how SDL can be used to model telecommunication systems.

Chapter 2 of the MMN describes general design principles that are relevant to all types of systems.

Chapter 3 outlines SDL and MSC concepts and offers the reader an overview of the presentation and language tools of SDL and MSC. Not all the SDL constructs shown are supported by SICAT.

The basis for the use of SICAT and SDL at ICN is provided by the Software Development Manual (SW DMN). The SDL documents generated using SICAT must conform to the DMN, i.e. it is necessary to define the points at which SDL and MSC documents can be used in the concept paper, requirement specification (RSpec), functional specification (FSpec) and design specification (DSpec).

Chapter 4 explains the points at which SDL and MSC diagrams can be used in the DMN. This chapter also contains a step-by-step explanation of the method for modeling a system with SDL and MSC on the basis of an example.

Chapter 5 contains a reference to an ICN guideline document that describes the integration of SDL documents in the CM pool and the procedure for producing SDL CM sources.

As well as summarizing the SDL and MSC symbol sets, the appendix (chapter 6) contains a subsection describing various divergences from standards Z.100 and Z.120 within SICAT.

In addition, the appendix also explains how SOLUTION O.N.E. structural units can be modeled using the SDL language tools.

The appendix also contains tables and diagrams that are intended as a quick reference guide for developers.

The appendix is rounded off with a glossary of technical terms.

The first version of the manual describes how to use SICAT and SDL for the DMN Analysis, Design and Implementation phases. Subsequent versions of the manual deal with the Integration and System Test phases, as well as expanding on the original text.

For SDL the manual is based on the 1992 Z.100 Recommendations (SDL 92) [6], excluding the object-oriented concepts, and for MSC on the 1993 Z.120 Recommendations [7].
1.2 Development model

The SW DMN divides software development into development phases that are separated from each other by so-called baselines. The DMN defines which results must be available at the end of each phase and which activities are required within a phase in order to achieve these results.

Milestones embedded within a phase allow projects to be monitored in an efficient way. For example, in the Analysis phase milestone M110 means that the R Spec (customer requirement specification) is available, while milestone M140 means that the feature sheets and F Spec have been generated.

One of the first phase models that represents the lifecycle of a software project is Barry Boehm’s cascade model [3]. This model is still used today and divides the development process into three main phases: Analysis, Design and Implementation (see Figure 1-2).

![Figure 1-2: SW lifecycle according to Boehm [3]](image)

The requirements placed on the system to be created or product to be developed are defined in the Analysis phase. The subcomponents of the system to be created, the communication structure between the subcomponents and their functional profile are all defined in the Design phase. The subcomponents are implemented and integrated in the overall system in the Integration phase.

New phase models have been developed over the years, such as the prototyping model and the spiral model. However almost all phase models are based on the classic cascade model. Even the DMN is based on the cascade model, identifying six development phases:

1. Analysis
2. Design
3. Implementation
4. Integration Test
5. System Test
6. Deployment
This methods manual describes how to use SDL and MSC for the Analysis, Design and Implementation phases.

Figure 1-3 shows an extract from the software development process for ÖN with baselines (defined end points for phases) and the development documents that represent the interfaces between the phases. For example, baseline B200 has been reached once the Analysis phase has been completed. The concept paper, requirement specification and function specification documents form the output from the Analysis phase and the input for the Design phase. The various development documents are the results of corresponding milestones within a phase. Figure 1-3 only lists those development documents that can contain SDL and MSC documents.

![Diagram of software development process]

**Figure 1-3: Extract from the ÖV TN software development process plan [20]**

### 1.3 Modeling views

A complete description of a system encompasses several perspectives (see Figure 1-4). A system is not defined by function, data and behavior alone, but must also be examined in terms of structure (modularity and hierarchy), communicative aspects (data exchange between system components and between system components and the system environment) and the view of the system presented to a user (user interface).

The function view takes account of the functional requirements placed on a system, i.e. its functional profile.

The structure of a system (division into components and hierarchical levels) defines the static system architecture.

Data represents the most stable element in a system's lifecycle and should not therefore be neglected.

As well as static aspects, systems also have dynamic aspects. These dynamic aspects include the system's behavior and the sequence of the data flow within a system and between the system and the environment (dynamic communication).

The static communication view only describes the interfaces of system components. Unlike the dynamic communication view, it offers no information about dynamic message flows.
Although ergonomic research results and rules for ergonomically designed user interfaces have been around for several years, powerful user interface builders that enable user interfaces to be created very quickly have meant that the design of the user interface has increasingly become a separate aspect within System Engineering [27].

Figure 1-4 indicates the methods and descriptor tools that can be used to model the various system aspects. Applications for SDL and MSC documents are highlighted.

**Figure 1-4: Complete system views**

### 1.4 Development environment for ÖN

This methods manual is intended to support the use of the SICAT SDL tool. SICAT is part of the software supplied on SAP/3 (Software workstation/3).

The SAP/3 consists of a PC running the OS/2 operating system. The PF3 (Productivity Facility 3) tool is available within SAP/3 for documentation purposes. It must be possible to integrate all SDL diagrams generated in SICAT in PF3 documents.

### 1.5 SDL and MSC description tools

SDL (Specification and Description Language) is a design method [5] and a specification language [6] that is ideally suited to the description of event-oriented and systems based on message exchange. SDL complies with ITU standards (Z.100 Recommendations) [6] and has established itself as the standard design method in the telecommunication sector.

The language description for MSC (Message Sequence Charts) also complies with ITU standards (Z.120 Recommendations) [7]. MSCs can be used to represent in terms of time the communication flow between system components and between system components and the environment. A Message Sequence Chart does not normally show the complete communication behavior of a process or system, but only one particular communication scenario in each case.

In contrast, an SDL process diagram always describes the overall behavior of a process.

As well as the graphical representation, SDL and MSC also have an equivalent textual form that uses a notation similar to a programming language.
For the sake of clarity, graphics have been used in this manual instead of textual descriptions. Text is mainly used as an output format and as the basis for further electronic processing.
2 General design principles

A number of principles and guidelines have become established within system development in order to help a system developer to create reliable, understandable and low maintenance systems. The principles outlined in this chapter are mainly based on the idea of structured design [14, 15, 28].

2.1 The 7±2 principle (Miller’s Law)

According to Miller’s Law [13] a human being’s rate of error increases considerably when he is required to deal with 7±2 things at the same time. This is the reason why modular and hierarchical breakdowns should not exceed this range when it comes to software design.

2.2 High cohesion

The contents of a subsystem, module or process should never be random but should instead have a certain internal consistency [14, 15, 27]. This module consistency is referred to as cohesion and should be as high as possible. According to the proximity principle, elements that belong to the same problem area should also be located in the same module in the model.

Cohesion is the state that holds the elements of a module together. There are several types of cohesion which differ in terms of quality. Functional consistency is the highest form of cohesion. A functionally coherent module executes a single precisely defined function that can be described briefly and concisely.

Function in this case does not mean a mathematical function that defines a mapping rule from one value range to another, but rather, when referred to in the MMN, function means a task or operation that does not necessarily return a value.

Module is understood to mean a structural unit within software technology that contains local data and functions.

The worst form of cohesion is random cohesion. Here, the elements of a module do not have any rational, recognizable interconnection.

Although functional cohesion is the highest level of cohesion, its consistent observance would result in a system consisting of innumerable modules, each of which is only responsible for one function. This is why informative cohesion is to be attempted when forming modules.

Informative cohesion brings together the functions that operate on the same data structure in a single module. Informative cohesion is an important design criterion when it comes to the formation of abstract data types (ADTs) and a major aspect in object orientation.

2.3 Loose coupling

While cohesion is concerned with the inner life of a subsystem, module or process, coupling refers to the relationships between two subsystems, modules or processes. Intramodular relationships should be as high as possible, while intermodular relations are kept as low as necessary.

Loose coupling is a sign of a well-structured system.

Loose coupling makes modules easier to maintain because no internal details about other modules are required for this. In addition, loose coupling suppresses the “ripple” effect whereby an error in one module appears as a symptom of another module. Coupling is defined by the type and number of transfer parameters. The number of parameters should not exceed the 7±2
limit. If individual transfer parameters are grouped together to form a data structure, a rational link should be apparent between the parameters. Random grouping of individual transfer parameters to form transfer structures may reduce the official number of transfer parameters, but will actually result in a poorer coupling. Grouping parameters together to form structures increases the risk of “tramp data”. Tramp data is data passed on by a system without being used. Tramp data means poorer serviceability and increased susceptibility to error.

Just as with cohesion, there are various types of coupling which differ in terms of quality. Data coupling is the best form of link. Two modules are linked together with data coupling if they communicate directly with each other, i.e. if they call each other explicitly and if their transfer parameters are homogeneous data structures. Homogeneous data structures are elementary data types or fields of elementary data types.

The worst type of coupling is content coupling. In this case a module refers directly to the “interior life” of another module or manipulates it. Coupling and cohesion are closely linked. The greater the cohesion of the various modules, the less coupling there is between them. However, it takes less effort to enter intramodular relationships than to examine the relationships between modules. For this reason one should primarily concentrate on cohesion during the design process [28].

2.4 Modularization

This means breaking a system down into subsystems in order to reduce the complexity of the overall system. However, if the relationships between subsystems is stronger than the relationships between components within a subsystem, the complexity of the overall system is not reduced but rather increased. To prevent this, modularization must satisfy certain criteria. Thus, each subsystem (each module) must contain a complete task. Complete tasks and agreed design decisions may not be distributed over several modules. If possible, changes in a module should not lead to changes in other modules (principle of mutual non-interference).

2.5 Establishment of a hierarchy

Hierarchical breakdowns make systems much more transparent. The establishment of a hierarchy (vertical division of a system into levels) as well as modularization (horizontal breakdown into subsystems) is intended as a response to the limits of the human capacity for dealing with diverse elements. The establishment of a hierarchy requires that levels of abstraction should be created whose elements meet certain user requirements through their degree of abstraction. The modularization, the establishment of a hierarchical structure must also follow certain rules. Thus, it does not make sense to structure 100 modules in such a way that 99 of them are grouped directly beneath a “chief module”. Such a "short and fat" hierarchy (in which a superordinate module must coordinate and integrate all other modules directly) is just as useless as a “tall and lean” hierarchy in which modules are arranged in such a way that each has no more than one module beneath it. The depth (number of hierarchical levels) of a hierarchical breakdown must remain as transparent as its breadth (average number of elements per level). The literature suggests that the depth and breadth should not exceed the 7±2 range [15, 27].

The upper levels of a hierarchy should be responsible for controlling and coordination tasks, while the lower levels should carry out calculation and execution tasks. Accordingly, elements in higher levels should operate with logical data and elements in lower levels should use physical data. This rule is referred to as balancing with structured design (balancing has another meaning in structured analysis (SA)).

A well-structured, consistent hierarchy requires that the relationships between the elements of different hierarchical levels should be preserved throughout. Typical hierarchical relationships...
for software technology are the “consists of” relation and the “calls” relation. The most common hierarchical relationships in switching technology are the “consists of” relation and the “responsible for” relation.

2.6 Abstraction

Abstraction is understood to mean reduction to the essential and generalization of the concrete. The abstraction process should provide the marked and major characteristics of systems, subsystems and processes. This is required if a large-scale, complex problem is to be broken down into separate units. Abstraction is therefore essential for modularization and the establishment of a hierarchy.

There are different types of abstraction. Functional abstraction is when an abstraction is made from the way in which an algorithm or process operates. All that is of interest in a functional abstraction is the effect (the what) of an operation command and not its implementation (the how).

If the data is abstracted from the concrete representation as well as the algorithms of the operation, then this is referred to as data abstraction.

Abstract SDL data types meet the criteria for data abstraction.

SDL supports functional abstraction by means of block diagrams and procedures. Block diagrams contain process reference symbols. However, the way in which a process operates is described by the relevant process diagrams.

The procedures used in the process diagrams have an effect when executed. A user who uses a procedure is only interested in the effect of the procedure and abstracts this from its implementation. A procedure is implemented in the corresponding procedure diagram. The system diagram, substructure diagrams and macros also support abstractions in SDL. MSCs support the establishment of hierarchies and abstraction because it is possible to create sub-MSCs.

2.7 Maximum possible FAN IN

A module’s FAN IN is the number of modules it requires for implementation. If a system has a high average module FAN IN, it probably has a good design [14]. A module’s FAN IN is a kind of measure of its reusability. A high FAN IN indicates good abstraction and rational generalizations.

However, maximum FAN IN should not be bought at any price. Modules with random (i.e. low cohesion) tend to have a high FAN IN. Modules with a high FAN IN must also have high cohesion.

2.8 Acceptable FAN OUT

The FAN OUT is the number of subordinate modules immediately required for the implementation of a module [28]. Very high and very low FAN OUT figures generally indicate poor modularization.

A reasonable FAN OUT should be in the range 7±2 [28].

The elements in the upper hierarchical layers of a system design generally have a higher FAN OUT, while the elements in the lower layers have a higher FAN IN.
2.9 Information Hiding

The secrecy or black-box principle, a criterion for the formation of modules proposed by Pamas. The implementation of a module must remain hidden from the user. Interest lies solely in WHAT a module does rather than HOW it does it. Modules may only intercommunicate by means of well-defined (i.e. completely and unambiguously specified) interfaces.

Information hiding is very closely related to the principle of abstraction.

2.10 Proximity

All information about an object (module), such as remarks, should be in the immediate vicinity of the object. According to DeMarco [8], a module's comprehensibility is inversely proportionate to the number of fingers needed to read a module.

Proximity is promoted when programming by using structured control elements (while, repeat etc.) and by avoiding gotos.

High cohesion requires compliance with the proximity principle.

2.11 Uniformity

Uniform module structures (subsystems, processes, procedures) and standardized naming conventions are essential for team work, reusability and transparency.

For this reason, uniform conventions should be observed when generating SDL diagrams (arrangement of text symbols, positioning of the process start symbol, etc.).

Uniformity can only be achieved through a disciplined approach.
3 SDL and MSC concepts

Chapter 3 describes the SDL and MSC concepts and the related language tools in accordance with ITU standards Z.100 and Z.120.

The full scope of the language of Z.100 and Z.120 is not covered within the current version of SICAT (V1602).

The following SDL constructs are not supported by SICAT:

Non-delaying channels, continuous signals, enabling conditions, dynamic process instancing, options.

By breaking a system down into blocks (subsystems) SDL allows a “basic” static system structure to be modeled.

The blocks are assigned SDL processes. SDL regards a system as a set of parallel and asynchronous processes.

An SDL process is an extended finite machine (cf. Figure 3-19) that communicates with other system processes or with the system environment via communication pathes (channels and signal routes).

All processes within an SDL system act on the same level even though they can be assigned different blocks.

The dynamism of an SDL system is defined by the behavior of the processes it contains and any inputs from the environment.

Message Sequence Charts allow the chronological interchange and flow of information between system components and with the environment. Because MSCs can only be used to show certain of a system’s communications scenarios, MSCs do not generally represent a complete system specification.

MSCs can be used in the Analysis and Design phases within the development process. However, MSCs can also be used to represent test scenarios for the implemented system and for a graphical animation of the behavior of components.

A SDL system principally obeys client/server architecture. Blocks and processes can request and provide services among each other. If a server requires services of another server, it is also a client.

The implicit client server principle means that SDL can be used to describe client/server structures simply.

SDL and MSC support a top-down procedure for system modeling (see Figure 3-1). Beginning with the system diagram, which represents a very abstract system view, the various system components are broken down into subsystems (blocks in SDL terms) using substructure diagrams, until a block only contains processes.

Decomposition into blocks only describes a system in static terms. In contrast, the processes describe the system’s behavior. They can be further structured by means of procedures.

Message Sequence Charts can be used on all decomposition levels in order to describe the relevant communication behavior.

MSC can be used to describe the global communication behavior of system components (external communication with the environment) and the internal communications behavior of their subcomponents (see Figure 3-1).
The following section explains how SDL and MSC can be used to describe static and dynamic system features.

### 3.1 System and communication structure

The starting point for SDL system modeling is the system diagram. This represents an SDL model in its most abstract form and delimits it within its environment. Everything contained in the system diagram is part of the system and requires further modeling, while everything outside the system diagram is part of the environment and is regarded as a black box.

The system diagram is the top level of an SDL specification. It indicates the subsystems that make up a system and the channels used by the subsystems to interchange data with each other and with the environment.
Figure 3-2: SDL system diagram

The Ges_Sys system shown in Figure 3-2 consists of three subsystems, A, B and C. The channels that link the subsystems together or with the environment must have a name. Channels can be unidirectional (e.g. channel C1) or bidirectional (e.g. channel C2) and delaying (C3) or non-delaying (C2). Delaying channels have their arrow heads (indicating the direction) in the middle of the channel, while the arrows for non-delaying channels are at the ends of the channels. The arrival of signals running on delaying channels is delayed for an indeterminate time span.

All signals used in a system must be defined. The signals are either defined at the point at which they are used, or higher up in the hierarchy. The definition of signals and data types is explained in greater detail in section 3.5 (data definition, data structuring and data modeling with SDL). Signals carrying user data are referred to as messages.

The blocks of the system diagram can be further refined. Blocks that in turn contain blocks are shown in a substructure diagram.

Figure 3-3: SDL substructure diagram

Figure 3-3 shows the substructure diagram for the subsystem_C block. It is necessary to define clearly with which channels of the superordinate diagram the channels of a substructure diagram are linked. For example in Figure 3-3 substructure channel K3 is linked with channel C5 of the system diagram.

The blocks of a substructure diagram can in turn contain blocks. This results in a tree-like system structure. Blocks that are not further refined by more blocks contain processes (see Figure 3-4). Blocks that consist of processes are described in so-called block diagrams. Unlike system and substructure diagrams, block diagrams can also express dynamic aspects because of the possibility of creating processes dynamically. They will be dealt with in greater detail in the next chapter for this reason.
As a refinement of a “full page block”, a process interaction diagram normally only contains processes. However, as well as the processes, SDL also allows a substructure diagram to be created within a block diagram (so-called “combined block specifications”) [1].

This means that SDL can look at a block from two different views. The processes describe the behavior, while the substructure describes the implementation structure.

However, Braek and Haugen advise against using this option as it can make specifications too difficult to understand [4].

Figure 3-4 shows the hierarchical arrangement of the components of an SDL specification. Such tree diagrams are not contained in the SDL language description.

Figure 3-4: Tree structure of an SDL system specification

In order to make the various SDL diagram types - system diagrams, substructure diagrams and block diagrams - easier to understand, Hogrefe introduces more precise names [12]. Hogrefe refers to diagrams in which blocks communicate with each other as block interaction diagrams and those in which processes communicate with each other as process interaction diagrams.

Block interaction diagrams therefore correspond to SDL substructure diagrams, while process interaction diagrams correspond to SDL block diagrams. The SDL system diagram is therefore a special block interaction diagram.

The behavior of the processes is described under Process diagrams (see section 3.4).

If there is any ambiguity regarding the names of the various diagram types in subsequent parts of this methods manual, the terms block interaction diagram and process interaction diagram are used.

3.2 Dynamic process creation

Block interaction diagrams enable the static structure of a system and the static interchange of messages between system components to be described. Dynamic aspects such as the creation and scheduling of processes during run-time or the representation of the interchange of messages between system components in terms of time cannot be represented in this way.
The behavior and dynamics of an SDL system is defined by its processes. SDL blocks that are not subdivided into (sub)blocks contain the SDL processes (see Figure 3-4). The description of the processes comprising a block and how the processes communicate with each other and with their environment is contained in the process interaction diagrams.

Figure 3-5 shows the process interaction diagram for the subordinate-system_1 block of the subsystem. C block interaction diagram. The subsystem_1 block contains the three process types P_1, P_2 and P_3. The behavior and mode of operation of a process type are defined in the process diagram (section 3.4).

**Figure 3-5: SDL process interaction diagram**

*Process types* are modeled when designing with SDL. Process types are like templates that are used to generate the instances of the relevant type.

**Figure 3-6: Example of instantiation**

The defined process types only become concrete process instances when the system is running. Each process instance is an individual version of a process type. This means that all instances of a type behave in the same way but differ in their local data. The instances of the “Runway” process type in Figure 3-6 can differ in the type of runway (take-off, landing or take-off and landing) and the length of the runway.

A process instance can be created in two different ways in an SDL system. Either *statically* when a system is started or *dynamically* by another process instance while the system is running. A process instance can be assigned current parameters (generation parameters) upon process start. These parameters determine the type of the instance.
When an SDL process type is defined, it is possible to specify the number of instances of this process type that are active when the system is started and the maximum number of instances that can be active simultaneously while the system is running. If this information is omitted, then exactly one instance of the process type exists when the system is started and the maximum number of instances is unlimited.

Process types P_1 and P_3 in Figure 3-5 each have an instance at the system start. In addition, no more than one instance may exist for either process.

Process P_1 generates process type P_2 instances while the system is running (Figure 3-5). There is no process P_2 instance at the beginning of the system. Up to 10 process type P_2 instances can exist simultaneously while the system is running.

A dynamically generated SDL process may itself create new instances. A process instance in the system only ends if a state transition to a process stop symbol occurs in its process diagram (see section 3.4). Dynamically generated process instances must be located in the same block as the creator process.

The block diagrams (or, more precisely, process interaction diagrams) show the process types that make up an SDL subsystem, which instances already exist at the beginning of the system, which can be generated dynamically while the system is running, and the maximum permitted number of a given type of instance.

In block diagrams the processes communicate with each other or with their environment by means of so-called signal routes.

Signal routes have similar semantics as channels. The difference between a channel and a signal route is that signals and signal routes cannot be delayed. The signals that are transported on a channel linked to a signal route must be identical to the signals on the signal route.

All processes are completely equal in importance in an SDL system. There are no hierarchically higher or lower processes. The fact that processes may exist at different levels of abstraction within an SDL system does not influence their rank in the system. Even a process that is created by another process has the same rank as its creator.

3.3 Message Sequence Charts

The dynamics of a system are not only defined by the creation and termination of processes alone, but also, primarily, by the sequence of the interchange of information between the system components.

The SDL system diagram, SDL substructure diagrams and SDL block diagrams only allow the information flow to be represented in static terms. The chronological progress of the flow of information, i.e. the sequence of the receipt, processing and transmission of messages cannot be shown in block interaction diagrams or process interaction diagrams.

Message Sequence Charts (MSC), which are closely related to SDL, supplement SDL at this point and make it possible to model chronological communication processes.

"Sequence Chart" is the generic term for all graphical notations of such communications processes. There are several different types and variations of sequence charts, however these only differ slightly in their basic notation. The description in this manual applies to Message Sequence Charts which, like SDL, exist in a form standardized by CCITT (Recommendation Z.120) [7].
3.3.1 MSC base elements

MSCs describe communications scenarios between concrete system components and not between types.

According to Z.120, subsystems and instances can be represented in two different ways, as vertical axes or vertical columns. Horizontal arrows represent the messages exchanged among the instances or between instances and the environment. Messages can overtake each other or cross each other (see Figure 3-7). The beginning of the arrow represents the sending of a message, while the point of the arrow represents its processing (the consumption of the message). There is no special symbol for the receipt of a message. Messages can carry parameters. The environment is represented by a frame that encloses each MSC (in other notations the environment is represented by a separate environment axis). The instance header symbol does not mean the creation of an instance, but rather the beginning of an instance description. The instance end symbol does not represent the termination of the instance, but rather the end of the description of an instance within an MSC.

There is no global time within a Message Sequence Chart. Each instance axis has a separate time view. The time “elapses” from the header to the end of an instance. This means that MSCs can be used to represent chronological sequences of messages.

An instance can also send itself messages (e.g. message s9 in Figure 3-7).

![Figure 3-7: MSC base elements](image)

MSCs can be used at different levels of the system design. Instance axes of an MSC diagram can therefore represent both SDL blocks and SDL processes. The MSC diagram in Figure 3-7 consists of the three instances subsystem A, B and C, which represent blocks. Subsystem B and C are instances of the Remote_Station block type. Subsystem_A is an instance that is not based on a type.
Figure 3-8: Dynamic generation of instances within an MSC

The execution of actions and the dynamic creation of process instances can be represented in Message Sequence Charts. In Figure 3-8 two instances of the subscriber process type are created, only to be terminated after a certain period.

Axes and columns of an MSC correspond to real instances of system components (blocks or processes).

MSC provides timers for monitoring the timing of events. SICAT uses a stylized hourglass symbol to indicate timers. The progress and resetting of a timer can be parametrized with the duration.

In Figure 3-9, timer T1 elapses after two time units and timer T2 is reset after the s5 signal is received.

Timers in MSCs use the same semantics as timers in SDL. The SDL timer concept is explained in section 3.4.6.

Figure 3-9: Using timers

Comments can be entered in Message Sequence Charts in two different ways, using the comment symbol and with Notes (/"text"). Individual elements of an MSC can be assigned a comment using the comment symbol or Notes. Global comments can be integrated in an MSC or submsc diagram using Notes within a text symbol (see section 3.3.2). The init action in
Figure 3-10 has a comment symbol. The note in the text symbol is a comment referring to the whole MSC diagram. The note in the header of the Supervisor instance is a comment for this instance.

**Figure 3-10: Comments in MSC**

MSCs can be used on different levels of abstraction in the system creation process. MSCs enable a system's typical communication scenarios to be represented on a very abstract level (MSC with just one "system" axis), as well as the interchange of information between a group of (SDL) blocks or (SDL) processes. A series of structuring concepts is available to increase the transparency of MSCs. In addition, certain parts and areas of an MSC can be presented in a focused way and examined.

### 3.3.2 Structuring concepts in MSC

At present there are four structuring concepts for Message Sequence Charts [7]:

- conditions
- coregion
- sub-MSC
- composition and decomposition

There are plans to include a macro construct in the next version of the Z.120 Recommendations as another structuring tool for MSC.

**Conditions**

Conditions can be used to express states within an MSC. There are two types of condition:

- global conditions
- non-global conditions

A global condition applies to all instances of an MSC diagram. A nonglobal condition only refers to a subset of instances.

Local conditions are a special type of nonglobal condition. A local condition only refers to a single instance. The "nonglob_cond_1" and "nonglob_cond_2" conditions in Figure 3-11 are nonglobal, while the "glob_cond" condition is global. The "nonglob_cond_2" condition only refers to the subsystem_A and subsystem_C instances. A struck-through instance axis in a condition symbol means that the condition does not refer to this instance. The "loc_cond" condition is a local condition of subsystem_B.
Conditions play an important role in the composition and decomposition of MSCs. According to [7], global conditions should be introduced at the beginning and end of an MSC for structuring reasons. Global start and end conditions can be used as links to other MSCs. For example, a global start condition can be used to express the fact that all instances of an MSC share the same starting conditions.

Conditions are a pure generalization of the states that SDL processes can take (see section 3.4). On the one hand, conditions can apply to several different instances, while on the other conditions can also be accepted during transition (change of state within an SDL process).

Coregion

Coregions can be used to discontinue the defined sequence (implicit time axis) of events within an instance. Messages within a coregion cannot be assigned a chronological sequence. Coregions are represented by broken sections of axis. For example, it is not possible to say whether message s2 or message S3 is consumed first in Figure 3-12.

Sub-MSM

An MSC instance can be refined by another MSC. An instance that is refined is indicated by the keyword decomposed (in the instance header symbol). The refining MSC is called submsc and is given the name of the refined instance. In Figure 3-13, the "subsystem_C" instance is refined.
by the "subsystem_C" sub-MSC. The order of the incoming and outgoing messages of the instance to be refined must be preserved in the refining diagram (submsc). In addition, it must be possible to map the external behavior of the refining diagram to the messages of the instance to be refined. The actions and conditions of the refining diagram can be regarded as a refinement of the actions and conditions of the corresponding "parent instance". However, unlike messages, actions and conditions of the refining diagram do not need to meet any other consistency conditions. MSCs therefore permit top down specifications.

**Figure 3-13: Sub-MSC (cf. Figure 4.3)**

**Composition and Decomposition**

Complex MSCs can be broken down into "simpler" MSCs and simple MSCs can be used to form complex MSCs. This requires that there are conforming conditions in each case. Conforming means that the global end condition of an MSC correspond to the global start condition of the other MSC. In Figure 3-14, the part_1_TS_C and part_2_TS_C MSCs merge at the global condition wait_for.resp to produce the composed_TS_C MSC. An MSC can be split up if it contains a global interim condition (decomposition).

Local and nonglobal conditions can also be used (at individual points in an MSC diagram) for composition and decomposition purposes.
Figure 3-14: Composition and decomposition of MSCs based on "global conditions"

Figure 3-15 shows how the two ‘simple’ MSCs `loc_part_1_TS_C` and `loc_part_2_TS_C` form the composite MSC `loc_composed_TS_C`, whereby the `subordinate_system_1` instance of the two simple MSCs are merged at the local condition `wait`.

As a corollary, decomposition is also possible at the local interim condition `wait`.

![Diagram of MSC composition and decomposition](image)

Figure 3-15: Composition and decomposition of MSCs based on "local conditions"

Like SDL, MSC has a textual representational form, MSC/PR (similar to a programming language) as well as the graphical form (MSC/GR). The SICAT tool allows GR to be transformed into PR (for SDL and MSC).

Because MSCs do not normally allow the entire communications behavior of components or processes to be shown, but only selected communications scenarios, MSCs do not generally describe the behavior of a process in full. The complete process is described in process diagrams.

3.3.3 SICAT differences and additions in comparison with Z.120

SICAT differs from the standard at certain points in its description tools:

- instances
- comments
- messages
- connections

The ways in which SICAT differs from MSC standard Z.120 and SDL standard Z.100 are described comprehensively and concisely in appendix 6.4. only the main SICAT differences and additions are dealt with in detail here.

**Instances and comments**

In the case of instances, SICAT supports a representation with an invisible axis in addition to the axis and column representation familiar from Z.120; in this case only the instance header and the end of the instance are visible.
Comments can only be integrated in SICAT as notes and not in explicit comment symbols (see section 3.3.1).

**Messages**

SICAT enables four different message types to be represented: asynchronous communication, remote communication, remote procedure call and inband communication. In contrast, the MSC standard only allows asynchronous messages to be represented. In addition, SICAT allows *optional* messages to be represented (small “o” at the start of the message).

![Figure 3-16: SICAT message types](image)

*Asynchronous communication* is the norm in an MSC. The instances can receive messages at any time and operate in parallel with each other after a message has been sent or received. In the case of *synchronous communications*, the transmitter sends a message with a job to the receiver. The latter handles the job and is blocked for other messages and jobs during this time. The transmitter is blocked until the receiver returns an acknowledgment (the result) after the job is complete. *Remote Procedure Call (RPC)* is a special case of synchronous communication. In an RPC, the sender calls a procedure on the receiver’s side and waits for the procedure to be executed. *Inband communication* means sending information (e.g. dial tone) in an existing connection.

**Connections**

SICAT provides a user with so-called connections in order to group messages and timers and in order to represent causal relationships between them. SICAT makes a distinction between *ordered* and *unordered* connections. Ordered connections reflect the order in which messages or timers are received or dispatched. Unordered connections leave the chronological order open in accordance with the coregion construct in Z.120.

![Figure 3-17: Example of an ordered connection](image)

Ordered connections are represented by a continuous vertical line (see Figure 3-17), while unordered connections are represented by a dotted line (see Figure 3-18).
3.4 Process behavior

Message Sequence Charts usually only enable snapshots or certain scenarios of components or systems to be described. Sequence charts can only be used to represent the complete behavior of only the most trivial SDL processes. SDL provides process diagrams in order to describe the complete behavior of a process.

Some SDL process concepts are explained below, as well as an outline of how SDL supports these concepts in the process diagrams:

- SDL process as a finite state machine
- SDL process diagram
- procedures and macros
- input behavior of SDL processes
- addressing of SDL processes
- SDL timer concept
- non-determinism in SDL
- SDL wildcards

3.4.1 SDL process as a finite state machine

Through the breaking down of a system into blocks and sub-blocks and the definition of channels and signal routes SDL enables “basic” static system structures to be modeled. The complete dynamic system behavior is defined by the SDL processes. The SDL processes are assigned to the “elementary” blocks (blocks that are not subdivided into sub-blocks). Each process is described in a separate process diagram.

An SDL process consists of a finite set of defined states. It can receive messages in each state. The receipt of an expected message causes a transition in states. After each transition in state, a process is either returned to a defined state or terminated. The new state can also be the original state. A process can send messages to other processes during a transition in state. In SDL a process is always in waiting a particular state or in a transition in states.

An SDL process behaves like a state machine. State machines are referred to as state overview diagrams in SDL. Figure 3-19 shows a state machine. Each machine is in an initial state to begin with which is marked in a special way (e.g. by an incoming arrow or source). If message S2 is received in state Z_1, the machine (depending on local variables) enters either state Z_2 or Z_3. Message S5 is output when the state changes to Z_3. No message is output when the state changes from Z_1 to Z_2. If the machine receives message S3 in state Z_1, it then enters state Z_2. No message is output in this case either. If the machine receives message S3 in state Z_2, then it terminates.

Figure 3-18: Example of an unordered connection
The functionality of a state machine can also be shown in SDL process diagrams (see Figure 3-20). However, it makes sense to clarify the input/output behavior of a process with a state machine before creating process diagrams.

State machines are not covered in the SDL Z.100 Recommendations.

![State overview diagrams](image)

**Figure 3-19: State overview diagrams**

### 3.4.2 SDL process diagram

The behavior of an SDL process is shown in an SDL process diagram. Each state of a process has one state diagram within a process diagram which contains all transitions permitted in the state. Figure 3-20 shows four state diagrams, one each for states Z_1, Z_2 and Z_3, and one for the transition from start to Z_1.

An SDL process instance only enters the start state once, i.e. immediately after a dynamic *create* or when the system is started. Creation and system start provide the impetus for a transition from start state to initial process state.

Initializations can take place while changing over from start state to initial state. Several possible initial states can also exist. In this case, the initial state of a process depends on the values of variable evaluated during start transition.

All of the state diagrams of a process are shown in a process diagram. The state diagrams of a process are enclosed in a frame. The frame provides a visual demarcation from other processes.

Each individual state diagram describes all possible procedures, beginning with the present state and going on to the subsequent state. The state diagrams are therefore describing the dynamic of a process. They show which messages are expected in a particular state, which transitions in state cause them to be received and which messages are sent to other processes. More than just messages can be sent during a state transition. Rectangles (task symbols) are used in Figure 3-20 to show that a process can carry out activities (manipulation of local data, initializations) during a transition.

If the SOD process from Figure 3-20 receives message S2 in state Z_1, it is not obvious whether Z_2 or Z_3 will be the next state. This is decided by interpreting the decision symbol.

Process diagrams can also be identified as a single state diagram in which identical state and nextstates are "stuck together". However, this is only a good idea for very small process diagrams. On the other hand, a separate state diagram for each state enhances transparency.
Figure 3-20: SDL process diagram for the state machine in figure 3.19

As a result of a transition a process can:

- execute internal tasks
- take decisions in accordance with conditions,
- output messages to other processes or to the environment,
- create new process instances,
- terminate itself.

SDL processes express more than “normal” state overview diagrams:

- within a transition it is possible to take decisions about the next state in accordance with local process data,
- the normal chronological sequence for receiving messages can be changed by buffering messages,
- a process can create a new parallel process.

Because of these properties, an SDL process is referred to as an extended finite state machine. The process diagram in Figure 3-21 defines the behavior of process P_1, which was introduced in the Subordinate_system_1 block diagram (see Figure 3-22). The octagonal elements in the block diagrams are references to process diagrams. The signal routes of a block diagram (process interaction diagram) correspond to the messages in the input symbols of the relevant process diagrams in their incoming and outgoing messages. The dynamic creation of a process is shown in the block diagram and the corresponding process diagram.
3.4.3 Procedures and macros

Procedures allow an SDL process to be given a more detailed structure and made more transparent. SDL procedures can be given formal parameters that are replaced with current parameters when called. Process $P_1$ in Figure 3-21 executes a `count` procedure in state $Z_2$ when signal "a" is received.

In SDL 88 it was only possible to carry out a procedure call by means of a procedure call symbol as part of a transition. This made it very awkward to show value returning procedures. It was necessary to create constructions with $\text{in/out}$ parameters (see Figure 3-23) [10].

SDL procedures recognize two types of transfer parameters: $\text{in/out}$ and $\text{in}$ parameters. The value of a current parameter whose corresponding formal parameter is labeled $\text{in/out}$ is redefined each time its formal parameter changes value (call by reference).

If a formal procedure parameter is identified as an $\text{in}$ parameter, then a local variable is created with the name of the parameter. The value of the relevant current parameter is assigned to this variable when the procedure is invoked. Changes to the values of the $\text{in}$ parameters do not influence the current argument (call by value).

Figure 3-24 shows the `count` procedure of process $P_1$ from Figure 3-21 is defined and called in a conventional manner (SDL 88).

As well as conventional procedures, SDL 92 also recognizes procedures that return direct values (value returning procedures). Figure 3-24 shows how the `count` procedure is defined and called as a value returning procedure.
Procedures were only known within a process scope in SDL 88. However, SDL 92 (as an extension of SDL 88) also allows a process to call procedures from other processes. Such procedures must be identified with the keyword `remote`. As well as procedures, SDL also provides macros as another structuring mechanism. However, macros should only be used extremely restrictively [4]. According to Z.100, macros can have any number of inputs and outputs. However, when it comes to the use of macros, SICAT requires that a macro may have no more than one input and no more than one output.

In contrast, procedures only have one input and one output. On the other hand, macros are not a semantic unit that can be understood in themselves. Macros are simply a graphical replacement that can only be understood if they are expanded in the relevant application (see Figure 3-25). Unlike macros that apply throughout a system, procedures have a scope. Procedures are units whose semantic structure can be examined independently, i.e. without a context. As separate semantic units, procedures represent important structuring tools for processes. Procedures can have local variables and states that are not apparent outside the procedure. The scope of procedures is expressed by means of the procedure reference symbol (scope symbol) (see Figure 3-21).

SDL procedures have roughly the same symbol set as SDL processes. The only differences are in the start and termination symbol. The counterpart of the process start symbol in a procedure diagram is the procedure start symbol. The process stop symbol corresponds to the return symbol within a procedural description.
The SDL philosophy supports the modeling of event-oriented systems. An SDL process (and therefore an SDL system) never activates itself. A process within a system only operates on the basis of a direct impetus. The activation of a process can always be traced to an impetus from the environment. SDL systems must always receive an external trigger in order to become active. In principle one external impetus is all that is required. A system can then operate for an unlimited time under its own dynamic. However a system will normally be in continuous contact with the environment and will interchange data with it.

The environment is regarded as a black box about which the only thing known is that is behaves (almost) like an SDL process. Unlike an SDL process, the environment can also initiate activities independently and spontaneously.

A process instance that runs dynamically to a process end symbol is not in a waiting or locked state, but actually physically no longer exists in the system.

### 3.4.4 Input behavior of SDL processes

SDL processes are *event-oriented* processes. An SDL process waits in a defined state for valid (legal) signals to be received. If a legal signal is received, then the process executes a transition and enters the next state.

**Input and Save**

All signals listed in the input symbols in a particular state are permissible signals for the process when it is in this state. In Figure 3-26, signals s1, s2 and s3 are permissible for the Supervisor_1 process in state Z_1. A process only reacts to the signals that it is explicitly permitted to receive in the relevant state. Other incoming signals are ignored.

Each SDL process and each process instance is assigned an input queue. This queue accepts each incoming signal. The queue is necessary because SDL processes communicate with each
other asynchronously. Asynchronous communication means that a process can receive messages at any time. The messages received during a transition are also placed in the queue.

The queue is normally processed according to the First In First Out (FIFO) strategy, however processing according to priority is also possible, although priority-based processing of a queue is very dependent on the implementation.

Each process takes the first signal from its queue and checks whether a transition can be carried out in the present state. If this is the case, transition takes place, otherwise the signal is removed from the queue without effect and the process remains in the relevant state.

The save symbol enables incoming messages or symbols to be retained in the input queue. This influences the chronological processing of incoming messages and insures against the loss of messages that are not permissible in the current state but will be required in a later state.

Figure 3-26: The use of input and save symbols

The *Supervisor_1* process in Figure 3-26 can take states *Z_1* and *Z_2*. If signals *s1* or *s2* are received in state *Z_1*, then the process executes a transition and enters next state *Z_2*. An instance of process *P_1* is created during this transition and message *s4* is output.

If signal *s3* is received in state *Z_1*, then Task_1 is executed and the process enters state *Z_2*. If signal *a* is received in state *Z_1*, then it is not discarded, but rather kept in the queue until it can be used in state *Z_2* as a legal signal.

If the *Supervisor_1* process is in status *Z_2* and signal *a* is received, then the process carries out a transition. It executes Task_2 in the process, transmits signal *b* and is subsequently in state *Z_1*. If signals *s1*, *s2* or *s3* are received in state *Z_2*, then these are not discarded, but are saved by means of the save symbol, so that they can be used in status *Z_1*.

If a signal is to be saved in several states, then it must be saved explicitly in a save symbol in each required status.

Continuous signal and enabling condition

A transition is normally activated in SDL by the receipt of a signal. However, SDL also provides two constructs in which the activation of transitions depends on the fulfillment of conditions: continuous signal and enabling condition (see Figure 3-27).

A continuous signal has a condition that activates a transition when it is met. Continuous signals are only evaluated if there are no signals in the input queue.

An input signal is linked to a condition in the enabling condition. The input signal is only triggered if the condition is met. The signal is treated as if it were a save symbol as long as the condition is not met.
Figure 3-27: Continuous signal and enabling condition

The constructs that define the input behavior of processes can be used in any combination (see Figure 3-28).

Figure 3-28: Input, enabling condition, continuous signal and save

Continuous signals and enabling conditions are constructs that go beyond the pure event orientation of SDL. SDL models can also be described without continuous signals and enabling conditions without losing their expression. However, it must be accepted that the effort involved in signaling can increase dramatically, making the SDL diagrams very difficult to read.

Nonetheless, explicit signaling normally makes subsequent implementation easier.

It certainly makes sense to use continuous signals and enabling conditions at many points, however it is necessary to weigh up from application to application whether continuous signals and enabling conditions would water down SDL's event-oriented philosophy.

3.4.5 Addressing SDL processes

In the final analysis, processes in an SDL model are always sources or sinks for signals or messages. SDL offers three addressing options for communications between processes:

- **indirect** addressing
- addressing with **TO**
- addressing with **VIA**

Figure 3-29: Addressing options in SDL

Neither the destination nor the route to the destination are specified in *indirect* addressing, but rather the system structure defines the destination process. The destination process is specified explicitly when addressing with *TO*, while the route to the destination is specified when addressing with *VIA*. 
**Indirect addressing**

If the destination process is obvious from the system structure and from the signals of the channels, then there is no need for explicit addressing (with TO or VIA).

In Figure 3-30, process P_1 sends signals. If P_1 sends signal c, the system structure and/or signaling of the channels make it clear that process P_2 receives signal c. However if process P_1 sends signal b without further details, it is not clear whether P_2 or P_3 is the receiver. In this case, SDL selects a receiver *at random* from the set of possible receivers.

![Figure 3-30: Indirect addressing: Addressing with TO](image)

**Addressing with TO**

Addressing with TO is the norm in SDL. It can be used if the name or address of the receiver process is known. Each process instance in an SDL model has an address that is unique within the system. However, this address is not assigned by the system’s creator, but is instead generated while the model is executed. The SDL data type of a process instance address is predefined as \( \text{PId} \) (process identification).

The following expressions return PId type values:

- **SELF:** A process’ own address.
- **PARENT:** The address of the process instance that dynamically created the instance under consideration. If a process instance was not created directly but already existed when the system was started, then the PARENT expression returns the value zero.
- **OFFSPRING:** The address of the process instance last dynamically created by the instance under consideration. OFFSPRING returns the value zero if the (parent) process has not yet dynamically created a process instance.
- **SENDER:** The address of the process instance from which the last input was received. SENDER returns the value zero if the process instance under consideration has not yet used a message.

In Figure 3-31, the TO construct is used once with a PId and once with a process name. TO can only be used with a process name if the transmitter and receiver process are in the same page block. Process names outside a page block cannot be seen by a transmitter process.

![Figure 3-31: Addressing with TO](image)
Addressing with VIA

The VIA construct can be used to specify the path on which a message is to be transported by the system until it reaches the destination process or environment. Figure 3-32 shows how the use of VIA enables the receiver of signal b sent by process P_1 from Figure 3-30 to be uniquely defined.

![Figure 3-32: Addressing with VIA](image)

The VIA ALL construct (introduced in SDL 92) can also be used to represent multicasting in SDL. In Figure 3-33, VIA ALL expresses that process P_1 sends signal b on the path defined by signal routes R3 and R4 and by channel K. If a VIA ALL path has more than one channel or signal route instance, then a separate signal instance is created for each of these path instances. Thus, signal b is sent to all current instances of process P_3 in Figure 3-33.

![Figure 3-33: Addressing with VIA ALL](image)

However, a system developer should not be unduly concerned about the paths used when sending messages, but rather his models should explicitly specify for whom the message is intended [4].

Figure 3-34 contains an overview of the addressing options in SDL.
OUTPUT

No destination specified

TO

Process name

PID-expression

VIA

Channels and/or signal routes

ALL

The path defines the destination

Multi-casting

If more than one destination specified, then one is selected at random

Process must be in same block

PID must be known

3.4.6 SDL timer concept

SDL enables time conditions to be specified using the timer mechanism. Timers work like alarm clocks. When their time elapses they emit time-out signals. Thus, a signal does not need to run a cyclical check to see whether a time-out has been reached, but is instead passive until a time-out signal is received.

The SDL and MSC timer concepts are identical. Figure 3-35 shows how the same timer $T1$ can be used in an MSC and an SDL process diagram.

Figure 3-34: Destination specifications in the output symbol [4]
Process A receives a job in ready state, forwards it to process B as a subjob and activates timer T1 with an expiry time of i time units. After this, process A enters wait status. If no acknowledgment is received from process B before the i time units have elapsed, then process A outputs an error message. However, if an acknowledgment is received from process A in wait status, then process A resets the timer (RESET (T1)). After the error message is sent or the timer is reset, process A enters wait status. If a message job is received in wait status during process A, then this is backed up until process A enters ready status and can accept the job.

The semantic structure of a timer can be represented as a finite state overview diagram (see Figure 3-36). A timer can assume two states: inactive and active. A timer is inactive at the start. The SET command activates the timer. The SET command has two parameters. These are the expiry time (absolute time at which the timer is to expire) and the name of the timer. The expiry time of a timer can be defined by the (predefined SDL type) time using the expression NOW. If the duration after which the timer is to elapse to NOW, then the result is the absolute expiry
time. Duration is also a predefined SDL data type and is mainly used in association with timers. If the expiry time is reached in active state, a signal with the name of the timer is written to the input queue of the process that set it and the timer enters inactive status.

The RESET instruction renders an active timer inactive and no timer signal is sent. If a timer has already elapsed before a RESET takes place, then the timer signal is removed from the queue insofar as it has not yet been used. A RESET in inactive state leaves the timer inactive. If an already active timer is set again, the timer remains in active status, but with the modified expiry time.

NOTE: Like all other signals, a timer signal can be placed in the input queue (FIFO). An emitted timer signal is then only processed immediately if there is no other signal in front of it in the queue.

The state of a timer can be determined using the expression ACTIVE. Several timers can be active at the same time.

![Diagram of a timer finite machine](image)

**Figure 3-36: Timer as a finite machine**

### 3.4.7 Non-determinism in SDL

With effect from SDL 92 it is possible to show non-determinist behavior with SDL.

![Flowchart of non-determinism](image)
Figure 3-37: Non-determinism in SDL

Figure 3-37 shows the constructs used to express non-determinism in SDL. A spontaneous transition enables a transition to be initiated in a non-deterministic way. This is expressed by none in the input symbol. A state symbol can be followed by any number of spontaneous transitions.

The Anyvalue operator returns a random value of a specified data type. The non-deterministic decision is based on a special application of the Anyvalue operators. “Any” in a decision symbol corresponds to a Boolean Anyvalue-operator and therefore returns the values “True” and “false” in a non-determinist way.

As a means of expression, non-determinism enables the semantic gap between the real world and a model to be kept small. This is because the real world often behaves in an unexpected and unreliable way [1].

3.4.8 SDL wildcards

Using wildcards allows SDL process diagrams to be made somewhat more transparent. SDL recognizes the asterisk (*) and dash (-) as wildcards. The asterisk can be used in the state, input and save symbols, while the dash can only be used in the nextstate symbol to indicate that the next state is the one that immediately preceded this one.

The asterisk in a status symbol means that the following transition applies to all states of a process.

The asterisk in an input symbol represents all remaining signals that do not occur either in input symbols or in save symbols in a certain state (including timer signals).

The asterisk in a save symbol represents all the remaining signals that are not listed either in input symbols or in other save symbols in a certain state (timer signals are also taken into account).

At this point it will simply be outlined how process diagrams can be made more transparent using wildcards. Appendix B contains examples of all wildcards. SDL wildcards are sometimes also referred to as shorthands.

Figure 3-38: Using wildcards

Figure 3-38 shows how to use the asterisk and dash in status symbols.
3.5 Data definition, structuring and modeling with SDL

All signals and messages in an SDL system (signals with user data) must be defined. Signal definitions are contained in text symbols and consist of signal names and, in the case of messages, the type name of the values that a signal can carry.

In Figure 3-39 processes $P_1$ and $P_2$ are linked together and with their environment by means of signal routes $r_1$, $r_2$ and $r_3$. The signals and types do not have to be defined in the block diagram in which they are used. They may already be introduced on a higher hierarchical level. Likewise, the signal and type definitions do not need to be housed in a single text symbol, but can be distributed over several text symbols.

![Figure 3-39: SDL signals, messages and data types](image)

Signals $u$ and $w$ in Figure 3-39 are not defined in the diagram in which they are used, but rather in a diagram in the hierarchy above it. Signals and messages can be grouped together to form signal lists (signal list $L$ in Figure 3-39 comprises signals $t_1$, $t_2$ and $t_3$). Message $s_1$ is of date type, i.e. $s_1$ acts as a stimulus that carries user data of type date. The process can access this user data during the corresponding transition.

SDL offers a series of predefined types, such as

- **Boolean, Character, Charstring, Integer, Natural, Real, PId, Duration, Time**

and predefined generators for forming

- **Strings, Arrays and Powersets.**

Powerset does not mean the exponential set, but rather a normal set in set theory terms. Variables are defined for process diagrams in the same way as signals are defined in text symbols (keyword decl). Figure 3-40 shows a text symbol that contains a timer declaration as well as variable definitions.

![decl](image)
A user can also use the struct construct to create self-defined structures (comparable with Pascal's record concept). The date type in Figure 3-39 is a user-specific defined structure.

However, a user is not confined to using predefined data types. The SDL data type concept is based much more on the principle of abstract data types (ADTs). The ADT is an extension of the elementary data types (integer and real) used in the classic programming languages. Just as only certain operations are permitted on an elementary data type (on integers, for example fixed point operations), an ADT only permits precisely defined operations on the data structure it represents. This duality principle of data structure and associated operations is a decisive feature of object-oriented software generation.

The ADT is not a defined object type (integer, real) with predefined operations (addition, subtraction), but rather a mechanism that enables users to define their own data types with appropriate operations. Thus, a user could introduce the following ADTs for the "Switching Technology" application area: trunk_group, signal_unit or connection.

Users only uses an ADT by means of its operations and do not need to worry about the details of implementation. This principle of information hiding means that they remain unaware of how operations are implemented.

```plaintext
newtype int_queue
  synonym maxlen Integer = 10; /* Maximum queue */
  literals emptyintegerqueue; /* Empty queue */
  operators empty: int_queue -> Boolean;
         insert: int_queue, Integer -> int_queue;
         remove: int_queue -> int_queue;
         length: int_queue -> Integer;
  axioms empty (emptyintegerqueue) == true;
          remove (emptyintegerqueue) == undefined;
          insert (q) == undefined if length (q) >= maxlen;
          remove(insert(emptyintegerqueue, t)) == emptyintegerqueue;
          remove(insert(q, t)) == insert (remove(q), t) if q <> emptyintegerqueue;
          length (insert(q, t)) == 1 + length (q);
          length (emptyintegerqueue) == 0;
endnewtype
```

Figure 3-41 shows an ADT that represents an integer queue of finite length. The operators keyword defines which operations are possible on an ADT. The queue that represents the ADT can only be manipulated using these operations. The effect of the operation is described by the axioms. Literals are identified values from the set of possible values of the data type.

[4] recommends using predefined types in SDL models wherever possible and consulting a specialist in order to define individual ADTs.

Constants are introduced in SDL using the keyword synonym. In Figure 3-41 the constant maxlen defines the maximum capacity of the queue as 10 integer entries.
Another useful data concept in SDL is the syntype construct. `syntype` can be used to rename existing sorts and to continue to use them under another name, similar to the “Define” statement in C programming language.

In addition, `syntype` and the `constants` keyword can also be used to limit the range of values for a sort.

Figure 3-42 shows how to use synonym, syntype and the array generator.

```plaintext
Synonym and syntype

synonym pi real = 3.14;
syntype whole_numbers = integers; endsyntype;
syntype Counter = integer constants 1:30;
endsyntype;

Array generator

newtype Holiday array (Counter, date)
endnewtype;
```

Figure 3-42: Using synonym, syntype and the array generator

### 3.6 Criteria for generating SDL and MSC models

The aim of an SDL model is to reproduce a part of the real world. To ensure that the semantic gap between the real world and the model does not become too great, inherent structures of the application area must be contained in the model. The standards, naming conventions and generally accepted modular and hierarchical structures of an application area must also be evident in the model.

At ICN WN CS, a switching system is normally broken down into functional areas, while a functional area is subdivided into subsystems or functional units. The relevant model must reflect the same structure. The smallest possible semantic gap between the real world and the model is a fundamental prerequisite for quality, transparency and acceptance.

In addition, the model must also comply with elementary software design principles.

#### 3.6.1 Design criteria for the system diagram

As the highest level of an SDL hierarchy, the system diagram shows the embedding and demarcation of the system to be modeled in the environment. Everything within the system limits is part of the examination and is modeled. Everything outside the system limits is only known to the system through the interfaces and is not part of the model. The environment is regarded as a black box. It behaves (almost) like an SDL process. However, an SDL process only operates on the basis of a direct impetus, while the environment can initiate activities on its own account. Thus, SDL systems must receive an external trigger in order to become active. SDL therefore supports the modeling of open systems.

The SDL system diagram contains at least one block. Therefore it cannot contain processes directly.

In particular, the system diagram has the following functions:
1. To show the connections and interactions with the environment
2. To structure the system on a higher level of abstraction.

In principal, a single channel is enough for the entire interchange of data with the environment. However, it is generally not a good design decision to model only one channel for the connection to the environment.

It is also a sign of a poor design if all communication with the environment involves just one single block (I/O block). Such blocks often cause bottle-necks in the system.

If each block that is the source or destination of environment data is linked directly to the environment, then bottle-necks are avoided, transparency and clarity are increased, and the coupling is improved. The risk of tramp data (i.e. unnecessary data) is also reduced.

The elements of data structures transferred to a channel should belong together in terms of content.

The blocks of the system diagram must reflect a rational system structure and define design decisions, i.e. they define what belongs together in the system and what is to be kept separate.

The blocks must therefore have a high level of cohesion and should comply with the criteria for forming modules and abstraction.

To ensure that each block can be modeled separately (division of labor within the team), the interactions between the blocks must be limited to what is absolutely essential (loose coupling).

The number of blocks in the system diagram should not exceed the 7±2 limit. If the number of blocks makes the system diagram too complicated, the designer must decide whether it makes sense to group blocks together to form a single block by forming a further abstraction level. This block would then contain the grouped blocks as subblocks.

3.6.2 Design criteria for substructure diagrams (block interaction diagrams)

Substructure diagrams show how a block is broken down into further (sub)blocks. A rational cohesion should be evident in the elements of a block. Elements in a block should not communicate more with elements of other blocks than with each other. The components of a substructure diagram must therefore comply with the proximity principle. This is a prerequisite in order that the substructure diagram should have a high level of cohesion.

The communication relationships between the elements of a substructure diagram should be limited to what is absolutely essential (loose coupling). Thus, all signals reaching a channel must also really be required in the destination block.

Elements in a substructure diagram are often grouped together for performance reasons. Even if the elements of a substructure diagram are grouped together in a diagram for performance reasons only, they should still have a certain cohesion. The developer will certainly have to make compromises at this point.

There are other criteria for the formation of blocks in addition to the general principles (high cohesion, loose coupling) and performance: the mapping of blocks on hardware or software units, mapping on natural functional units (e.g. functional areas or functional units) and the reuse of blocks in new systems.

Blocks are also information hiding units. General services and services that are non-application-dependent may not be integrated together with application-specific services in shared blocks. The encapsulation of application-specific and non-application-dependent services in separate blocks means that horizontal abstraction units are created within a substructure or system.
diagram. The general and independent higher services abstract from the mode of operation of the application-specific services. This means that application-specific services can be easily modified or replaced with others, provided that the interfaces are preserved.

In totality the substructure diagrams (block interaction diagrams) define the structure of an SDL model. To ensure that the entire system remains transparent, the depth (number of refinement levels) and width (number of substructures/blocks on a refinement level) should not exceed the $7 \pm 2$ range.

An SDL model can easily explode due to uncontrolled refinements. If each diagram contains 7 blocks and each block in turn contains 7 subblocks, etc. then the result is the following number of blocks in the system:

- Level 3: 343
- Level 4: 2,401
- Level 5: 16,807
- Level 9: 40,353,607

Parts of substructure diagrams that recur at different points in a system can be defined as macros. Using macros can enhance transparency. However, macros should only be used in a very disciplined way because the semantic structure of a macro can only be fully understood if it is expanded in the appropriate context.

### 3.6.3 Design criteria for block diagrams (process interaction diagrams)

The block diagrams represent the pages of an SDL model. They contain the SDL processes. SDL processes are linked together by signal routes. The principles of loose coupling also apply to the definition of the signal routes and the messages and signals to be transported on them.

All messages exchanged between processes should also really be required by the receiver (no tramp data).

The principles of proximity and high cohesion apply to the processes of a block. Processes from different blocks may not therefore communicate more closely with each other than processes within the same block.

The number of process types within a block should not exceed $7 \pm 2$. According to Figure 3-43 the number of processes (in a system) has a direct influence on the (system’s) complexity. Overall complexity is a product of the external and internal complexity. External complexity is defined by the intercommunication (interaction) of the processes Internal complexity is defined by the internal structure of the processes. If a system (or block) contains a large number of processes, then its external complexity is very high because the effort involved in communication is very great. If, on the other hand, a system consists of a very small number of processes, then external complexity is low, but internal complexity is high. For example, a system with just one process has minimal external complexity and extremely high internal complexity because the entire behavior of the system is expressed by a single process.
A separate process must be defined for each requirement and each independent behavioral aspect. In addition, a process should be created for each pool of shared resources or data that manages access to resources or data [4]. This makes it easier to integrate changes and additions.

A process can only create a new process instance within a leaf block. The limits of a block thus limit the options for forming process instances. Thus, when a system is designed, it should be remembered that the (leaf) blocks represent a scope of process creation.

It may make sense for blocks only to contain one process. If a specific behavioral aspect of a system is to be shown logically or physically separated from the rest of the system, then the associated process may be alone in a block.

Processes can easily be added to a block with just one process if the interfaces are obeyed at the block limits.

### 3.6.4 Design criteria for process diagrams

SDL process diagrams should not be created in isolation.

Before the full behavior of a process is described in a process diagram, it may make sense to illustrate the global behavior of complex processes using a machine (state overview diagram, see Figure 3-19). A state overview diagram shows all the possible states of a process, all permitted transitions between the process states as well as the stimuli (inputs) that cause the transitions and possible outputs that can occur during transitions.

As well as state overview diagrams, message sequence charts can also support a developer in obtaining information about the behavior of processes. Thus, the actions in MSCs directly correspond to the tasks in SDL process diagrams and local conditions in the MSC can be interpreted as process states. Timers in MSC have the same semantic structure as in SDL. This means that a lot of information can be transferred from an MSC directly to an SDL process diagram.

The SDL philosophy is state-oriented rather than action-oriented [4]. For this reason, external states that are visible from the outside should not be expressed by internal variables in an SDL process diagram, but rather by explicit states (see Figure 3-44).

In addition, relevant decisions should depend on different signals and not on parameters (see Figure 3-45).

![Figure 3-43: Complexity and number of processes](image)

A separate process must be defined for each requirement and each independent behavioral aspect. In addition, a process should be created for each pool of shared resources or data that manages access to resources or data [4]. This makes it easier to integrate changes and additions.

A process can only create a new process instance within a leaf block. The limits of a block thus limit the options for forming process instances. Thus, when a system is designed, it should be remembered that the (leaf) blocks represent a **scope of process creation**.

It may make sense for blocks only to contain one process. If a specific behavioral aspect of a system is to be shown logically or physically separated from the rest of the system, then the associated process may be alone in a block.

Processes can easily be added to a block with just one process if the interfaces are obeyed at the block limits.

---

*Figure 3-43: Complexity and number of processes*

---

*Figure 3-44: External states in state symbols and not in decision symbols*
Important decisions should depend on signals and not parameters

Notational conventions should be complied with and procedures should be used so as to increase the transparency of process diagrams. Complex process diagrams (i.e. diagrams with a large number of states and transitions) can be made transparent by using a separate sheet of paper for each initial state and its transitions. All input symbols should be arranged horizontally directly under the (initial) state symbol. Procedures should be created for more complex transitions. Procedures can also be used more than once. All next states should be listed in a horizontal row at the end of the page (see Figure 3-46).

Notational convention in process diagrams

Process diagrams can be made more transparent through the use of wildcards (see section 3.4.6 and Appendix B).

The asterisk wildcard operator is used in the state and save symbols in Figure 3-47. This shows that all incoming signals that have no transition are saved in all states.
3.6.5 Design criteria for MSCs

MSCs can be used at all levels of system modeling, from the definition of requirements to the specification of test scenarios.

MSCs represent the interchange of message exchange between subsystems. Because MSCs cannot be used to represent the complete message exchange on a system, their use must be limited to selected scenarios.

In early analysis phases, MSCs can be used, for example, to show global intercommunication behavior between the system and environment or between the main system components (blocks of the system diagram).

Other possible applications for MSCs are the modeling of scenarios for the orderly setting up and clearance of communications or the examination of critical points within the system.

MSCs can become unmanageable very quickly. The developer has to decide when it is rational to create “wall-to-wall” MSCs.

Message sequence charts should conform to the relevant levels of abstraction (system, block and process level) of an SDL model. New information can be systematically added to MSCs during the course of the system creation process:

- action symbols
- conditions
- timers

This information can be included in SDL process diagrams. Thus, careful creation of MSCs makes it easier to produce process diagrams.

3.6.6 Design criteria for sub-MSCs

The sub-MSC shows the components (subaxes) that make up a parent instance. All incoming and outgoing messages to and from the parent axis must also appear in the sub-MSC.

According to Miller’s Law, the number of axes of a sub-MSC should not exceed 7±2.

Figure 3-47: Increasing transparency with wildcards
MSC and sub-MSC should conform to a section of the SDL model in structure. This means that the parent instance and the associated sub-MSC must be located together with their messages in the basic SDL model.
### 4 Using SDL/SICAT during the ÖN TN development process

<table>
<thead>
<tr>
<th>SDL diagrams</th>
<th>Modeling</th>
<th>Purpose</th>
<th>Design principles</th>
<th>Tool</th>
<th>DMN phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>System diagram</td>
<td>Demarcation from environment</td>
<td>Localization</td>
<td>SICAT-BD</td>
<td>Analysis</td>
<td></td>
</tr>
<tr>
<td>Definition of the components to be modeled</td>
<td>High cohesion</td>
<td>(F Spec Level 0)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Structuring a system on a high level of abstraction</td>
<td>Loose coupling</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Information hiding</td>
<td>Number of blocks</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>7~2 law</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Substructure diagrams</td>
<td>Forming levels of abstraction</td>
<td>Localization</td>
<td>SICAT-BD</td>
<td>Analysis</td>
<td></td>
</tr>
<tr>
<td>Units for:</td>
<td>High cohesion (informative cohesion)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Functionality</td>
<td>Loose coupling</td>
<td>(F Spec Level 1 &amp; Level 2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Reusability</td>
<td>Information hiding</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Performance</td>
<td>Step-by-step refinement</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Width, depth (7~2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Possibly using macros</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Block diagram</td>
<td>Process container</td>
<td>High cohesion</td>
<td>SICAT-BD</td>
<td>Design</td>
<td></td>
</tr>
<tr>
<td>Scope of process creation</td>
<td>Loose coupling</td>
<td>(D Spec)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Number of processes:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>7~2 law</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process diagram</td>
<td>Description of dynamic or behavior</td>
<td>High cohesion</td>
<td>SICAT-PD</td>
<td>Design</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Loose coupling</td>
<td>(D Spec, PD frames)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Possibly using procedures</td>
<td>Implement ation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Observing graphical conventions</td>
<td>(Refining PDs)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Using wildcards</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSC</td>
<td>Typical scenarios</td>
<td>Conformity with the SDL model</td>
<td>SICAT-SC</td>
<td>Analysis</td>
<td></td>
</tr>
<tr>
<td>(for all phases of system creation)</td>
<td>Adding information</td>
<td>(F Spec)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Step-by-step</td>
<td>Design</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Actions</td>
<td>(D Spec)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Conditions</td>
<td>Test</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Timers</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sub-MSC</td>
<td>Detailed scenarios</td>
<td>see MSC</td>
<td>SICAT-SC</td>
<td>Analysis</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Design</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Test</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 4-1: Design with SDL and MSC diagrams**

Figure 4-1 shows the various DMN phases in which the SDL and MSC diagrams can be used and which design principles must be followed when using them. In addition, Figure 4-1 shows the purpose to which the diagrams can be put and which SICAT component supports the creation of diagrams.

The main task of a developer of switching systems is usually integrating new features in a particular version of an existing switching system (e.g. EWSD). Completely new switching systems are only produced at greater intervals (i.e. very rarely). For this reason, the DMN does not deal with the production of concept papers. The Requirement Specification is not dealt with either because it does not contain MSC or SDL components.
The software development process plan and the development manual divide the software development process into demarcated development phases [20]. The results of the various phases are defined and activities required to produce the results are listed. Each phase builds on the results of the previous phases. The quality of the results of the phase is ensured by institutionalized inspection.

The activities and steps in the various phases follow the principle of “step-by-step refinement) (top-down procedure).

The DMN envisages six development phases:

1. Analysis,
2. Design,
3. Implementation,
4. Integration Test,
5. System Test,
6. Deployment

Figure 4-2 shows the Analysis, Design and Implementation phases as a section of the DMN and outlines how the SDL/MSC methodology is used in the process.
4.1 Analysis

Activities that are mainly of significance for Sales and Project management are only touched upon at this point. Our primary concern here is with activities that are important for the developer. The Analysis phase consists of the following (development) tasks:

- defining user requirements (R-Spec)
- drawing up feature requirements and feature sheets
- drawing up the function specification

The Analysis phase must outline (new) product ideas and user/customer requirements in precise terms.

The feature requirements (LMA) are compiled to form a system version.

The design process described in the DMN begins with the creation of the function specification (F Spec). The F Spec describes how feature requirements are embedded in the system structure. Embedding is achieved either by forming new requirements or by adding development-oriented units (subsystems, hardware modules, firmware, SPUs).

The design specification (DSpec) contains a description of how a hardware module or an SPU provides its functions (or services), for example.

Creating the function specification

Because of their complexity, F Specs have a hierarchical structure:

```
System
   F-Spec Level 0
      FA
   F-Spec Level 1
      FG
   F-Spec Level 2
      FU
```

*Figure 4-3: Establishing a hierarchy of systems in terms of functional areas, functional groups and functional units*

F-Spec Level 0 describes how a system is divided into functional areas. Functional areas are broken down into functional groups in F-Spec Level 1 and into functional units within an F-Spec Level 2
Three views of a system are examined within the F Spec: system components, system functions and system applications (see Figure 4-4).

The function specifications describe how a new feature is integrated in a system (see Figure 3-11).

The following sections of the F Specs should be represented with SDL or MSC:

- Implementation/overview (SDL system diagram)
- Implementation/software/functions (SDL block diagrams, message sequence charts)
- Interfaces/overview (SDL system diagram, SDL block diagrams)
- Message sequences (message sequence charts)

**FSpec (Level 0)**

The system architecture is described at its highest level in FSpec Level 0 (subsystem level). The system architecture can be examined in a variety of views. The most important system views are the component view (showing the physical components that make up a system), the function view (showing the functional areas that make up the system) and the application view.

SDL system diagrams and message sequence charts are suitable graphic tools for the FSpec (Level 0). SDL diagrams can be used to show system components on a very highly abstract level with their static interfaces to each other and to the environment. Dynamic communications behavior can be described with MSCs.

**FSpec (Level 1)**

Functional units are the basic building blocks of an FSpec [20]. If a functional area or subsystem become too complex to be described directly with functional units, then they must be further subdivided. Thus, a functional area can consist of function groups (FG).

Function groups that only contain function units are part of FSpec Level 2.

SDL substructure diagrams (block interaction diagrams) and MSCs can be used in FSpecs Level 1 and Level 2.

---

**Figure 4-4: System levels and views dealt with in F Specs**

Three views of a system are examined within the F Spec: system components, system functions and system applications (see Figure 4-4).

<table>
<thead>
<tr>
<th>System levels</th>
<th>Views</th>
<th>System components</th>
<th>System functions</th>
<th>System applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional Area</td>
<td>e.g. LTGB, LTGG, DLU, CP113</td>
<td>Maintenance, Recovery, Traffic measurement</td>
<td>e.g. CENTREX, ISDN</td>
<td></td>
</tr>
<tr>
<td>(FA)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Functional Group</td>
<td>e.g. CP113 I/O</td>
<td>e.g. Traffic monitoring</td>
<td>e.g. Transmission services</td>
<td></td>
</tr>
<tr>
<td>(FG)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Functional Unit</td>
<td>HW modules</td>
<td>SPUs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(FU)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
FSpec (Level 2)
The elements of a switching system are described on the functional units level in the FSpec
(Level 2). Functional units correspond to SPUs (SOLUTION O.N.E), SW subsystems (EWSD)
or HW modules.

The SDL block interaction diagrams can be used in the FSpecs (Level 0 to Level 2) to describe
the static parts (system structure and static interfaces). Message sequence charts can be used
on all FSpec levels to describe dynamic behavior.

The description of a functional unit is part of the Design and Implementation phases.

4.2 Design

The Analysis phase is complete and the development order is available (baseline B200) when
milestone M199 is declared. Baseline B200 is the starting point for the Design phase. Plans are
made in the Design phase about how the development order is to be implemented. In a
continuous and integrated procedure, the Design phase builds directly on the results of the
Analysis phase. In the ÖN TN development process the SDL and MSC diagrams of the F Spec
(Level 2) continue to be used directly in the subsequent design. SDL process interaction
diagrams describe the processes that make up an FU, the signal routes used by the processes
to communicate with each other and the processes that are dynamically created. Dynamic
communications scenarios between the processes or between the processes and their
environment can be illustrated with MSCs.

MSCs can be created where there is complex internal communication within a D Spec.

Important activities in the Design phase that are modeled with SDL and MSC:

- Drawing up the design specification (D Spec) Activity T210 in the DMN
- Drawing up the functional SDL process diagrams (PD scaffolds)
- Coding the subsystem interfaces Activity T220 in the DMN
- Validating the design with the SICAT simulator

4.2.1 Drawing up the D Spec and process diagram scaffolds

SDL block diagrams (process interaction diagrams, see Figure 4-5) can be used as illustrations
within a D Spec. Message sequence charts can be included in the D Spec when intensive
interprocess communication takes place.

Figure 4-5: D Spec SDL block diagram (process interaction diagram)

The functional behavior of the processes is described by means of comments and so-called
functional process diagrams or process diagram scaffolds (see Figure 4-6).
A process diagram scaffold consists of states, events (incoming signals), messages (outgoing signals) and the next states. Thus, a functional process diagram only contains pseudo-transitions.

Procedures which are needed to understand the functional behavior, are also described in the form of process diagram scaffolds.

The PD scaffolds or functional process diagrams are not part of the D Spec. Although they are inspected together with the D Spec at I210, the D Spec only contains a reference to the PD scaffolds.

The process diagram scaffolds are only refined to produce complete process diagrams during DMN activities T220 (coding the subsystem interfaces) and T315 (coding).

Elementary modules are created as compilation units during the design phase. The processes from the SDL process interaction diagrams are distributed among the modules. Either one module is created for an entire SDL or SICAT process or each SICAT drawing board of a process is given a separate module. A state event matrix (SEM) is generated for each process from its PD scaffold. Unlike the PD scaffolds, the SEMs are incorporated in the D Spec.

The DMN [20] recommends that the module structure should be defined in the form of SDL substructure diagrams.

These block interaction diagrams contain an overview of the module structure and become part of the D Spec [18].

Static interfaces are represented by block interaction diagrams.

MSCs are only included in the D Spec in the case of complex communications behavior.

The SICAT system structure represents the structure of a system in a hierarchical overview and becomes part of the D Spec up to process level.

The PD scaffolds are transmitted into the SW DB in the CM system [18].

**SDL/MSC components of a D Spec:**

- Module structure as block interaction diagram
- Process interaction diagrams describe the static communication of the processes in an FU
- Processes are described in the form of PD scaffolds (PD scaffolds are not part of the DSpec, but are only referred to)
- Distributing processes among modules
  - SICAT: Entire process in one module
  - One module for each drawing board of a process
• Generate a state event matrix (SEM) for each process from the PD scaffolds; the SEMs become part of the D Spec
• Block interaction diagrams for the static interfaces
• MSCs in the case of complex internal communication

4.2.2 Coding the subsystem interfaces

Within DMN task T220 the PD scaffolds referenced in the D Spec that are contained in the CM system as SICAT documents are refined by the developer until the exported interfaces (signals, signal types and remote procedures) are fixed.
A program module is generated from the refined PD scaffold with the SICAT generator (CHILL, C, ASS).

4.2.3 Design validation with the SICAT simulator

The SICAT simulator can be used to simulate the SDL designs from the “Design” phase on the basis of the process diagrams. This enables design errors to be detected and eliminated before implementation (see Figure 4-7). In the case of systems using the C development environment, the simulator also allows a test frame to be created for the already implemented components. This means that the modules and subsystems can be tested successively during the development process and after implementation is complete.

You will find some hints on how to use the SICAT simulator for testing purposes below. Because testing large systems requires wide-ranging concepts, only a few fundamental principles are presented within the framework of the MMN.

![Figure 4-7: Design validation with SICAT](17)

4.2.3.1 The principles and methods of systematic testing

As a rule, the systems to be tested are so big that tests must be carried out in stages. If a system is tested in its entirety, then it is difficult to cover a sufficient number of program paths with tests and to pinpoint the errors identified. The tests should be carried out in several stages for this reason:

Module tests
A module is understood as the smallest program unit that can run independently. It can consist of several cohesive procedures. According to the MMN, a module corresponds to an SDL drawing board, which can contain an SDL process or an SDL procedure.
Test scenarios that test process behavior in comparison with the specification in the SDL process diagrams must be created for each module. White box test procedures are suitable for testing modules. These are tests that follow the internal program logic of the processes. The feasibility of reaching states and branches, the execution of conditions, time-out behavior and the transfer of message data together with signals are checked, among other things.

Subsystem tests

After the module tests are complete, the modules (processes) are merged to form subsystems. In SDL systems it is best to follow the system structure from the Analysis or Design phase. The interaction of the processes in each process interaction diagram is tested in an initial step. The cooperation of the functional units from the Analysis phase is checked in the next step.

When the subsystem tests are carried out, the main emphasis is placed on testing the interfaces between the processes or functional units because it is assumed that the module test has already been carried out. Black box tests that are controlled from outside by means of input values (input signals) are suitable for this purpose. The tested behavior is judged on the basis of the output values (output signals) generated. To begin with MSCs created in the “Analysis” and “Design” phases that show a specification of the flow of communications between processes and FUs can be used as test scenarios. It is usually necessary to expand the set of MSCs in order to cover more tests.

System Test

Depending on how the system is structured, the functional units in the Analysis phase are merged on a step-by-step basis and their interfaces are tested. This means that the functional groups are first brought together, then the functional areas to form the entire system.

The principle of the test is the same as for the subsystem test, i.e. the message procedures between the FUs and FAs are tested against existing or new MSCs.

4.2.3.2 Test specification and preparation of the test runs with the SICAT simulator

The way in which test scenarios are specified depends on the type of tests to be carried out. A distinction is always made and depends on whether the test is to be based on the specifications from the Analysis and Design phases (e.g. in the case of module tests) or on the feature requirements (e.g. in the case of subsystem and system tests).

Specifying module tests

In the case of module tests the behavior of the processes is tested on the basis of the SDL process diagrams specified in the design. To limit the number of test scenarios, it is recommended that a selection should be made, for example:

- Execute all transitions (all STATE INPUT combinations). A check is made to see whether the specified NEXT STATEs are reached
- Check that all states and branches can be reached
- Check the possible paths
- Execute all conditions and descriptions. A check is made to see whether the specified DECISION and SELECTION symbols are correctly executed under various conditions.
- Test for correct time-out behavior
- Test the correct transmission of message data together with signals

SICAT offers the following support for creating test specifications:
State Event Table
A file named `name.sta` created by the SDL analyzer. All information about the STATE INPUT and NEXT STATE combinations are contained in the State Event Table.

Cross Reference Table
A file named `name.xrf` created by the SDL analyzer. This contains a table listing all the SDL symbols of a process diagram with type, content of the inscript level and root of symbol. The line and column of the symbol in the relevant PD are also specified.

Root Diagram
This can be created using the “view” menu in the SICAT control program. It contains a graphical presentation of all STATE INPUT and NEXT STATE combinations for a process.

**Specification of subsystem and system tests**

Test scenarios are created for subsystem and system tests that are based on the MSCs and feature requirements created in the Analysis and Design phases. MSCs or TTCN trees (Tree and Tabular Combined Notation [22]) can be used to specify the test scenarios. At least the following test scenarios should be covered:

- All straight-forward scenarios, i.e. sending/receiving all subsystem-internal signals and signals with message data.
- Checking the existence of the transmitter and receiver of a signal
- Checking the existence of deadlocks in the system, i.e. is process A waiting for a signal from process B while the latter is waiting for a signal from process A?
- Checking run-time behavior, i.e. is process A sending a signal to process B that can only be processed at a later point, but that has not been saved?
- Suppressing or delaying expected signals.
- Duplicating expected signals or sending them several times.
- Sending unexpected signals
- Sending unexpected message data

SICAT offers the following support for creating test scenarios:

<table>
<thead>
<tr>
<th>Message sequence charts</th>
<th>In the case of subsystem and system tests the main check involves the correct flow of signals between the processes or subsystems to be tested. The test scenarios are created in the form of MSCs using the MSC editor.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decision continuations</td>
<td>Lists with values for decision continuations can be defined for each process instance in the “Decision Editor” using the SICAT simulator.</td>
</tr>
<tr>
<td>Breakpoints</td>
<td>Lists with breakpoints can be defined for each process instance in the “Breakpoint Editor” using the SICAT simulator.</td>
</tr>
</tbody>
</table>

**Archiving test scenarios, test data and test results**

Test data, test results and test execution must be stored for re-use, particularly in respect of regression tests. It is recommended that an archive should be created with the following content:

<table>
<thead>
<tr>
<th>Test scenarios:</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Name and serial number</td>
</tr>
<tr>
<td>• Description of the aim of the test (what type of errors are to be found)</td>
</tr>
<tr>
<td>• Preliminary conditions (what is the defined state of a process, subsystem or</td>
</tr>
</tbody>
</table>
using sdl/sicat during the on tn development process

system prior to the beginning of the test and how is this state reached; what tests must be carried out before the current test is carried out

- The name and contents of the simulation configuration
- Subsequent conditions (what is the state of a process, subsystem or system after the test has been executed; what data and/or conditions have been changed through implementation).
- Test scenario specification (depending on the type of test, e.g. STATE INPUT NEXTSTATE combinations in module tests or MSCs in subsystem or system tests, etc.)
- Expected results (depending on the type of test, e.g. state reached in STATE INPUT NEXTSTATE combinations; branches passed through in decision continuations; signals received on external interfaces, etc.)

Test data

The pool for test data can contain a variety of data, for example:

- Signals to be fed in
- Lists for determining the continuation of decisions
- Lists for setting breakpoints
- Message data

Archived procedure protocols

With regard to regression tests, the procedural protocols for already completed tests should be incorporated in the archive so that they can be used for comparative purposes in regression tests.

Test sequence

It makes sense to define a sequence for executing a test. In module tests this can be influenced by the fact that, for example, the initial states for certain tests correspond to the final states for other tests. For integration purposes it should be taken into account in the test strategy that, for example, the modules called by the subsystem to be tested have already been tested.

automation of testing

the following options are available for automating testing with the sicat simulator:

feeding expected signals

so-called “stub” processes can be specified so that the signals expected from the processes or subsystems to be tested do not need to be fed using the signal feeder alone. these stub processes simulate non-existent system parts and the environment. they are specified as sdl-pds and contain output symbols that send some or all of the expected signals to the test object, depending on the test specification. it is possible to interrupt the stubs before certain signals are sent by setting breakpoints or inserting signals that are sent by a process to itself. in this case, the stub process must run in “transition” mode. in addition, it is also possible to control the step-by-step sending of signals using “symbol” mode.

stub processes must be included in the archive and must be contained in the simulation configuration.

defining decision continuations

the values for defining branching continuations within the processes can be defined for each process instance using the decision editor and saved in a file. a file of this kind should be created for each corresponding test scenario and stored in the archive. the current file must be located in the same directory as the executable files of the current test suite during execution.

defining breakpoints

for test scenarios in which breakpoints are to be set for certain state input combinations or procedure calls, these break points can be defined using the breakpoint editor and saved in a file. a file of this kind
should be created for each corresponding test scenario and stored in the archive. The current file must be located in the same directory as the executable files of the current test suite during execution.

4.2.3.3 Executing the test using the SICAT simulator

For any tests carried out with the SICAT simulator, all execution steps of the process instances and the protocol for communications between the process instances and with the environment are stored on file. These files are used to check and document the test results and are transferred to the archive as a basis for regression tests. The simulation configuration only contains the process to be tested for module tests and/or the procedure to be tested and the "stub" process required for automation, if any. In the case of subsystem and system tests, all the processes involved (as well as stub) are included in the configuration.

Before simulation begins, a number of parameters must be set in the user's SICATPD.INI profile file [17]. Because the code generator of the simulator generates C code, any existing content of the symbols must be adapted accordingly on code level. If CHILL is used as the development environment, then the processes adapted for the SICAT simulator should be kept in a separate version.

Below you will find some examples illustrating how to carry out special tests with the SICAT simulator:

| Module Tests | Tests in which the feasibility of reaching states, the coverage of paths and the execution of all transitions are checked should be carried out either in "Transition" or "Run" mode. In the former case, all branches are processed on a state-to-state basis. The sequence of tests should be selected in such a way that as few repetitions as possible are required. In the second case, stub processes can send the expected signals and set breakpoints to specific STATE INPUT combinations that are to be tested. Branching continuations with DECISION and SELECTION symbols are tested using the "decision editor". Correct time-out behavior can be achieved by suppressing the signal expected for resetting the timer. In order to test the correct transmission of message data together with signals, user-specific data buffers must be defined in the code level of the TEXT symbol of the process diagrams (for further details see [17]). |
| Subsystem/ System Test | When subsystem tests are carried out using the SICAT simulator, it is necessary to reach a compromise between the number of processes to be tested and the transparency of the entity window shown. You are advised to select “Run” mode. The execution of communications between the subsystems to be tested is already apparent on screen when the tests are carried out; actual evaluation and elimination of errors may only take place after the test sequence is complete, so that the processes are always in a defined processing state. |

4.2.3.4 Evaluating the results

The test results are evaluated on the basis of the saved execution log files. Automatic evaluation (partial) is particularly desirable in respect of regression tests. However, tools must be created for this purpose that interpret the results and compare them with the specified test scenarios. This means that the test specification and execution log files should be available in a form that can be automatically interpreted. The simulator does not offer support during automatic test evaluation in the present version, so that log files have to be checked and compared manually.
Checking the log files
During simulation, log files are created with the execution steps of the process instances, the communication of the processes with each other and with the environment. The log files contain information about the states passed through and the signals expected or actually received with or without message data.

The information from the execution log files for the entire simulation and entity window is compared with the test scenario specifications. Errors or deviations from the specified behavior are documented, corrected and retested using the simulator.

Log files are transferred to the design and archived as design validation documents.

4.3 Implementation
The PD scaffolds, which are extended to include the interface information, are refined within DMN task T315 to produce complete SDL process diagrams. The developer creates "real" transitions for the SICAT PDs in the CM system with pseudo transitions. This means that the PD scaffolds are extended to include SDL control structures (Decision, Task, Process Create, etc.).

Code in the relevant programming language must be added to the SDL symbols of the current process diagrams. More than just "one" programming language statement can be included in a task symbol.

The developer uses the SICAT generator to produce an entire program module from the complete SDL process diagrams.
Summary of F Spec, Design and Implementation:

- System
- Subsystem
- Functional unit
- Block interaction diagrams for the module structure
- PD frames
- Complete process diagrams
- Implementation

Figure 4-8: Summary of F Spec Level 0, Level 1, Level 2, D Spec and Implementation-
4.4 Phase transitions/ consistency

The SDL and MSC diagrams created as part of the DMN development process must be consistent in their refinements (vertical integrity) and must also correspond in its signals and components within a level (e.g. F Spec Level 0) (horizontal integrity).

In the future, a DB connection will enable consistency to be checked using tool support.
Using SDL/SICAT during the ÖN TN development process

4.5 Summary of procedure with SDL/MSC

SDL supports a top-down design method. Entry level for an SDL model is represented by the SDL system diagram. The system diagram demarcates a system from its environment and...
shows it in its most abstract components. Beginning with the system diagram, the system architecture is described as part of a step-by-step refining process using SDL block diagrams (or, more precisely, block interaction diagrams). The blocks at the lowest level contain the elements that carry the system’s activity, the processes. The block interaction diagrams only describe static aspects of a system (the structure as a "consists of relation" and the static interfaces). Selected procedures (scenarios) can be specified alongside this architectural description using message sequence charts (MSCs). The behavior of the processes is modeled with SDL process diagrams. Before the complete behavior of a process is represented using SDL process diagrams, it is best to describe the global input/output behavior of a process by means of a state overview diagram (see Figure 4-11). Information for SDL process diagrams can also be gathered from MSCs. Information can be added successively to MSCs in the development process (integration of conditions, action symbols and timers). The MSC conditions correspond to states, MSC action symbols correspond to SDL tasks and MSC timers have the same semantic structure as SDL timers. This information can be transferred directly to SDL process diagrams.

A further level of semantic refinement can be introduced within an SDL process in the form of procedures.

Horizontal and vertical integrity conditions must be observed in the modeling process (see section 4.4 "Phase transitions/consistency").

Section 3.6, "Criteria for generating SDL and MSC models", explains which principles and design criteria are to be complied with when creating SDL and MSC diagrams.

Figure 4-12 shows the application of SDL and MSC within the DMN development process.

SDL methodology is also dealt with in [4, 5 and 19].
Figure 4-11: SDL engineering strategy: Top down procedure
### R-Spec

Requirements placed on the system to be developed by the environment should be formulated in such a way they are as independent as possible.

| Form a block in the system as a counterpart for each requirement that is responsible for satisfying a requirement |
| Blocks correspond to functional areas on this level |
| Introduce any additional blocks for context knowledge, shared data and shared resources |
| Link the blocks with channels and define the messages and signals for the channels |
| Use MSCs to model communications scenarios between the blocks or between the blocks and the environment |

/* System diagram created with functional areas */

### F-Spec (Level 0)

FA-)blocks as separate systems and repeat the steps above beginning with the formulation of requirements

/* Create block interaction diagrams with function groups */

### F-Spec (Level 1)

(FG-)blocks as separate systems and repeat the steps above beginning with the formulation of requirements

/* Create block interaction diagrams with function units */

### F-Spec (Level 2)

Create a separate process for each requirement/task and define independent behavior aspects in separate processes

Encapsulate shared data or bundles of shared data in separate processes

Define an (administrative) process for each pool of shared resources

Link the processes with signal routes

/* Create process interaction diagrams */

### D-Spec

Use MSCs to describe process communication (in the event of complex communications)

Describe the external communication behavior of a process with a state overview diagram or as a PD scaffold (not part of the DSpec) and/or

Extend MSCs (conditions, tasks, timers etc.) and transfer the information to the SDL process diagrams

/* Create process interaction diagrams */

### Implementation

Use MSCs to describe process communication (in the event of complex communications)

Create procedures for large and frequently recurring transitions

/* Create process diagrams */

Enhance the SDL symbols with programming language statements

/* Process diagrams ready for code generation */

Code generation: CHILL, C, ...

Figure 4-12: SDL/MSC methodology in the DMN
4.6 Example

There follows an example of an SDL specification using SICAT. The example reflects the phases of the development process in chapters 1 to 3 of the DMN and explains how to use SICAT when creating the specification.

Section 4.6.1 contains an overview of the “system for processing sheet metal” example. Section 4.6.2 describes the SDL specification for the example using SDL/SICAT. The various steps to be carried out during the “Analysis” and “Design” phases are dealt with in detail.

4.6.1 Description of a system for processing sheet metal

A system for processing sheet metal consists of two production cells. In cell 1 the metal sheets are pressed and then bent into a particular shape in a bending machine in cell 2.

In cell 1, the sheet metal is fed onto a conveyor belt from where it is picked up by a robot and placed in a press. After pressing, the sheets are transported to another conveyor belt by the robot. The transport arm picks up the pressed sheets from the conveyor belt and places them on a belt in production cell 2. Here the sheets are taken from the belt by another robot and placed in the bending machine, where they are bent to the required shape. After this a robot removes the bent sheets from the bending machine and transports them to a container.

Figure 4-13 shows the structure of the system for processing sheet metal. For simplicity, the figure only shows production cell 1 in detail, while production cell 2 is simply touched upon.

4.6.1.1 General notes on the components of the production cells

The components of the production cells control electric motors and solenoids which execute the actions of the components. The components also require information from sensors, which specify current positions, for example.

![Diagram of the system for processing sheet metal]

Figure 4-13: Overview of the system for processing sheet metal
4.6.1.2 Description of production cell 1

Because of the complexity of the example, only the components in production cell 1 are described in greater detail. Cell 1 consists of the following components:

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F_Belt (feeding belt)</td>
<td>Conveyor belt on which the metal sheets are fed to the system.</td>
</tr>
<tr>
<td>Turntable</td>
<td>Table for the metal sheets that arrive at the end of the F Belt. This supplies the metal sheets to the robot for further processing.</td>
</tr>
<tr>
<td>Robot</td>
<td>The robot removes the sheets to be pressed from the turntable and places them in the press. When pressing is complete, it takes the pressed sheets from the press and transports them to the R-Conveyor.</td>
</tr>
<tr>
<td>Press</td>
<td>Device for pressing the metal sheets.</td>
</tr>
<tr>
<td>R_Belt (removing belt)</td>
<td>Conveyor belt on which the pressed sheets are placed by the robot. The sheets are fetched from there by the transport arm and brought to cell 2.</td>
</tr>
</tbody>
</table>

Figure 4-14 contains an overview of a complete cycle in cell 1. The various components of production cell 1 are then described in greater detail.

**Figure 4-14: Complete cycle in the production cell**

**F_Belt**

The function of the F_Belt is to transport the metal sheets to be pressed to the turntable. The metal sheets are deposited on the belt by the user and transported until a light barrier signals that the sheet has reached the end of the belt. The belt is stopped and the turntable receives
the signal for a sheet to be fetched. As soon as the turntable signals that it is ready for a sheet to be picked up, the conveyor starts again and moves the sheet onto the table.

After the table has registered the receipt of the sheet of metal, the conveyor belt is ready to transport more sheets.

**Turntable**

The turntable takes the sheets to be pressed from the F_Belt and transfers them to the robot. Because the F_Belt and robot are at different heights and angles, the turntable must pivot and its height must be adjustable.

As soon as the table receives the signal to fetch a sheet of metal, it moves to the correct position through a series of vertical movements and rotations and receives the sheet. It then turns and moves to the position in which the robot can pick up the sheet.

The table is only ready to take more sheets when the robot has removed the sheet.
Robot
The robot consists of two gripper arms on a pivot. Gripper1 handles the transport of metal sheets from the turntable to the press. Gripper2 takes the pressed sheets from the press and transports them to the R_Belt.

The gripper arms can be extended or retracted in a horizontal direction because the turntable, press and R_Belt are positioned at different distances from the robot. Each gripper arm also has an electromagnetic gripper that can be used to pick up metal sheets.

When a metal sheet is to be transferred from the turntable, the robot turns until Gripper1 has assumed the required position in relation to the turntable. Next, Arm1 extends to pick up the metal sheet. The robot turns until Gripper2 is positioned in front of the press. Arm2 is extended in order to remove the pressed sheet from the press. The robot now turns so that Arm1 can reach the press and Arm2 can reach the F_Belt. Both arms are extended and deposit the metal sheets in the press or on the F_Belt.

When these actions are complete, the grippers retract to their original extension and the robot returns to its original position (in front of the turntable).

Press
The metal sheets are pressed after the robot has placed them in the press. The press consists of two horizontal plates, the lower of which moves in a vertical direction. The press has three vertical positions for the lower plate: the lowest position for unloading the pressed sheets, the central position for loading the sheets to be pressed, and the top position for actually pressing the sheets. The press keeps the robot informed of the position of the lower plate, so that the robot knows whether the press can be loaded or unloaded.

R_Belt
The pressed sheets are placed on the R_Belt and transported until a light barrier signals that the end of the belt has been reached. A transport arm picks up the sheet of metal and places it on the conveyor belt that will take it to the bending machines in cell 2.

4.6.2 SDL specification for the system for processing sheet metal
The use of SDL/SICAT during the ÖN TN development process is explained below on the basis of a "system for processing sheet metal". Reference is made to the "Analysis" and "Design" phases contained in the DMN. Each of the above-mentioned phases is described in a subsection with the following structure:

In line with the division of phases into functional specifications of different levels, there is a separate section for each specification. The execution of the function specifications is divided into a number of steps which are described in a two-column table. The left column contains the name of the step to be carried out in bold type together with a symbol and a short name for the result achieved in italics. The corresponding right column describes the execution of the steps on the basis of the "system for processing sheet metal" example and offers some information on how to use SDL/SICAT and differences between the SDL standard and the tool. The diagrams for the example created using SDL/SICAT are referred to in italics at the end of each column. The diagrams themselves follow at the end of each section, after the description of the steps.

4.6.2.1 Analysis
The Analysis phase consists of the following: the definition of the user requirements (R Spec) and feature requirements (LMA) and the creation of the feature sheet (LM-B) and function
specification (F Spec Level 0). Because the Analysis process described in the MMN begins with
the creation of F Spec Level 0, it is assumed that the R Spec and LMA are already available in
the “sheet metal processing” example.

**F Spec Level 0**

The division of the system into functional areas (FAs) is described in F Spec Level 0. This
includes system architecture at the highest level.

| Division of the system into functional areas | The system is divided into functional areas in an initial step. This division is shown in the SDL system diagram.

The system in the example consists of two independent production cells 1 and 2 and the transport arm as a connection. AN FA is defined for each cell and for the transport arm.

see 15 “Functional Areas in the system diagram for the system for processing sheet metal” |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>FA 1</td>
<td>FA 2</td>
</tr>
</tbody>
</table>

*SDL system diagram with functional areas*

<table>
<thead>
<tr>
<th>Demarcation of the system within the environment</th>
<th>The task of the components within the production cells consists of controlling the motors (see section 1.1.1). The motors and sensors are regarded as part of the environment. The user who supplies the feed belt with metal sheets is also part of the environment. The environment components are not shown in the SDL system diagram. However, for the static interface description it must be defined which external components exist and what communication takes place between external components and system components.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not represented</td>
<td></td>
</tr>
</tbody>
</table>

*Static interface description*

| Channels and signals in the SDL system diagram | The flow of signals between the FAs and with the environment components must be taken into account for the purposes of the static interface description. The interfaces are represented as channels with signals in the SDL system diagram.

Interfaces are required between the following FAs (the interface names in Figure 4-15 are specified as **(SS name)**): see Figure 4-15 “Channels and signals in the system diagram for the system for processing sheet metal” |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>FA 1</td>
<td>FA 2</td>
</tr>
</tbody>
</table>

| Cell 1 | Cell 1 communicates with all motors that execute the actions of the components within the cell (**SS Z1_MOT**). It also requests the information required to execute the actions from the sensors (**SS Z1_SENS**). It receives a message from the user indicating that a new sheet has been fed in and sends an acknowledgment that the sheet has been received (**SS Z1_User**). Cell 1 informs the transport arm that a pressed sheet is to be picked up with (**SS TA_Z1**). |
| Transport arm | The transport arm sends cell 1 an acknowledgment that the pressed sheet has been received (**SS TA_Z1**) and informs cell 2 that it has placed the sheet on the conveyor belt leading to the bending machine (**SS TA_Z2**). |
| Cell 2 | Like cell 1, cell 2 communicates with all motors that execute the actions of the components within the cell (**SS Z2_MOT**). It also requests the information required to execute the actions from the sensors (**SS Z2_SENS**). Cell 2 sends (**SS TA_Z2**) to the transport arm as an... |
Dynamic interface description

<table>
<thead>
<tr>
<th>Acknowledgment that the sheet has been picked up; (Z2_CONT) forms the connection to the container.</th>
</tr>
</thead>
</table>

As well as the static interface description in the SDL block diagrams, selected dynamic communications procedures between the functional areas are represented with MSCs. Characteristic procedures, such as straight-forward scenarios, are selected on FA level.

see 16 “Straight-forward scenario with a processing cycle in the system”

**Message Sequence Charts (MSC)**

![Message Sequence Chart](image)

**Figure 4-15: System diagram for the “production cell” system**

Figure 4-15 shows how the system is divided into functional areas. The signals and signal lists appear to the right of the block diagram. The signals belonging to a signal list are indented beneath the signal list.
Figure 4-16: MSC for signaling between the functional areas of the entire system

Figure 4-16 shows the MSC for signaling between the functional areas of the overall system. The straightforward scenario is shown in the message procedure. The user transfers a sheet of metal to cell1, where it is pressed and forwarded to the transport arm. The procedure ends with the acknowledgment from cell2 that the sheet has been received. For simplicity, the signaling...
between the components of cell 1 and their sensors is only shown for the feed belt as an example.

**F Spec Level 1**

The functional areas from F Spec Level 0 are further broken down in F Spec Level 1. The FAs are divided into functional groups (FGs) which are represented using SDL substructure diagrams. In the lowest level of refinement, the functional groups of F Spec Level 1 only contain functional units.

<table>
<thead>
<tr>
<th>Division of functional areas in functional groups</th>
<th>FG 1</th>
<th>FG 2</th>
</tr>
</thead>
</table>

**SDL substructure diagram with functional groups**

The functional areas are examined for complexity in an initial step and then divided into functional groups as necessary. For simplicity, the refinement of production cell 1 only is taken into account. Cell 1 consists of the following 5 independent components: feed belt, turntable, robot, press and removal belt. An FG is defined for each component.

*see Figure 4-17 "Functional groups in the substructure diagrams of cell 1"*

<table>
<thead>
<tr>
<th>Static interface description</th>
<th>FG 1</th>
<th>FG 2</th>
</tr>
</thead>
</table>

**SDL substructure diagrams**

As in F Spec Level 0, the static interfaces are described by channels and signals that are sent on the channels. They are represented in the SDL substructure diagram of the “cell 1” FA.

This requires that the flow of signals between the components of the “cell 1” FA should be taken into account. In the case of signals exchanged between components of different FAs, the associated channels must be directed to the corresponding channels in the SDL system diagram.

Interfaces are required between the following components: (The interface names in Figure 4-18 are specified as (SS name)).

*see Figure 4-17 "Channels and signals in the substructure diagrams of cell 1”*

| F belt | The F belt communicates with the motor that controls the movement of the belt (SS ZM). It receives information from a light barrier in order to ascertain whether a sheet of metal has reached the end of the belt (SS ZS). It receives a message from the user indicating that a new sheet has been supplied (SS ZB).
|        | It sends a signal to the turntable indicating that a sheet of metal has been deposited there (SS ZD). |
| Turntable | The turntable communicates with the motors that are responsible for the rotation or vertical movement (SS DMR, SS DMV). It also requires information about the current angle of rotation and the current vertical position (SS DS). It sends a message to the F belt indicating that it is ready to receive a sheet of metal (SS ZD). It informs the robot that a sheet is to be picked up (SS DR). |
| Robot | The robot communicates with the rotation motor (SS RMR) and the motors that are responsible for extending and retracting the grippers (SS RME). It uses (SS RMG) to send control commands to the electromagnetic grippers. It requires information from the sensors about the current angle of rotation and the degree of extension of the arms |
It sends a signal to the turntable indicating that a sheet of metal has been picked up (SS DR). After the press has been loaded/unloaded, the latter sends an appropriate acknowledgment (SS RP). The robot sends a message to the R belt indicating that a sheet of metal has been deposited with (SS AR).

The press communicates with the motor that carries out the vertical movement of the lower plate (SS PMV) and requests information about the current position of the lower plate from the sensor (SS PSV).

The press sends the robot signals via (SS RP) indicating whether the lower plate is in the central position for loading or in the lower position for unloading (SS RP).

The R belt sends the motor responsible for the movement of the belt commands for starting and stopping (SS AM).

It sends an acknowledgment to the robot indicating that a new metal sheet has been received (SS AR) and a message to the gripper arm indicating that a pressed sheet is to be picked up (SS TC_AB).

As in F Spec Level 0, selected dynamic communications sequences between the functional groups are represented with MSCs. Characteristic sequences are selected on FG level, for example straight-forward scenarios, time-out handling and alarm messages.

see Figure 4-18 "Straight-forward scenario with a processing cycle of cell 1"
Figure 4-17: Substructure diagram for cell 1 - division into functional groups
The signalisation of the components and their sensors is represented simpler by the signal "REQ_ANSW_ON/OFF".

Figure 4-18: MSC of a processing cycle for functional groups in cell 1.

The MSC in Figure 4-18 describes the processing cycle for the functional groups in cell 1. The user transfers a metal sheet to the F_Belt, which transports it to the turntable. Here it is picked up by the robot. The robot rotates, unloads the press, places the pressed sheet on the R_Belt and reloads the press. The press closes, executes the process and opens again for unloading.

**F Spec Level 2**

The functional groups of the lowest level of refinement are divided into functional units in F Spec Level 2. Functional units are units that only contain SDL processes.
It is evident in the “Production cell 1” example that some of the FGs must execute identical or similar actions (e.g. the vertical movement of the turntable and press, the rotations of the turntable and robot, etc.). Figure 4-19 contains an overview of these shared features.

It is therefore a good idea to divide all FGs in cell 1 into an action unit that controls the motors, magnets, etc. and a manager unit that monitors and initiates the execution of actions. This means that each functional group is divided into the following functional units:

- **Manager** (monitoring and initiating actions within an FG)
- **Unit** (controlling and executing the actions themselves)

This division into monitoring and action units also offers advantages in terms of reusability in the other functional areas or new production systems (e.g. the “Gripper arm” functional area requires the “Arm” and “Magnet” action units, among others, which can be reused).

The functional units of a functional group are represented by SDL substructure diagrams. For simplicity, only the “robot” functional unit will be described below.

### SDL substructure diagram with functional unit

The static interfaces between the FUs are described by the channels and signals as in the other F Specs. They are shown in the SDL substructure diagram of the functional groups.

In all functional groups the “Manager” unit is responsible for communications with the other units and sends the relevant action commands to the “Unit” action unit. After receiving the action commands, the action unit sends its signals to the external components that are to execute the actions (motors, magnets, etc.)

Only the static interface description of the “Robot” functional group is explained in further detail below.

---

**Figure 4-19: Shared action units of functional units in cell 1**
### RO_MAN
Robot manager RO_MAN communicates with the managers of the other functional groups (turntable, press, R-Belt) via the interfaces (SS DR, SS RP, SS AR). The signals exchanged correspond to those in the static interface description in F Spec Level 1.

All commands sent by the RO_MAN robot manager to the RO_UNIT action unit run via the (SS MU) channel. The "PROCCED" signal list contains the commands for rotating, extending and retracting the arms and picking up/putting down the sheets of metal. Depending on the signals received from the other FGs (turntable, press, R-Belt), RO_MAN decides which action to carry out next.

### RO_UNIT
After an action command is received from RO_MAN, RO_UNIT sends the relevant signals to the arms (SS RME), rotation motor (SS RMR) or magnets (SS RMG). The sensor values indicating the current angle of rotation (SS RSR) or the position of the arm (SS RSE) are continuously checked. Depending on these values, RO_UNIT decides when the required action has been completed and sends the "DONE" signal to the manager (SS MU).

### Dynamic interface description
As in F Spec Level 0 and 1, selected dynamic communications sequences between the functional units are represented in MSCs. Characteristic sequences are selected on FU level, such as straight-forward scenarios, time-out handling and alarm messages.

---

**Message sequence charts (MSC)**
The functional units of the F Spec Level 2 are broken down into SDL processes in the Design phase. This breakdown is represented by SDL process interaction diagrams. The process behavior is described in SDL process diagrams and the subsystem interfaces are coded. The design is subsequently validated using the SICAT simulator.

**D Spec**

The division of the functional units into SDL processes is described by means of process interaction diagrams when the D Spec is created. The process behavior is described by SDL process diagrams (PD scaffolds), however the scaffolds themselves are not included in the D Spec. In addition, a state event matrix (SEM) is generated for each process and MSCs are created for complex process communications.

### Dividing the functional unit into SDL processes

| Each functional unit in production cell 1 is looked into to see into which processes it is to be divided. Because all manager functional units are characterized by independent functionality, these are not further subdivided. The relevant block diagrams only contain one process. On the other hand, the action FUs are divided in such a way that a process is responsible for the execution of a particular action (e.g. vertical movement, rotation, etc.). It is apparent at this point that some of the action functional units contain the same processes. (Note: The same processes only need to have different names in different block diagrams because of a restriction in SICAT (e.g. ROT_U1 and ROT_U2).) |
### SDL block diagrams (process interaction diagrams)

The ARM_UNIT and MAG_U1 processes are only contained in the RO_UNIT functional unit once. However, they are contained in 2 instances because the robot has 2 arms. In terms of the establishment of instances, both ARM_UNIT instances exist when the system is started, while the two MAG_U1 processes are only created by the ARM_UNIT process instances after the system has started. *(Note: SICAT does not allow the generation of a process to be shown in block diagrams)*

---

### Static communication between processes

Static communications between the processes are described by signal routes and signals that are sent on the routes.

The interfaces of the RO_MAN and RO_UNIT robot functional units will be described here as an example. The signal routes and signals in Figure 4-28 are referred to in the text with `(signalname on routenname)`. *(Note: see Figure 4-19 and Figure 4-20 "Process interaction diagrams of the RO_MANAG and RO_UNIT functional units)*

---

### RO_MAN

The RO_MAN process receives a message from the turntable indicating that a metal sheet is to be picked up *(ORD on SDR)*. It then sends a command to process ROT_U2 to turn towards the turntable *(ROT_LEFT on SMRU)* and to process ARM_UNIT (1) to extend to the length of the turntable *(EXT on SMEU)*. When both processes have confirmed execution with "DONE", RO_MAN sends the MAG_U1 (1) process the command to pick up the metal sheet *(PIC_UP on SMMU)*.

The following robot actions are initiated in the same way. When RO_MAN receives a signal from the press indicating that a pressed sheet is to be removed *(OPEN on SRP)*, ARM_UNIT (2) receives the command to extend Arm2 to the appropriate length *(EXT on SMEU)* and MAG_UNIT receives the command to retrieve the sheet from the press *(PIC_UP on SMMU)*. The manager confirms to the press that the sheet has been removed *(OPENCF on SRP)*. It then sends a command to ROT_U2 to rotate enough to allow Arm1 to load the press again *(ROT_RIGHT on SMRU)*. After the press has signaled that it is ready for another sheet *(MIDDLE on SRP)*, RO_MAN sends the command for extending Arm1 and putting down the sheet *(EXT on SMEU)* and *(DROP on SMMU)*. ARM_UNIT (2) receives the signal to extend *(EXT on SMEU)* and MAG_U1 receives the signal to place the sheet of metal on the R Belt *(DROP on SMMU)*. A robot cycle is then complete and RO_MAN again waits to receive the ORD signal from the turntable.

---

### ROT_U2

Process ROT_U2 controls the motor that carries out the rotations. It receives the commands to execute a rotation in a particular direction *(ROT_LEFT, ROT_RIGHT on SRU)* from the robot manager RO_MAN. ROT_U2 forwards the rotation command to the rotation motor *(LEFT_ON on SRU)*.

It continuously checks the current angle of rotation of the
**USING SDL/SICAT DURING THE ÖN TN DEVELOPMENT PROCESS**

| **ARM_UNIT** | The ARM_UNIT process controls the motor that retracts and extends the arms. The execution of the actions and communication with RO_MAN is based on the same principle as with ROT_U2 and is therefore not dealt with further here. |
| **MAG_U1** | The MAG_U1 process is responsible for executing the actions of the electromagnetic grab. The same applies for the execution of these actions as for ARM_UNIT and ROT_U2. |

### Dynamic interface description

| **Dynamic communications between the SDL processes are represented by MSCs.** |

**Message sequence charts (MSC)**

| **see Figure 4-23 “Message sequence of the processes of the RO_UNIT functional unit”** |

### Process description using PD scaffolds and comments

| **The behavior of the processes is described in the form of process diagrams (PD scaffolds) and comments. When SICAT is used, the content of the process diagram symbols is only shown on comment level ("inscript" in SICAT). The code level is only added in the “Implementation” phase.** |

| **The PD scaffolds of the processes from “Production cell 1” are explained on the basis of an example below:** |

| **The RO_MAN and ARM_UN processes of the “Robot” functional group represent the PD scaffolds and will be described in greater detail. The description is followed by a list of special SDL constructs and how they are used in the “Cell 1” example.** |

| **see Figure 4-24 and Figure 4-25 “The ARM_UN and RO_MAN processes”** |

### Process interaction diagrams

| **ARM_UN process** | When in wait state, the ARM_UN process accepts "EXT(value)" or "RETRACT(value)" commands from RO_MAN. The commands define the degree to which the arm is to be extended or retracted. ARM_UN then sends a "REQ" request to the sensor and compares the "pos" measurement result sent by the sensor with "SE_ANSW(pos)" with the 'value' value. If the required value has not been reached yet ("degree = act_pos" check), ARM_UN sends the start command to the motor ("EXT_ON" or "RETR_ON" command). It then checks the current value of the sensor again and runs in this loop until the "value" value is reached. The motor then receives the "OFF" command and the RO_MAN manager receives the "DONE" signal. |
| **RO_MAN process** | When the system is started, the RO_MAN process first receives the process IDs of the arm and grab processes. |
It is assumed that arm 1 is positioned in front of the turntable as the initial angle of rotation. The manager sends both arms a command to extend to their initial position ("EXT(pos0)" and "EXT(ZB)"). After initialization, RO_MAN waits in “wait” state for the "ORD" signal from the turntable. After this it sends the relevant commands to the ARM_UNIT1,2, MAG_U1 and ROT_U2 processes (see also the "Static interface description between processes" section). Because there is no metal sheet in the press in the first cycle after the system is started, this sends the "MIDDLE" signal, which is received by RO_MAN in "Wait_for_press_state" state. RO_MAN sends the command to turn directly towards the press with Arm1 (angle "a3") and to load the press. After this, Arm 1 is retracted to "pos0" and the robot returns to the original position. RO_MAN is again in “wait” state and waits for the "ORD" signal from the turntable. Because there is now a metal sheet in the press, RO_MAN receives the “OPEN” signal in the "Wait_for_press_state" state and the normal cycle is passed through.

| Procedures / Macros | The "ZB_GEN" process in the FU "GEN_ZB" of the F-Belt contains the "BELT_UNIT" procedure. The "ZB_GEN" process must check whether a metal sheet has reached the end of the belt (range of the light barriers) in order to stop the belt. The process must also check whether the metal sheet has left the range of the light barrier in order to be conveyed to the turntable. In both cases, the values of the sensor at the light barrier are checked and the motor is started or stopped accordingly. This check is defined as a procedure and is called within the process. This distinction between arrival within the range of the light barrier ("state_var = true") and leaving it again ("state_var = false") is made by means of the "state_var" variable, which is transferred to the "BELT_U" procedure (see Figure 4-26 ). (Note: The procedure scope symbol cannot be inserted in SICAT to make a procedure known) |
| Creating SDL processes | There is an example of how to create processes in the RO_UNIT action unit. Each of the two arm processes creates a type MAG_U1 grab process. The "UN_ARM" processes save the process ID of the magnet processes created after creation in the subsequent task with "Gripper := OFFSPRING" (see Figure 4-25 ). |
| Using SAVE | The "RO_MAN" process contains an example of how to use SAVE. The process can receive the “MIDDLE” signal from the press in "Wait6, Wait7, Wait18 and Wait_for_conf" states but can only process it in "Wait_for_middle" state. This means that the signal must be saved using SAVE (see Figure 4-24). |
| Continuous signals / enabling condition | The use of continuous signals and enabling conditions is not supported in SICAT version 1201. |
| Indirect addressing | Indirect addressing without special address specifications is used in all sensor checks on the processes responsible for rotations and other movements (U1/2_ROT, VERT_U, UN_ARM). The "REQ" signal is always sent by these processes to the corresponding channels via unique signal routes and channels, so that no address specifications are required (see Figure 4-25 ). |
| Addressing with TO | There is an example of addressing with "TO" in the robot's "RO_UNIT" functional unit. Robot manager "RO_MAN" addresses arm processes "Arm1" and "Arm2" and their grippers "Gripper1" and "Gripper2" |
directly. Because the manager is located in a different block, it must know the process IDs of these processes. It is sent the "ARM_ID(grip_id)" signals by the arm processes. It saves the PIDs in the subsequent task with Gripper := grip_id and Arm := SENDER (see Figure 4-24).

Addressing with VIA

Addressing with VIA is used when sending commands and acknowledgments between the manager processes and action units, for example. Because the processes are in different blocks, they do not know the corresponding PIDs. However, addressing is unique because of the unique signal routes and channels between these processes (see Figure 4-24).

SDL timer concept

An SDL timer is used in the "MAN_PR" process of the press. Timer T1 sets the duration for the press. T1 is started within a task: "SET (NOW + PR_Timeout,T1)". Here "PR_Timeout" corresponds to the value for the duration of pressing. In "Pressing" state the process receives time-out signal "T1" and knows that the pressing process is complete (see Figure 4-27).

SDL wildcards

SDL wildcards are used in the robot's "RO_MAN" process, for example. The robot manager can receive an "ORD" signal from the turntable in any state except "WAIT". This is how the turntable indicates that a new sheet of metal is to be picked up. The robot manager only processes these signals after it has completed a full cycle (see Figure 4-24).

Using SDL data types

In order to obtain the most abstract specification possible, the values for positions, angles of rotation or timers are defined as variables. The variables are defined in the text symbols of the process diagrams and are assigned fixed values with "synonym". The advantage here is that any changes to the values only need to be made in the text symbols and not in all the other symbols of the process diagrams.

The arm positions and angles of rotation in the "RO_MAN" process offer an example. Here "pos0" corresponds to the initial position of Arm 1, while "ZB" represents the position for picking up the metal sheet from the turntable, "P" the position for loading and unloading the press and "AB" the position for unloading the sheet onto the R-Belt. In the angles of rotation, a1 represents the angle of rotation of arm 2 between the initial position and the press, a2 represents the angle of rotation from arm 1 to the press or arm 2 to the R-Belt and a3 the rotation to initial position (see fig. 12).

SDL variable definition

The local process variables are defined in the text symbols of the process diagrams. You will find examples in almost all process diagrams.

Generating a state event matrix

A state event matrix is created for each process. The matrix is generated when the SICAT analysis function is called and is stored in a file named "processname.sta". This file is included in the D Spec documentation.

State event matrix

see Figure 4-28 "State event matrix of the ARM_UN process"
Figure 4-22: Process interaction diagram for the "RO_UNIT" functional unit
Figure 4-23: Message sequence between the processes of the "RO_UNIT" functional unit.
The MSC in Figure 4-23 shows the message sequence between the processes of the “RO_UNIT” functional unit. For simplicity, only one instance is shown for each of the "ARM_UN" and MAG_UN" processes. The sequence shows the creation of the "MAG_UN" instance, the positioning of the arm, the picking up of the metal sheet from the turntable and the start of the robot's rotation towards the press.
Figure 4-24: Process diagram for the RO_MAN process
Figure 4-25: Process diagram for the "ARM_UN" process
Figure 4-26: Process diagram for the "ZB_GEN" process
**Figure 4-27:** Process diagram for the "PR_MAN" process
Coding the subsystem interfaces

The PD scaffolds are refined until the exported interfaces (signals, signal types and remote procedures) are defined. The SICAT code generator is used to create program modules from the interface definitions.

The interface definitions in the “cell 1” example can be found in the PD scaffolds.

4.6.2.2 Validating a design using the SICAT simulator

The SDL diagrams are checked for consistency and completeness using the SICAT simulator.

Preparations for working with the simulator

There is a detailed description of how to use the simulator in section 4.1 under the heading “Validating the design with the SICAT simulator”. The “production cell” example must also be adapted in accordance with the rules for the SDL process diagrams to be simulated. The following changes must be made to the existing process diagrams:

<table>
<thead>
<tr>
<th>Adaptations to addressing</th>
<th>Addressing with &quot;VIA&quot; must be replaced with direct addressing using the keyword &quot;TO&quot; and specification of the name of the receiver process.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>In the case of the RO_MAN process, the instances of the ARM_UNIT and MAG_U1 processes must be addressed by means of their process IDs. The &quot;SIDestinationPID&quot; variable must be set accordingly in the code level of the OUTPUT symbols for this purpose:</td>
</tr>
<tr>
<td></td>
<td>e.g. SIDestinationPID = Arm1;</td>
</tr>
<tr>
<td></td>
<td>The value of &quot;Arm1&quot; must be obtained from the &quot;SISenderPID&quot; variables before this. The &quot;Arm1&quot; instance is then addressed in the inscript level of the OUTPUT symbols using the keyword &quot;DESTINATION&quot;:</td>
</tr>
<tr>
<td></td>
<td>e.g. EXT TO DESTINATION</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Transferring data together with signals</th>
<th>All processes in the &quot;cell 1&quot; example expect or send some data that is transferred together with signals. For this reason, a data buffer must be defined in the TEXT symbol of each process and made known to the simulator. The example below is for rotation process ROT_U2:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>typedef struct RotBuffer{</td>
</tr>
<tr>
<td></td>
<td>long inputbuff;</td>
</tr>
<tr>
<td></td>
<td>} RotBuffer;</td>
</tr>
<tr>
<td></td>
<td>#define SIBufferType RotBuffer</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Receiving data together with signals</th>
<th>The processes that control the motors, e.g. the ROT_U1/U2 rotation processes, also receive data from their manager processes together with the control signals, for example the angle at which they are to rotate. They also receive measurement values from the sensors that indicate the current positions or angles.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The simulator provides the “SIBufferType” type default send and receive buffers “SI SendBuffer” and “SI ReceiveBuffer”. The data is retrieved or set in the code level of an SDL symbol (e.g. TASK, INPUT, OUTPUT) by means of a local variable:</td>
</tr>
<tr>
<td></td>
<td>receivevar = SIReceiveBuffer; or SI SendBuffer = sendvar;</td>
</tr>
</tbody>
</table>

| Sending and receiving predefined user-specific data | In the case of the RO_MAN, ER_MAN and PR_MAN manager processes, separate data buffers should be used. These contain the |
buffers

values for angles and positions that are sent with the control signals to the processes that execute the actions. The data buffers are also created in the code level of the TEXT symbols of the manager processes. As an example, the TEXT symbol of the RO_MAN process contains the following definitions:

```c
typedef struct RoBuffer{                  /* Definition of the */
    long inputbuff;                    /* general */
} RoBuffer;                                     /* receive or*/
#define SIBufferType RoBuffer;    /* send buffer    */
#define pos0 8           /* Initial position of arm1 */
#define ZB 12            /* Arm position for picking up the metal sheet from the F belt */
#define P 17              /* Arm position for loading and unloading the press */
#define AB 22           /* Arm position for unloading the metal sheet onto the R belt */
#define a1 30            /* Angle of arm2 in relation to press and arm2 in relation to R belt */
#define a2 50            /* Angle of arm1 in relation to press */
#define a3 85            /* Angle for return to original position */
struct SIBufferType RoBuffer[7] = {
    pos0,ZB,P,AB,a1,a2,a3};
long Arm1, Arm2;
long Gripper1, Gripper2;
```

Definition of variables local to the process

Variables local to a process, e.g. for retrieving data from the receive buffer, are defined in the TEXT symbol of the processes. The RO_MAN process, for example, requires variables for the process IDs of the arm processes and their gripper processes. The TEXT symbol contains the following definitions:

```c
long Arm1, Arm2;
long Gripper1, Gripper2;
```

Handling timers

A timer is set in the manager process of the PR_MAN press which defines the length of the pressing process. The timer is set in the Inscript level of a TASK symbol:

```
SET(PressPeriod,PressTimer)
```

PressPeriod and PressTimer are variables that are defined in the TEXT symbol of PR_MAN. PressPeriod is set to the value 10 with a "define" statement.

Enabling conditions

Continuous signals and enabling conditions are not supported in the current version of SICAT. The corresponding symbols must be removed from the process diagrams of the RO_MAN and DR_MAN processes and the "CONF" may only be sent by RO_MAN to DR_MAN to confirm that the metal sheet has been picked up after these actions are complete.

Wildcards

The current version of SICAT does not supports the use of wildcards in STATE symbols. The DR_MAN process must be changed accordingly. It is possible to insert a SAVE symbol in all STATE-INPUT combinations or to insert an additional signal between RO_MAN and DR_MAN. The latter option was used in the example. RO_MAN sends the signal "OK" to DR_MAN when it is ready to pick up another metal sheet.
Carrying out simulation

The simulation is carried out in accordance with the description in chapter 4.1 under the heading "Carrying out simulation".
5 CM and the production of SDL CM sources

The transmission of SDL documents into the CM pool and the production of SDL CM sources are described in the "Software Entwicklung mit SDL und zentraler Produktion. CASE-Verfahren und Tools" (Software development with SDL and central production. CASE procedure and tools) guidelines (P30308-A5858-A000-01-0035).
6 Appendix

6.1 The SDL block diagram symbol set

Block interaction diagram:

- Block reference, frame
- Unidirectional channel(-) non-delaying
- Bidirectional channel (-) non-delaying
- Unidirectional channel(-) delaying
- Bidirectional channel(-) delaying

Process interaction diagram:

- Frame
- Signal route (unidirectional)
- Signal route (bidirectional)

(in contrast with channels, there are only Non-delaying signal routes)

(-) = not supported by SICAT
6.2 SDL process diagram symbol set and wildcards

Symbols in process diagram:

- Start
- Stop
- State
- Nextstate
- Save
- Input
- Task
- Create Process
- Decision
- Output
- Comment
- Text extension (-)
- Transition Option

- Procedure Call
- Procedure Scope
- Macro Call
- Macro Inlet
- Macro Outlet
- Text Symbol
- Enabling Condition(-)
- Service Reference(-)
- Connector

(-) not supported by SICAT
Wildcards in the process diagram

The * in the input signal represents remaining signals that do not occur in a particular state either in input or save symbols.

The * in the input signal represents the remaining signals that are not listed in a particular state either in input or save symbols.

Note: All wildcards are supported by the SICAT PD editor. However, the SICAT code generator only interprets the wildcard that expresses that the next state is the same as the initial state.
6.3 The MSC symbol set

subscriber_A

Instance (column form)

subscriber_B

Instance (line form)

Message

overtaking message

crossing messages

Environment

Action

Create process

Stop process

Setting and sequence of a timer

Setting and resetting a timer

Coregion

Condition

Entity refinement

Submsc (refined entity)

Comment(-)

Text symbol

(*) = not supported by SICAT
6.4 Differences in SICAT from Z.100 and Z.120

Differences and additional features in SICAT in comparison with Z.100

- **SICAT does not support SDL/PR completely for the labeling of graphical symbols.**
- **SICAT does not support the SDL abstract data types concept**
- **Differences from the SDL block diagrams**
  - SICAT only allows delaying channels to be shown
  - SICAT does not allow channel substructures to be shown
  - SICAT only supports generic systems in the process diagram (transition option) and not in the block diagram (SICAT does not allow option symbols to be shown in block diagrams).
  - SICAT does not allow comment symbols to be inserted in block diagrams. Comments can only occur within SICAT block diagrams (/*comment*/).
  - SICAT does not support the process in block diagrams (process interaction programs).

\[\text{SDL block diagram constructs not supported by SICAT:}\]

- Non-delaying channel (unidirectional)
- Non-delaying channel (bidirectional)
- Channel substructure
- Option

- **Differences from SDL process diagrams**
  - SICAT allows a process to be divided into so-called drawing boards. SDL does not recognize the term drawing boards. However, SDL allows a process to be distributed over several pages.
  - Procedures are created in separate diagrams in SDL. SICAT does not support separate procedure diagrams. procedures are created within process diagrams in SICAT.
  - The comment symbol in the SICAT process diagram corresponds to the text extension symbol in SDL. SICAT itself does not recognize text extension symbols.
  - Symbols in SICAT-PD can be given informal text and program code. The informal text is not seen on the screen or in the printout, but appears as a comment in the generated source code.
  - SICAT does not support SDL services.
  - SICAT offers semantic extensions that simplify automatic conversion to structured program code: loops, yes/no branches and multiple branches.
  - SICAT has an EXIT_ON symbol: SDL decision meaning "exit loop".
  - According to Z.100, macros can have any number of inputs and outputs. However, SICAT only permits macros with one input and output.
  - SICAT differentiates between global and local text symbols. Local text symbols are optional and can be assigned a process start or procedure start symbol. Each process diagram must contain precisely one global text symbol. The global text symbol is intended for global variable declarations and for include statements for files, while local text symbols are intended for declaring variables that are local to procedures [17].
Differences and additional features in SICAT in comparison with Z.120

SICAT contains some differences and additional features in comparison with the MSC standard in respect of the representation of instances, messages, timers, connections and comments.

Instances

- Z.120 only recognizes the representation of instances in column and dotted line form. SICAT also supports another representational form with an invisible axis in which only the instance header and footer are shown.

   **Entity constructs:**

<table>
<thead>
<tr>
<th>SICAT</th>
<th>Z.120</th>
</tr>
</thead>
<tbody>
<tr>
<td>process</td>
<td>process</td>
</tr>
<tr>
<td>process</td>
<td>process</td>
</tr>
<tr>
<td>process</td>
<td>process</td>
</tr>
</tbody>
</table>

Process diagram constructs:

- **SICAT**
  - Not supported
  - Comment
  - EXIT_ON (Exit from a loop)
  - Only one input and output permitted)

- **SDL**
  - Service
  - Enabling Condition
  - Text extension
  - Comment
  - Macro call

  No direct correspondence with SDL (can be modeled as a decision)

  (Any number of inputs and outputs)
Messages
- Z.120 only recognizes asynchronous message exchange between instances and between instances and the environment.
- SICAT also supports the representation of synchronous communication, remote procedure calls and inband communication. Inband communication means sending information (e.g. sound) in an existing connection.
- SICAT allows optional messages that can be sent to be identified. These messages are marked with an "o" at the beginning.
- If two messages cross, then the representation of the “crossed” message is interrupted at the point of intersection in SICAT.

Message constructs:

<table>
<thead>
<tr>
<th>SICAT</th>
<th>Z.120</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asynchronous communication</td>
<td></td>
</tr>
<tr>
<td>Synchronous communication</td>
<td></td>
</tr>
<tr>
<td>remote procedure call</td>
<td></td>
</tr>
<tr>
<td>inband communication</td>
<td></td>
</tr>
<tr>
<td>inband communication</td>
<td></td>
</tr>
</tbody>
</table>

Timers
- At present, timers are represented differently in Z.120 and SICAT. At present, the setting of a timer is indicated by a small rectangle in Z.120, while a stylized hour-glass is used for the same purpose in SICAT. However, the timer symbol already used in SICAT will be introduced in the next edition of Z.120 (1996).
- Z.120 recognizes two timer constructs: setting with time-out and setting with reset.
- SICAT supports 5 separate timer constructs that can be represented independently in the diagram: setting the timer, setting the timer and time-out, setting the timer and reset, time-out and, finally, reset.
- In SICAT the timers are shown next to the instances in axial representation and inside the instance in column representation. In Z.120 the timers are shown next to the instances both in column and dotted line representation.
Connections

- SICAT uses so-called connections to allow messages and timers to be grouped together. Causal relationships between messages, timers and messages and timers can also be represented. Z.120 does not provide constructs for this purpose.

- SICAT makes a distinction between "ordered" and "unordered" connections. Ordered connections reflect the order in which messages and timers are received or sent. Unordered connections leave the chronological sequence open in accordance with the coregion construct in Z.120.

- Ordered connections are represented by a continuous vertical line, while unordered connections are represented by a dotted line.

Examples of ordered connections:

/* Receiving message s1 causes s2 to be sent */

/* First message s1 is received, then s2. Both messages together cause s3 to disappear */

/* The arrival of message s1 starts the timer and sends s2. As a consequence of the time-out, message s3 is sent in the left diagram. In the right hand diagram the arrival of s3 resets the timer */
Examples of unordered connections

/* First messages s1 and s2 are received. No information is provided about the sequence, i.e. s1 before s2 or s2 before s1. s3 is only sent when both messages have arrived */

/* The arrival of s1 messages causes s2 and s3 messages to be sent. It is not possible to provide information about the chronological sequence, i.e. s2 before s3 or s3 before s2 */
6.5 SOLUTION O.N.E

SOLUTION O.N.E (Optimized Network Evolution) is ÖN’s strategy for the architecture of switching systems and is aimed at meeting the requirements of future broadband networks (such as video on demand). SOLUTION O.N.E contains a number of additional features in comparison with classic EWSD architecture. Only the basic principles of SOLUTION O.N.E will be dealt with here. The main aim is to show how the SOLUTION O.N.E architecture units can be modeled using SDL diagrams.

6.5.1 The client/server principle

Client/server architectures are based on the principle of division of tasks. The client requests services and the server provides these services. A server can itself assume the role of a client and request services from other servers in order to complete its work.

![Client/Server Model](image)

Within the SOLUTION O.N.E software, so-called service provision units (SPUs) provide services to other system components or subscribers. Thus, SPUs assume the role of servers in SOLUTION O.N.E systems. However, SPUs can also operate as clients, i.e. when they use the services of other servers themselves [24].

Services are accessed by means of interfaces. Externally visible services are made known to the clients by means of so-called published interfaces (PIFs). A PIF consists of a number of functionally related operations. Clients can use services by calling the relevant operations of the server interface.

The server provides the services by implementing the operations. The client remains unaware of how a service is implemented within the server.

SOLUTION O.N.E supports synchronous and asynchronous operation calls. Synchronous communication is by means of remote procedure calls (RPC). While an operation is being carried out, the control function is transferred from the client to the server. When the server has completed the operation, control is returned to the client, together with any results. Thus, the client explicitly waits for a ready message from the server during synchronous communications.

Control is not transferred from the client to the server during asynchronous communications. In this case, the client continues to work independently of the server after the service has been requested. Asynchronous communication is handled by means of CAST calls.
Figure 6-2: SPU with one service and SPU with several services

An SPU can be purely a client SPU, purely a server SPU or both a client and a server. SDL models regard a system as a client/server architecture in principle. SDL models a system as a large number of parallel and concurrent processes that communicate with each other asynchronously. Each process can be both a client and a server. A process (client) can activate another process (server) to provide services by sending a message. If the server process in turn requires services from another process, then it is simultaneously a client.

Because SDL implicitly uses the client/server principle when modeling the system, SDL can naturally be used easily to describe client/server architectures.

SPUs are comparable with the concept of abstract data types (ADT). A user only sees the specification part of an ADT (corresponding to the PIF). The implementation part of an ADT (corresponds to the service implementations) is hidden from the user.

6.5.2 SOLUTION O.N.E - shell model

SOLUTION O.N.E software is structured hierarchically, i.e. vertically in shells, in a shell model. The shell model is also structured horizontally by means of so-called loadtypes.

A loadtype is defined by the set of all capsules belonging to a particular processing platform. A capsule is a SOLUTION O.N.E container class containing SPUs (service provision units). The terms capsule and SPU are explained in the next subsection.

Possible processing platforms:
- GPV (Group Processor)
- SLT (Signaling Link Terminal)
- MPU (Main Processing Unit)

The shell model has the following characteristics [23,24,25]:
- SOLUTION O.N.E software is arranged in shells. Each shell consists of a set of SPUs.
- The shells are not created individually for each hardware platform. The shells apply throughout the system for all new SOLUTION O.N.E CHILL platforms.
- An SPU is located in precisely one shell [25].
- An object from shell \( n-1 \) is only visible in shell \( n \) if it has been explicitly exported from shell \( n-1 \) previously.
- Objects from a shell (e.g. PIFs) may not be imported from SPUs of lower shells [25].
- The message flow from an inner shell to an outer one is permitted if it has been explicitly requested by an object from the outer shell previously.
- It must be possible to modify and extend software from higher shells without making changes to elements in lower shells.
- From a client's perspective, a server must be located in the same shell or an inner shell.
- Shells 2 and 3 form the CHILL processing platform together with the hardware (shell 1) [25].
- The software of shell 2 is not defined in the form of SPUs [25].
- Shell 3 only contains a capsule with 2 SPUs:
  - Supervisor SPU (with OS kernel, ITP, SW error treatment, startup supervisor, etc.)
  - Supervisor Definition SPU (parts of the OS kernel user interface, not already defined in the compiler) [25]
### 6.5.3 SOLUTION O.N.E structure units

In addition to the already existing EWSD structural elements (process, procedure, module and region, the following new structural units were introduced for the SOLUTION O.N.E system architecture [25]:

- capsule,
- service provision unit (SPU),
- published interface (PIF),
- recovery suite (RS),
- virtual CPU (VCPU).

- A processing platform contains one or more capsules.
- A capsule contains one or more SPU's.
- An SPU consists of a set of processes, procedures, regions and data with a strong functional relationship. Instead of these elementary modules, an SPU can also contain group modules (SPU portions). However, elementary modules and group modules may not be contained in an SPU at the same time.
- The services of SPUs are made known in the system by means of PIFs (published interfaces) and can only be addressed via PIFs.
- Several processes can be grouped together to form a recovery suite (RS) in an SPU. An RS can be restored without affecting other system operations.
- A virtual CPU is a series of processes that are assigned a shared CPU time budget. VCPUs exceed SPU and capsule limits. SPUs can contain processes from several different VCPUs. There are no hierarchical relationships between VCPUs and other structure units.

SOLUTION O.N.E thus supports a top down structure hierarchy that is only broken down by the virtual CPU concept.
As well as the architecture and structure units mentioned above, project management units are also significant in switching systems, and therefore in SOLUTION O.N.E systems. These include system, functional area and feature.

A functional area (e.g. call processing or recovery) is formed by several functionally cohesive SPUs. A system consists of a set of functional areas or (depending on the view) a set of features. The features define the service profile for customers. From a customer perspective, a system basically consists of a collection of features which dictate the purchasing price.

SOLUTION O.N.E makes a distinction between design time units and build time units in the structure units, depending on the phase of the development process in which the units are created (see Figure 6-4).

Design time units are created in the Design phase. The design units must satisfy the requirements in the Analysis phase and the general design principles for software units (see chapter 2).

On the other hand, build time units are responsible for assigning resources. Capsules form a protected address space for a set of SPUs and provide a guaranteed amount of heap and timer budget for these SPUs.

As well as capsules, virtual CPUs also exist as build time units. Virtual CPUs consist of a set of process instances with a shared CPU time budget. Build time units are not created by the developer. Special groups (Systems Engineering) are responsible for bringing capsules and VCPUs together.

![Figure 6-4: SOLUTION O.N.E: Extract from lifecycle](image)

### 6.5.4 Design time units

The structure units for the Design phase are described below [25]. Their special properties are outlined and there is an explanation of how the units can be modeled with SDL and MSC.

#### 6.5.4.1 Module

A module is the smallest possible compilation unit. It can contain type and constant definitions as well as data, procedure and process declarations. A module also represents a unit for information hiding. The elements of a module are only externally visible if they are explicitly made visible (e.g. with GRANT or EXPORT statements). Only elementary modules (which, unlike group modules, do not contain other modules) can be reused in different SPUs (Compiler Option <>REUSABLE<> [29]). However, a module may only occur once in the same SPU.
There are also generic modules [29].

A module is a structural unit for:
- information hiding
- compilation
- configuration management
- documentation
- testing
- reuse (restricted)

**Mapping an elementary module in SDL:**

![Mapping an elementary module in SDL diagram]

**Figure 6-5: Mapping an elementary module in SDL**

**Modeling aspects of a module:**

<table>
<thead>
<tr>
<th>Purpose and requirements</th>
<th>Design principles</th>
<th>Descriptive technique</th>
<th>Tool</th>
<th>DMN phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functionality</td>
<td>Inform. hiding</td>
<td>SDL substructure diagram</td>
<td>SICAT-BD</td>
<td>Design</td>
</tr>
<tr>
<td>Reusability</td>
<td>Localization</td>
<td>SDL process diagram</td>
<td>SICAT-PD</td>
<td></td>
</tr>
<tr>
<td>Testability</td>
<td>High FAN IN</td>
<td>SDL procedure diagram</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compilation unit</td>
<td>Functional cohesion (elementary modules)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Informative cohesion (group modules)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 6-6: Modeling aspects of a module**

6.5.4.2 Process

A distinction must be made between process type and process instance. Process types describe templates or samples. Process instances are correct copies of certain types. Process instances can be created when the system is started or during runtime. Dynamically created processes are always located within the SPU of the parent instance. Process instances can be parallel and concurrent. EWSD and SOLUTION O.N.E process instances communicate with each other by message exchange or with shared data mechanisms.
Although SDL allows data to be exchanged through global data (view/reveal and export/import), the exchange of messages through explicit symbols is a fundamental philosophy of SDL. This is why transitions of an SDL process should also be activated by incoming signals (input symbols) rather than by continuous signals.

EWSD and SOLUTION O.N.E processes are finite state machines. This means that they can be represented very well by SDL processes.

EWSD and SOLUTION O.N.E processes are always defined within elementary modules. Processes can be further structured by procedures.

A process is a structure unit for:
- concurrency (parallelism)
- reusability
- documentation
- testing
- recovery

**Mapping the process structure in SDL:**

![Diagram of process structure in SDL]

**Figure 6-7: Mapping the process structure in SDL**

**Modeling aspects of a process**

<table>
<thead>
<tr>
<th>Purpose and requirements</th>
<th>Design principles</th>
<th>Descriptive technique</th>
<th>Tool</th>
<th>DMN phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concurrency</td>
<td>Inform. Hiding</td>
<td>SDL process diagram</td>
<td>SICAT-PD</td>
<td>Design</td>
</tr>
<tr>
<td></td>
<td>Localization</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>High FAN IN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Moderate FAN OUT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Explicit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>exchange of messages</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 6-8: Modeling aspects of a process**
6.5.4.3 Region

A region contains data and procedures, but no processes. The region is a means of synchronizing processes because it only permits exclusive access to shared resources. Only one procedure in a region can be called at a time. A process that attempts to access an already busy region must wait until the current process has left the region again. Thus, a region corresponds to a monitor concept according to Hoare.

A region is a structure unit for:

- information hiding
- compiling
- configuration management
- documentation
- testing
- reusability
- access controls and synchronization

Mapping the region structure in SDL:

![Mapping the region structure in SDL](image)

Figure 6-9: Mapping the region structure in SDL

Modeling aspects of a region:

<table>
<thead>
<tr>
<th>Purpose and requirements</th>
<th>Design principles</th>
<th>Descriptive technique</th>
<th>Tool</th>
<th>DMN phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronization (exclusive access)</td>
<td>Information hiding</td>
<td>SDL process diagram</td>
<td>SICAT-PD</td>
<td>Design</td>
</tr>
<tr>
<td>Reusability</td>
<td>Localization</td>
<td>SDL procedure diagram</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Testability</td>
<td>High FAN IN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Moderate FAN OUT</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 6-10: Modeling aspects of a region

6.5.4.4 SPU and PIF

SPUs (service provision units) play a major role in SOLUTION O.N.E. client/server architecture. They are the components that provide the services for the other system components (clients) or
subscribers. An SPU can itself act as a client and can request services from other SPUs. An SPU can be regarded as the basic (functional) design unit in SOLUTION O.N.E:

"Basic unit of functional partitioning within SOLUTION O.N.E" [25]

An SPU consists of a set of data, procedures, processes and regions that have a strong functional relationship with each other and together provide a shared service or a set of services. An SPU must be made known within the system in order for a service to be requested from it. This is achieved by means of so-called public interfaces (PIFs) [26]. The PIFs form the SPUs' interfaces to the environment. The services of an SPU can only be accessed via PIFs. The implementation of a service remains hidden within an SPU (information hiding).

Objects within an SPU are only externally visible if they have been explicitly exported and objects outside an SPU are only visible within an SPU if they have been imported.

Figure 6-11: SPU as a visibility border [25]

An SPU be used on a variety of processing platforms (e.g. for distribution of load and reusability).

An SPU can be migrated from one capsule to another or from one processing platform to another during build time (location transparency) (see Figure 6-4). For this reason, the design and implementation of an SPU must be locally transparent. A consequence of this is the fact that SPUs may only communicate by "interprocess communication" and not by means of "shared data". An SPU process must therefore be activated for the purposes of a transition, and therefore a service, by means of an input signal and not by continuous signals.

An SPU has dynamic and static properties:

SPUs consist of a set of processes. Processes are dynamic in nature [9, 25]. SPUs should replace the EWSD concept of the subsystems and can be used on a variety of processing platforms. An SPU thus has a static architectural aspect [25].

6.5.4.4.1 Design influences and rules when creating SPUs

Structuring rules and principles:

- The main objective when designing an SPU is to achieve a clear, functional structure [25].
  -> concise, meaningful and clear names for the various services.
- SPUs, that only provide one service must comply with functional cohesion. SPUs that provide several services must support informative cohesion [14, 25].
Informative cohesion is when the services administer access to a shared data structure or a shared resource [14, 25].
- Design and implementation must be "location transparent", i.e. they must take place irrespective of the processing platform on which the SPU is to run [25].
• From the client's perspective, a server may only be located in the same shell as the client or in an inner shell [25].

• Information about SPU resource requirements must be transferred to the next phase of the SW lifecycle in dispatched form. Information about resource requirements plays an important role in the formation of capsules.

• AN SPU either consists of processes, regions, etc. (so-called elementary modules) or of SPU portions (group modules). However, an SPU may never contain elementary modules and SPU portions at the same time [25]. SPU portions can be used to incorporate old EWSD software in SOLUTION O.N.E architecture[9,25].

• The software of shells 5, 6 and 7 is completely formed from SPUs. Each shell can contain one or more capsules.

• Shell 3 contains only one capsule with two SPUs:
  - Supervisor SPU (with OS kernel, ITP, SW error treatment, startup supervisor, etc.)
  - Supervisor definition SPU (parts of the OS kernel user interface not already defined in the compiler)[25]

• An SPU may only occur once in a capsule [25].

• An SPU may only occur once in a shell [25].

• The software in shell 2 is not implemented in the form of SPUs.

• SPUs may not be nested [25].

Communication rules and principles:

• Communication within an SPU (shared data, local procedure calls) is faster than communication between different SPUs. Large SPUs with numerous processes, regions, etc. mean good performance. However, large SPUs are less flexible than small ones. In addition, large SPUs usually have low cohesion. The communication performance of small SPUs can be increased if they are located in a shared capsule [25].

• The elements of an SPU are always located in a shared address space. This is why shared memory and local procedure calls can be used for communication within an SPU [25].

• SPUs may not communicate via shared data [25].

• The number of external interfaces (operations of a PIF) of an SPU must be kept as small as possible [25]. Loose coupling[15, 29]!

• The flow of information from an SPU in an inner shell to an SPU in an outer shell is only permitted if this has been explicitly requested by the higher SPU before hand [25].

• SPUs must communicate via recovery-tolerant protocols.

Different views in SPU design
Mapping SPU and PIF on SDL:

As its name suggests, a service providing unit provides services. Services are defined using processes and/or remote procedures. If a service is defined by a remote procedure, this service is provided *synchronously*. If the service is provided through the transition of a process in the server, the message that activates the transition is processed *asynchronously*.

Processes are the most important components in an SPU. This is why it makes sense to represent an SPU as a page block (process interaction diagram) in SPU.

The messages are of the same type as the service they activate (service type). Different messages can be of the same type. If a service is provided asynchronously, the PIFs contain messages of the same type as the relevant services. The messages and associated service types must be defined outside the SPU in order to ensure that the services can be used by other SPUs. The positioning of this definition in the system can be used to regulate access to the services.

Process P_2 in Figure 6-13 provides a service that is supplied by a remote procedure. This remote procedure “A” is explicitly marked as exportable in the procedure diagram (keyword “exported” in the procedure scope symbol). The positioning of a text symbol in which procedure A is identified as a remote procedure regulates access to procedure A.
Figure 6-13: SDL/MSC modeling for SPU and PIF

MSCs can be used to show the external and internal communication behavior of an SPU.

**Modeling aspects of SPU and PIF:**

<table>
<thead>
<tr>
<th>Purpose and request</th>
<th>Design principle</th>
<th>Descriptive technique</th>
<th>Tool</th>
<th>DMN phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPU gen. One service</td>
<td>No external interfaces Concise, meaningful and clear description of services</td>
<td>SDL block diagram (Process interaction diagram) MSC</td>
<td>SICAT-BD SICAT-SC</td>
<td>Design</td>
</tr>
<tr>
<td>Functionality</td>
<td>Functional cohesion Informative cohesion</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flexibility and reusability</td>
<td>High FAN IN (with high cohesion) Tendency towards small SPUs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Performance, Real Time</td>
<td>Small external interfaces Tendency towards large SPUs Low FAN OUT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Communication</td>
<td>asynchronou</td>
<td>SDL block diagramm (process interaction-diagram) MSC ADT</td>
<td>SICAT-BD SICAT-SC</td>
<td>Design</td>
</tr>
<tr>
<td>PIF gen.</td>
<td>Loose coupling Information hiding</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 6-14: Modeling aspects of SPU PIF

Although the assignment of resources is a build time activity rather than a design activity (see SOLUTION O.N.E lifecycle in Figure 6-4), the developer of an SPU should provide details of the resources required for the build time phase. This is because the developer knows this information best and can therefore simplify the work of the creator of the capsules (unit of resource budgeting) [25].

Budgetary information can be provided in comment form in SDL (text symbols, notes, comment symbols) or can be read directly from the SDL diagrams (timers, maximum number of process instances).
6.5.4.5 Recovery Suite

Recovery unit in SOLUTION O.N.E systems. A recover suite (RS) is a collection of process types of a shared SPU that mat need to be recovered together. The recovery mechanism within an RS does not interfere with the mode of operation of the rest of the software [25].

Characteristics and significance of an RS

A full RS is contained within one SPU [25].

The recovery process does not interfere with the other system units.

An RS can contain two types of members: passive and controlling members. Restoring a controlling member of an RS causes the whole RS to be recovered. On the other hand, a passive member can be restored by itself without necessarily affecting the other members of the RS.

If an SPU is reproduced (at build time) (SPU = unit of replication), any existing recovery suites are also reproduced. Recovery activities within replications do not affect each other [25].

The formation of recovery suites is not a normal design decision, but rather an exception in the design process. It is therefore not necessary for a process type to belong to a recovery suite, nor for an SPU to contain any recovery suites at all.

Design influences when an RS is created

- Recovery-units are created at design time.
- If an RS contains passive members, it is necessary to ensure that the controlling members of the RS are not affected in their mode of operation while a passive member is being restored [25] (achieved through a recovery tolerant protocol).
- If an RS contains only controlling members, then no restrictions need be observed during intra-suite communication [25].
- When two processes from different recovery suites communicate, it is necessary to ensure that each of them can continue to operate while the other is in a recovery state.
- The scope of a recovery suite is always based on an SPU.

Modeling a recovery suite with SDL:

Process types within an SPU that belong to a recovery suite can be identified as such by means of comments.
Figure 6-15: SOLUTION O.N.E structural hierarchy
Figure 6-16: Classification of SOLUTION O.N.E structure units
## 6.6 Tables and overviews

### Quick reference guide: Conventional switching software

#### Design principles

<table>
<thead>
<tr>
<th>Modeling Design units</th>
<th>Purpose and requirements</th>
<th>Design principles</th>
<th>Descriptive technique</th>
<th>Tool</th>
<th>DMN phase</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Feature</strong></td>
<td>Functional Flexibility Testability</td>
<td><strong>Software cohesion</strong></td>
<td>SD substructure Text</td>
<td><strong>SICAT-BD PF3</strong></td>
<td><strong>Prelim. phase Analysis</strong> System test</td>
</tr>
<tr>
<td><strong>System</strong></td>
<td>Functional Testability System morphology criteria: Width, depth (7+2) Balancing</td>
<td><strong>SDL system diagram SDL substructure MSC</strong></td>
<td><strong>SICAT-BD SICAT-SC</strong></td>
<td><strong>Analysis</strong> System test</td>
<td></td>
</tr>
<tr>
<td><strong>Subsystem group (functional area)</strong></td>
<td>Functional Reusability Testability</td>
<td><strong>High cohesion Loose coupling Localization</strong></td>
<td><strong>SDL block diagram MSC</strong></td>
<td><strong>SICAT-BD SICAT-SC</strong></td>
<td><strong>Analysis Integration test</strong></td>
</tr>
<tr>
<td><strong>Subsystem</strong></td>
<td>Functional Reusability</td>
<td><strong>High cohesion Localization</strong></td>
<td><strong>SDL system diagram SDL substructure SDL block diagram MSC</strong></td>
<td><strong>SICAT-BD SICAT-SC</strong></td>
<td><strong>Design</strong></td>
</tr>
<tr>
<td><strong>Module</strong></td>
<td>Functional Reusability Testability Compiling unit</td>
<td><strong>Inform. hiding Localization High FAN IN Functional cohesion (Elementary modules) Informative cohesion (Group modules)</strong></td>
<td><strong>SDL substructure SDL process diagram SDL procedure diagram</strong></td>
<td><strong>SICAT-BD SICAT-PD</strong></td>
<td><strong>Design Integration test</strong></td>
</tr>
<tr>
<td><strong>Region</strong></td>
<td>Synchronization (exclusive access) Functional Testability</td>
<td><strong>Inform. hiding Localization High FAN IN Moderate FAN OUT</strong></td>
<td><strong>SDL process diagram SDL procedure diagram</strong></td>
<td><strong>SICAT-PD</strong></td>
<td><strong>Design Integration test</strong></td>
</tr>
<tr>
<td><strong>Process</strong></td>
<td>Concurrency Functional Testability</td>
<td><strong>Inform. hiding Localization High FAN IN Moderate FAN OUT Explicit exchange of messages</strong></td>
<td><strong>SDL process diagram SDL procedure diagram</strong></td>
<td><strong>SICAT-PD SICAT-SC</strong></td>
<td><strong>Design Integration test</strong></td>
</tr>
<tr>
<td><strong>Procedure</strong></td>
<td>Functional Reusability</td>
<td><strong>Localization</strong></td>
<td><strong>SDL procedure diagram</strong></td>
<td><strong>SICAT-PD</strong></td>
<td><strong>Design Integration test</strong></td>
</tr>
</tbody>
</table>

*Figure 6-17: "Conventional Switching Software" quick reference guide*
### Quick reference guide: VISION ONE software

#### Design principles

<table>
<thead>
<tr>
<th>Purpose and requirements</th>
<th>Design principles</th>
<th>Descriptive technique</th>
<th>Tool</th>
<th>DMN phase</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SPU gen.</strong></td>
<td>Concise, meaningful and clear description of services</td>
<td>SDL block diagram (process interaction diagram)</td>
<td>SICAT-BD SICAT-SC</td>
<td>Design</td>
</tr>
<tr>
<td><strong>One service</strong></td>
<td>Functional cohesion</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Several services</strong></td>
<td>Informative cohesion</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>High FAN IN (with high cohesion)</strong></td>
<td>Tendency towards smaller SPUs</td>
<td></td>
<td>SICAT-BD SICAT-SC</td>
<td>Design</td>
</tr>
<tr>
<td><strong>Small external interfaces</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Low FAN OUT</strong></td>
<td>Unidirectional channels</td>
<td>SDL block diagram (process interaction diagram)</td>
<td>SICAT-BD SICAT-SC</td>
<td>Design</td>
</tr>
<tr>
<td><strong>Bidirectional channels</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Recovery Suite:

<table>
<thead>
<tr>
<th>Spec. recovery requirement</th>
<th>Loose coupling</th>
<th>SDL block diagram (with comments)</th>
<th>SICAT-BD SICAT-SC</th>
<th>Design</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Flexibility</strong></td>
<td></td>
<td>MSC ADT</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Relocability</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**Figure 6-18: “SOLUTION O.N.E. Software” quick reference guide**
### SDL/MSC Methodology quick reference guide: Description of procedures within the MMN

<table>
<thead>
<tr>
<th>R Spec</th>
<th>Requirements placed on the environment by the system to be developed should be formulated in such a way that they are as independent as possible</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Form a block in the system as a counterpart for each requirement that is responsible for satisfying a requirement</td>
</tr>
<tr>
<td>F Spec (Level 0)</td>
<td>Connect each block to the environment by means of a channel</td>
</tr>
<tr>
<td></td>
<td>Introduce any additional blocks for context knowledge, shared data and shared resources</td>
</tr>
<tr>
<td></td>
<td>Link the blocks with channels Use MSCs to model communication scenarios between blocks</td>
</tr>
<tr>
<td></td>
<td>Record (FA) blocks as separate systems and repeat the steps above beginning with the formulation of requirements Use MSCs to model communication scenarios between blocks</td>
</tr>
<tr>
<td></td>
<td>Create block interaction diagrams with functional groups</td>
</tr>
<tr>
<td></td>
<td>Record (FG) blocks as separate systems and repeat the steps above beginning with the formulation of requirements Use MSCs to model communication scenarios between blocks</td>
</tr>
<tr>
<td></td>
<td>Create block interaction diagrams with functional units</td>
</tr>
<tr>
<td></td>
<td>Create a separate process for each requirement/task and define Independent behavior aspects in separate processes</td>
</tr>
<tr>
<td></td>
<td>Encapsulate shared data or bundles of shared data in separate processes</td>
</tr>
<tr>
<td></td>
<td>Define an (administrative) process for each pool of shared resources</td>
</tr>
<tr>
<td>D Spec</td>
<td>Use MSCs to describe process communication (in the event of complex communication)</td>
</tr>
<tr>
<td></td>
<td>Describe the external communication behavior of a process with STDs or as a PD frame Extend MSCs Conditions, Tasks</td>
</tr>
<tr>
<td></td>
<td>(not part of the D Spec)</td>
</tr>
<tr>
<td></td>
<td>Describe the complete behavior of the processes with process diagrams</td>
</tr>
<tr>
<td></td>
<td>Create procedures for large and frequently recurring transitions Use MSCs Conditions, Tasks</td>
</tr>
<tr>
<td></td>
<td>Store the SDL symbols with programming language statements Use MSCs Conditions, Tasks</td>
</tr>
<tr>
<td></td>
<td>Process diagrams ready for code generation Use MSCs Conditions, Tasks</td>
</tr>
<tr>
<td>Impl.</td>
<td>Code generation: CHILL, C ...</td>
</tr>
</tbody>
</table>

---

**Figure 6-19:** "SDL/MSC Methodology" quick reference guide
6.7 Glossary

Abstract data type (ADT)
The ADT is an extension to the elementary data types (integer, real, char) of the classic programming languages. Just as only certain operations are permitted on an elementary data type (for example fixed point operations on integers), an ADT only permits precisely defined operations on the data structure it represents. However, an ADT is not a defined object type with prescribed operations, but rather a mechanism that enables users to define their own data types with corresponding operations.

Abstract syntax notation One (ASN.1)
A non-target-environment-dependent descriptive means for specifying data and interfaces in higher protocol levels. Standardized by the CCITT in the X.208 recommendations since 1984.

Abstraction
Reduction to the essential and generalization of the specific.

Analysis
A phase in the software and system creation process. The requirements placed on a system are defined in the analysis phase.

Backus-Naur form (BNF)
A metalanguage for describing (programming) languages.

Baseline
A defined end point for a phase in the SDPP.

Bottom up
A system development procedure in which systems are created through successive composition, beginning with primitive elements.

Build time
The phase during system creation in which the whole system is created (generated) after the individual components have been designed and implemented. The components are assigned to various processing platforms during the build time and their resources are allocated (memory, processor time).

CAPSULE
Build time container class in SOLUTION O.N.E

Computer Aided Software Engineering (CASE)
CASE promotes an institutionalized software (or system) creation process and requires that methods and tools should be available for all phases of system creation.

CCITT High Level Language (CHILL)
Higher, problem-oriented programming language. Tailored to the implementation of switching software.

Client
Program unit in the client/server model that requests one or more services from a server.

Client server model
Structural model for organizing distributed systems. A system consists of program units that request services (clients) and program units that provide services (servers).
Cohesion
Qualifiable dimension for the coherence and internal strength of modules, subsystems and processes.

Commitee Consultativ International Telegraphique et Telephonique (CCITT)
International standards organization for the telecommunication sector. In 1993, the CCITT was renamed the ITU (International Telecommunication Union).

Coupling
Qualifiable dimension for the (communications) relationships between modules, subsystems and processes.

Data dictionary
A catalog of all data contained in a system. The structure and application of data are also apparent in the data dictionary.

Data flow diagram (DFD)
Data flow diagrams are part of the structured analysis method. They show the flow of information between processes (tasks) in graphical form.

Development manual (DMN)
manual containing the system development guidelines for ÖN. A Software Development Manual (SDMN), exists for software developments and a corresponding Hardware Development Manual (HDMN) exists for hardware developments.

Design
A phase in the software and system lifecycle. In the Design phase, an analysis (requirement specification) is used as a basis for dividing the system to be developed into subsystems and to model the communication relationships between the subsystems and between the subsystems and the environment. The design forms the basis for subsequent implementation.

Design specification (D Spec)
The D Spec is the main document of the Design phase in the SDPP. Drawing on the preceding R Spec and F Spec documents, solution concepts are created for system attributes that are to be added or modified. In particular, the required processes are defined and the internal interfaces are fixed.

Entity relationship modeling (ERM)
Data modeling according to Chen that permits entities, relationships and attributes to be represented in graphical form (entity relationship diagram).

Establishing a hierarchy
Creating levels of abstraction whose elements satisfy certain user requirements through their degree of abstraction.

Fan in
A module’s fan in is the number of modules that use it.

Fan out
The fan out is the number of subordinate modules directly required in order to implement a module.

Feature
The part of a system visible to the customer. From the customer’s perspective, a system consists of a set of features. The design and implementation of a feature usually has more effects than the creation of a subsystem or functional area.
Feature (LM)
Customer and system requirements are defined in the form of features.

Functional area (FA)
Functional area in project management with a strong functional cohesion. A functional area usually consists of a set of several functionally related service provision units. Complex functional areas can be further divided into functional groups. Call processing and recovery are examples of functional areas.

Functional group (FG)
Grouping of functional units with a strong functional relationship. Complex functional areas can be structured to form functional groups.

Functional unit (FU)
Functional units are the elementary service or function providers in switching systems. SPU, SW subsystems and hardware modules are all functional units.

Function specification (F Spec)
Along with the R Spec, the F Spec is one of the main documents of the Analysis phase in the SDPP. The feature requirements defined in the R Spec are embedded in the system structure in the F Spec, i.e. the way in which the features can be implemented in the hardware functional units (modules) and software functional units (subsystems) is described. The volume of tasks in an FSpec usually refers to a feature or to several logically related features. Because of its complexity, an F Spec is divided into F Spec Level 0, F Spec Level 1 and F Spec Level 2.

F Spec Level 0
This describes the functional areas that make up a system.

F Spec Level 1
This defines the function groups that make up a functional area.

F Spec Level 2
This describes the functional units that make up a functional group.

Functional cohesion
Software unit (module) that provides a precisely specified function (service). This function must be described in a concise and meaningful way. Functional cohesion is the highest level of cohesion.

Information hiding
Design principle of Parnas. Users of an object or an algorithm only sees the effect of their call and not how it is implemented.

Information modeling (IM)
Analysis of entities and their relationships within an area of application and their graphical representation in the form of entity relationship diagrams or corresponding derivatives.

Informative cohesion
Functions (services) that operate on the same data structure are grouped together to form a software unit. Each access operation must meet the criteria for functional cohesion and have its own separate entry point. Informative cohesion is the aim when forming abstract data types.

ITU
International Telecommunication Union. The new name for the CCITT since 1993.
Message
generally a message between system units or between system units and the environment.
A signal that carries user data is referred to as a message in SDL.

Message sequence chart (MSC)
A methods for representing communication scenarios between system components.
Standardized by the ITU in the Z.120 recommendations.

Method
Schedules, applied, rationalized procedure for achieving defined goals.

Migration
Moving a software unit from one processing platform to another during run-time.

Modular design (MD)
A design method that uses the same concepts as structured design but places greater emphasis on the term module.

Operation
Server services are addressed by a client by means of operations. Operations are part of a server's export interface. There are synchronous and asynchronous operations.

Process
Generally a sequence of activities in which input data is usually transformed into output data. AN SDL process consists of a finite set of defined states in which it can receive messages as stimuli from other processes or from the environment. Receiving of expected messages leads to transitions after which a process is once more in a defined state. AN SDL process can carry out actions during a transition and send messages to other processes or to the environment.

Processing platform
Hardware platform together with the programming language and underlying operating system.

Proximity
Design criterion. All information about an object (module) and all points from which an object can be manipulated should be in the “immediate vicinity” of the object. An important consequence of adhering to the proximity principle is the avoidance of "gotos".

Published interface (PIF)
Formal definition of a service interface. A PIF contains operations with which the services can be requested as well as the required type definitions for the associated parameters.

Recovery Suite (RS)
A SOLUTION O.N.E structure unit. Grouping together of a set of processes within an SPU that can be recovered together.

Relocation
Moving a software unit from one processing platform to another at build time, i.e. while the system is being created.

RPC (Remote Procedure Call)
Synchronous operation call in distributed systems.

Requirement specification (R-Spec)
A detailed description of (functional) requirements that are to be satisfied within a given budget and schedule. Also referred to as the customer requirement specification at ÖNTN. The R Spec is a major document of the Analysis phase within the DMN.
Scope  
Measure of the influence and range of a module.

SDL block diagram  
The term block diagram is used for two different types of diagram in SDL: block interaction diagrams and process interaction diagrams. Block interaction diagrams are the SDL substructure diagrams (keyword: "substructure"), while process interaction diagrams are the actual SDL block diagrams (keyword: "block"). The SDL system diagram is a special block interaction diagram.

SDL process diagram  
The input/output behavior, the states and transitions of an SDL process type are described in the form of SDL process diagrams.

SDL substructure diagram  
SDL substructure diagrams can be used to break down a complex system into sub-blocks until a sub-block only contains process types. Sub-blocks that only contain process types are described in process interaction diagrams (keyword: "Block").

SDL system  
A system modeled with SDL.

SDL system diagram  
The SDL system diagram forms the root of substructure diagrams and is therefore the highest level of refinement for a system modeled with SDL. In open systems, the interfaces to the environment are shown in the system diagram.

Server  
Program unit in the client/server model that provides one or more services on request.

Service  
Functional service that can be provided to a client by a server. Before it can be requested, a service must be made known in the system by means of a public interface.

Service Provision Unit (SPU)  
SPUs are the fundamental design unit in SOLUTION O.N.E. client/server architecture. SPUs are the servers that provide services for other system components (clients) or subscribers.

Signal  
Signals are event indicators that cause a transition in an SDL process. Signals that carry user data are referred to as messages.

Specification and Description Language (SDL)  
A specification method standardized by the ITU (Rec. Z.100) for describing event-oriented systems that are base on the exchange of messages.

Software Development Process Plan (SDPP)  
The SDPP divides software development into development phases (steps) that are separated from each other by defined baselines. The steps observe the principle of step-by-step refinement. The SDPP is part of the DMN.

SOLUTION O.N.E (Optimized Network Evolution)  
Architecture strategy for switching systems. Successor name for EWSD systems with effect from version 11.
State Overview Diagram
- Graphical language for representing a state-dependent input/output behavior of processes and subsystems. State overview diagrams are based on the theory of finite state machines.

Structured Analysis (SA)
- Method for analyzing system requirements. A system is analyzed using flow charts, data dictionaries and mini-specifications.

Structured Design (SD)
- Method for designing a system that places great emphasis on functional aspects. Structure design combines the following basic techniques: data dictionary, p-spec and structure chart.

Structured Programming (SP)
- Systematic and methodical programming with the requirement that only three control structures are used to describe the progress of a program: sequence, iteration and selection.

Subsystem
- Structure unit for project management. A system consists of a set of subsystems. The volume of tasks for a design specification is usually based on a subsystem. Modules are grouped together in accordance with implementation and functional cohesion in a subsystem.

Subsystem group
- Several subsystems grouped together to form a functional unit.

System
- Limited section from the real or notional world for achieving a given purpose.

System engineering
- Systematic use of principles, methods and tools for creating systems reliably and efficiently.

Task
- Sequence of steps (statements) to be executed together. SDL task statements within an SDL task symbol.

Virtual CPU
- Virtual CPUs count in SOLUTION O.N.E in the same way as capsules for the Build Time Container classes. A virtual CPU consists of a set of processes to which a shared CPU time budget is assigned.