Chapter

# 70

# *SOMT Concepts and Notations*

**This chapter describes the major concepts and notations used in the SOMT method. The notations treated includes object models, state charts, SDL diagrams and Message Sequence Charts (MSCs). Note that the descriptions in this section only briefly describes the different notations. For a more thorough treatment, please consult a textbook on the specific notation.**

# Activities, Models and Modules

As seen in the previous chapter, the SOMT method describes object oriented analysis and design as a number of activities that produce one or more models. A *model* is here used in an abstract fashion to denote a collection of diagrams, text documents or whatever is needed in the particular model. The concept of a model is used to be able to discuss the results of an activity without going into detail of how the actual diagrams/documents are organized.

A *module*, on the other hand, is the concept used in SOMT to define how the diagrams/documents are organized. The major purpose of the modules is that they form units that should be fairly self-contained and that can be developed by themselves, maybe by different teams. A module is a container of e.g. diagrams and textual documents that has no semantics by itself but that forms a scope unit for names (if this is not given by the notation used). For example, if the Analysis Object Model is described in two modules that both contain a class called “Person”, then these definitions do not refer to the same class. There will be two different “Person” classes, one in each module.

In practice it is beneficial to have a simple mapping between models and modules, either a one-to-one or, if a model is too large, a one-to many mapping where a model is decomposed into several modules.

The actual documentation produced according to SOMT is thus a collection of modules containing diagrams/documents that together form this particular projects representation of the SOMT models.

# Implinks and the Paste As Concept

The SOMT method introduces a number of different models that are used to describe different aspects of the system. In particular there are three levels formed by:

- The requirements that describe the problem and external requirements on a system
- The system analysis describing the concepts used in the system
- The system and object design that defines the structure and behavior of the system

One very common relation between objects in different models is that one object can be seen as an implementation of an object in another model. For example, an object in the object model created in the system analysis phase may be reintroduced in the design model as a process type or an ADT (abstract data type). Another example may be an object that was identified in the requirements analysis as something visible on the system boundary, and that later is reintroduced in the analysis object model and finally ends up as a signal in the SDL design. To represent this type of relations among objects the concept of implementation links (*implinks*) is used. An implink is a directed relation between two objects, usually (but not necessarily) in different models. Conceptually we get a picture as in Figure 606.

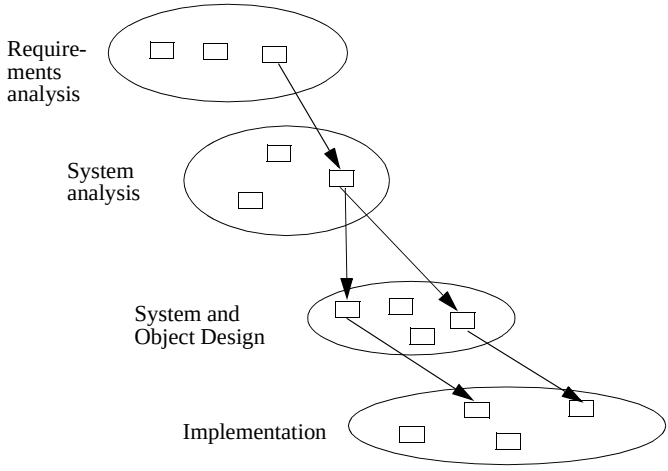


*Figure 606: Implinks between objects in different models*

If used carefully, the implinks give a possibility to trace requirements all the way down into code. There are several situations where the implinks are very useful:

- *What if* analysis. What are the consequences if a particular requirement is changed?
- Consistency checking. Are all concepts in e.g. the system analysis model implemented?

- Understanding design by following the links backwards from implementation to requirements. What is the purpose of a particular design object? What happens if we change it?

It is important to see that the act of creating an implink is a creative design action that encapsulates a design decision. The *Paste As* mechanism is a special concept used in SOMT to support the task of creating implinks. The idea is that an object in one model can be copied and then *pasted as* a new object in another model, see Figure 607. This action serves both to create the new object and to document the design step using an implink.

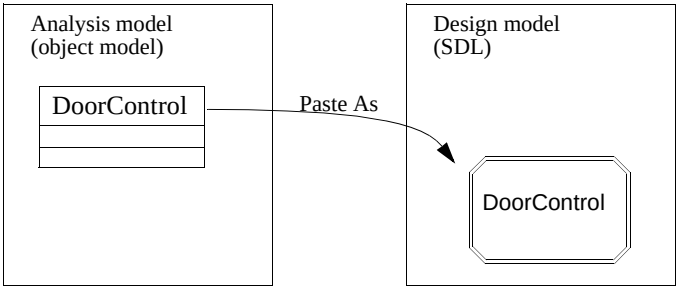


*Figure 607: Using Paste As to capture a design step from the system analysis model to SDL*

# Consistency Checking

The SOMT method uses a number of models and notations and the checking of various aspects of the models is an important part of a development project. This checking can be formulated as a question of identifying “entities” or “concepts” in different models (or in one model), to identify some rules of how these entities should relate to each other, and finally to check that the models are consistent with respect to these rules. This type of checking is in this document called *consistency checking*. Three different types of consistency checking can be identified:

- Checking the internal consistency within one model, e.g. checking that an SDL system is correct with respect to the syntactical rules for SDL

- Checking that two models are consistent with respect to each other and some consistency requirement, e.g. checking that all classes in an object model are described in the data dictionary
- Checking traceability aspects, i.e. checking that the entities in two models are correctly linked (using implinks) to each other. For example, checking that all objects in an analysis model are implemented in the design model

In some sense the checking of traceability aspects is a special case of the checking of the consistency between two models, but it is an important special case and it is given a special treatment in SOMT.

One general observation that can be made is that the identification of concepts/entities is very much depending on the particular notation used. Each separate notation will have to be treated separately following the particular rules that apply to this language. For formal languages like SDL, the concepts and procedures how to find the entities are well defined, while for other languages the rules are different, and for plain, informal text the entity identification will have to be explicitly done by the user. In SOMT there is a special possibility to “mark” words or phrases in text documents. The intention is that this marking means “this concept is important” or “this is a concept that I would like to for consistency checks”.

In the rest of this volume, each activity of the SOMT method together with its associated models will be described in different chapters. Each of these chapters will also include a discussion on consistency rules that are relevant for this particular model.

# Object Model Notation

The object model notation from Object Modeling Technique (OMT) and Unified Modeling Language (UML) is a commonly accepted graphical notation used for drawing diagrams that describe objects and the relations between them. The notation that is used in examples in this volume is shown in Figure 608 through Figure 613. For more details about other, more advanced OMT object model concepts, please consult [20], and for details about the UML notation see [36] and [37].

## Class

The most important concept in an object model is the class definition. A *class* is a description of a group of similar objects that share the properties defined by the class. The object model notation for a class is exemplified in Figure 608, where the second class definition also shows how to define attributes and operations.

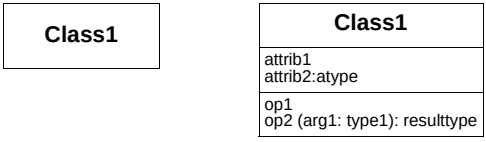


*Figure 608: A collapsed class symbol and a class symbol with attributes and operations*

In some cases it is necessary to reference classes from an external module. The notation used for this purpose is *ExternalModule::Class*.

Classes may *inherit* attributes and operations from other classes. The object model notation for this is shown in Figure 609.

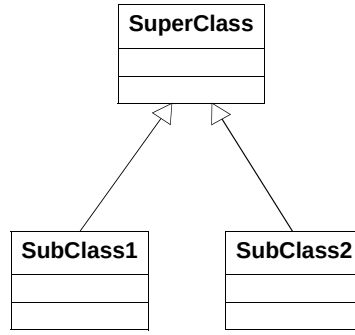


*Figure 609: Inheritance between classes*

## Relations and Multiplicity

Classes may be physically or logically related to each other. This is shown in the object model by means of *associations* as shown in Figure 610. An association may have a name and/or the endpoints of the association may be labeled by the role of this endpoint.

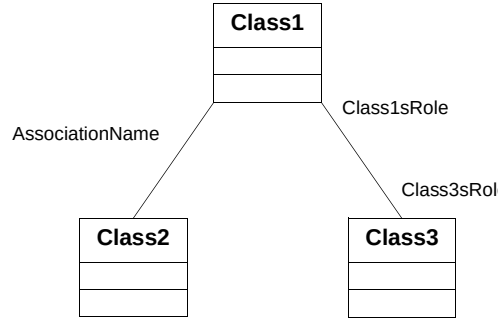


*Figure 610: Associations between classes*

*Aggregation* is special kind of association that has its own notation as shown in Figure 611.

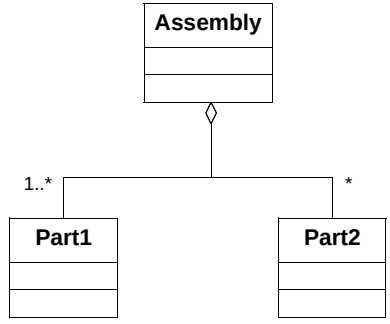


*Figure 611: Aggregation*

The endpoints of associations and aggregations may have a multiplicity as shown in Figure 612.

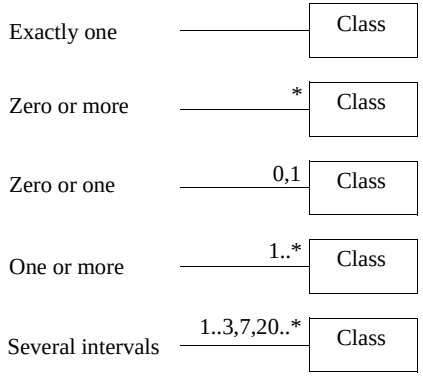


*Figure 612: Multiplicity of associations and aggregations*

## Module

In practise the complete requirements object model is often too big to fit into one diagram. To solve this problem it is possible to use multiple object model diagrams that can be organized into a module, which simply is a list of diagrams. It is important to notice that a class may be present in more than one diagram and still only represent one logical class.

## Objects

Besides class definitions, object models may of course also contain objects and their relations. The relation that exists between objects are links, which corresponds to the associations for classes. The object symbol has one field containing the name of the object together with a reference to the class, and an attribute field where constant or default values can be assigned to the object attributes. See Figure 613.

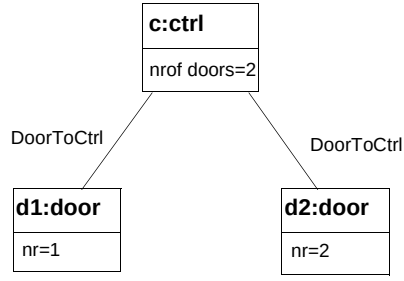


*Figure 613: Objects related by links*

# State Chart Notation

The notation presented in this section is a subset of the notation for state charts presented by David Harel [35] which is used in the Unified Modeling Language (UML) [36] as well as in the Object Modeling Technique (OMT) [20].

A state chart model is suitable to use together with class and object models. The descriptions of the behavior of a class in a class diagram is collected into a state chart which describes the dynamic view of the model by means of states and transitions. The notation to use is presented below.

## Notation

### State

To describe state charts a state symbol is needed. The state symbol is divided into three compartments, State Name, State Variable and Internal Activity, which are all optional. The top compartment contains the optional name of the state. State symbols with the same name within the same context are considered to be the same. It is not necessary to name states and if several anonymous states exists; each anonymous state symbol is considered to be an individual state. Figure 614 shows a collapsed state and a state with state variables and events.

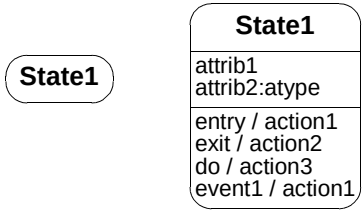


*Figure 614: A collapsed state symbol and a state symbol with state variables and events*

A state symbol may contain a State Variable compartment. This compartment highlights attributes of the class which are used or in some way affected by the behavior described in the state chart.

The third compartment is the Internal Activity compartment which describes the activities of the current state, activities that are done upon entering the state, activities taking place while in the state and activities executed when exiting the state. Each activity is described in the format:

*event-name argument-list '/' action expression*

Each event name may appear only once within a state symbol. The event names "entry", "exit" and "do" are reserved and they describe the following actions:

*'entry' '/' action expression*
An atomic action performed upon entering the state

*'exit' '/' action expression*
An atomic action performed upon exiting the state

*'do' '/' action expression*
An action performed during the state

### Transition

The second necessary symbol for drawing state charts is the transition. The transition symbol is an arrow which connects two symbols of either type of state, start and termination, see Figure 615.

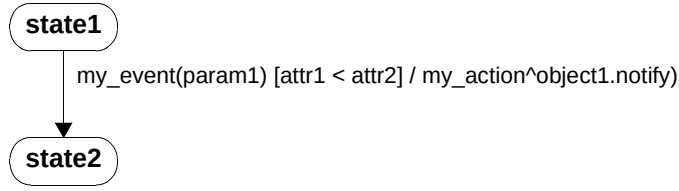


*Figure 615: The transition from state1 to state2 is triggered by the event my_event and the condition that attr1 is less than attr2*

The syntax for the transition symbol follows the format:

*event-signature '[' guard-condition']' '/' action-expression '^' send-clause*

The event-signature consists of the parts:

*event-name '(' parameter ',' ... ')'*

The guard-condition is a Boolean expression formed by the parameters of the triggering event together with possible attributes and links of the object described by the state chart.

The action-expression describes the action that is executed during the transition. The action may be described by procedures, affected attributes and links.

The send-clause has the format:

*destination-expression '.' Destination-event-name '(' argument '.' ...')'*

The destination-expression identifies the receiving object or a set of receiving objects.

The Destination-event-name is the name of an event that may be received by the receiving object(s).

### Start and Termination Symbol

The start symbol denotes the starting point of a state machine described by a state chart and the termination symbol denotes the point of termination of a state machine. Figure 616 shows a simple state machine, describing the behavior of a door, including a start symbol and a termination symbol.

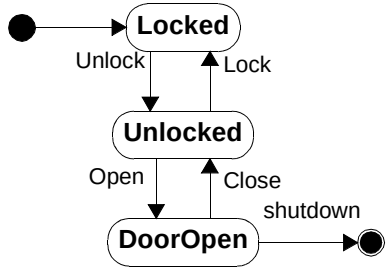


*Figure 616: A simple state chart with a start symbol and a termination symbol*

### Substates

States may be refined into nested diagrams of sub-states, or hierarchical states. The state represents a simplification of more complex behavior expressed in the nested diagram.

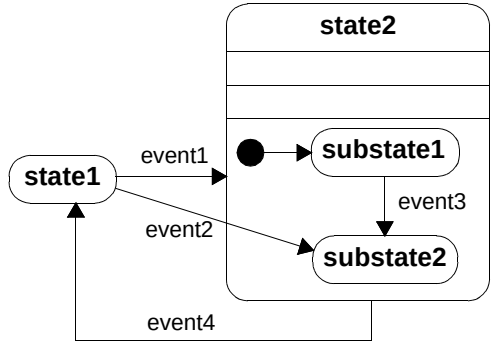


*Figure 617: A state chart with states and substates.*

## State Charts in SOMT

### State Charts and SDL

Both State Charts and SDL process graphs are two notations used to express state machines. The two notations have both their advantages. State Charts are good for expressing high level functionality typically used in early activities like analysis. SDL is good for expressing detailed functionality as in design activities. SDL is also a formal notation, with well defined semantics, from which it is possible to generate code.

### State Charts in Requirements Analysis

State charts might be used to express high level functionality of a system by focusing on the behavior of the system rather than the behavior of the system's parts. It might also be useful to express the behavior of complex actors that are interacting with the system.

### State Charts in System Analysis

In the context of system analysis, state charts are useful to express the behavior of the system parts and how these parts interact dynamically. The architecture of the system is described in a class diagram named Logical Architecture. Using state charts in addition to the class diagram makes it possible to describe the dynamic properties and the behavior of the system or parts of the system. The behavior descriptions should be included in a set of documents called the Object Behavior Diagram which together with the Logical Architecture is part if the Analysis Object Model.

### State Charts in System and Object Design

During the activities of System and Object Design, SDL process diagram is the preferred notation for describing the behavior of the system. The product of the Object Design activity should be a complete description of the system which enables code generation. State Charts, due to their weak semantics, may be useful to express a less detailed overview of rather complex behavior expressed in SDL.

# Message Sequence Charts

A message sequence chart (MSC) is a high-level description of the message interaction between system components and their environment. A major advantage of the MSC language is its clear and unambiguous graphical layout which immediately gives an intuitive understanding of the described system behavior. The syntax and semantics of MSCs are standardized by ITU-T, as recommendation Z.120 [25].

There are various application areas for MSCs and within the system development process MSCs play a role in nearly all stages, complementing SDL on many respects. MSCs can e.g. be used:

- To define the requirements of a system
- For object oriented analysis and design (object interaction)
- As an overview specification of process communication
- For simulation and consistency check of SDL specifications
- As a basis for automatic generation of SDL skeleton specifications
- As a basis for specification of TTCN test cases
- For documentation

## Plain MSC

The most fundamental language constructs of MSCs are *instances* (e.g., entities of SDL systems, blocks, processes and services) and *messages* describing the communication events, see Figure 618.

Another basic language construct is the *condition* symbol which is drawn as a hexagon. A condition describes either a global system state referring to all instances contained in the MSC, or a state referring to a subset of instances (a non-global condition). The minimum subset is a single instance.

An MSC can reference another MSC using an *MSC reference* symbol. (Such a symbol can also reference a High-level MSC, explained later.) This symbol is drawn as a rectangle with rounded corners and has the name of the associated MSC stated inside it. MSC references can for example be used to have one MSC describing an initialization sequence and then reference this MSC from a number of other MSCs.

The reference symbol may not only refer to an MSC but can also contain MSC reference expressions that reference more than one MSC. This construct gives us a very compact MSC representation and it also provides an excellent means for reusability of certain MSCs.

The textual MSC expressions are constructed from the operators alt, par, loop, opt and exc:

- An MSC reference with the keyword **alt** denotes alternative executions of MSC sections. Only one of the alternatives is applicable in an instantiation of the actual sequence.
- The **par** operation defines parallel executions for MSC sections. All events within the parallel MSC sections will be executed (free merge) with the only restriction that the event order within each section must be preserved.
- An MSC reference with a **loop** construct is used for iterations and can have several forms. The most general construct, `loop<n,m>`, where n and m are natural numbers, denotes iteration at least n and most m times. The operands may be replaced by the keyword inf, like in `loop<n,inf>`. This means that the loop will be executed at least n times. If the second operand is omitted, like in `loop<n>`, this will be interpreted as `loop<n,n>`. If both operands are omitted the interpretation will be `loop<1,inf>`.
- The **opt** operation is an operator with one operand only. It is interpreted in the same way as an alt operation where the second operand is an empty MSC.
- An MSC reference where the text starts with **exc** followed by the name of an MSC indicates that the MSC can be aborted at the position of the MSC reference symbol and instead continued with the referenced MSC. If the exception does not occur the events following the exc expression are executed. The exc operator can thus be viewed as an alternative where the second operand is the entire rest of the MSC. MSC references with exceptions are frequently used to indicate exceptional cases when using MSCs to formalize use cases.

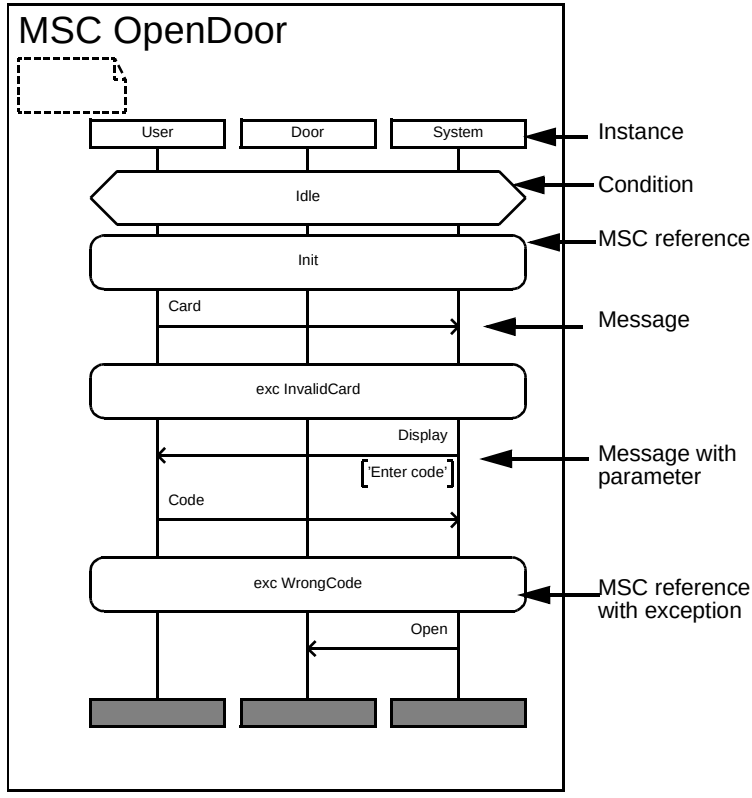


*Figure 618: Example of a plain MSC*

By means of *inline operator expressions* composition of event structures may be defined inside an MSC. Graphically, the inline expression is described by a rectangle with dashed horizontal lines as separators. The operator keyword **alt**, **loop**, **opt**, **par** or **exc**, placed in the upper left corner, are used with the same meaning as when used together with the MSC reference symbol.

Whether to use inline operator expressions or MSC reference symbols with an operator is a matter of taste. The same things can be expressed with both notations. Using inline expression several scenarios can be expressed in one single diagram, i.e. we just have one single file to handle. If we use MSC reference symbols to express e.g. exceptions and alternatives there is a need for us to handle several files. On the other

hand, the diagram becomes less cluttered if we refer to other MSCs instead of trying to express all possible scenarios in it.

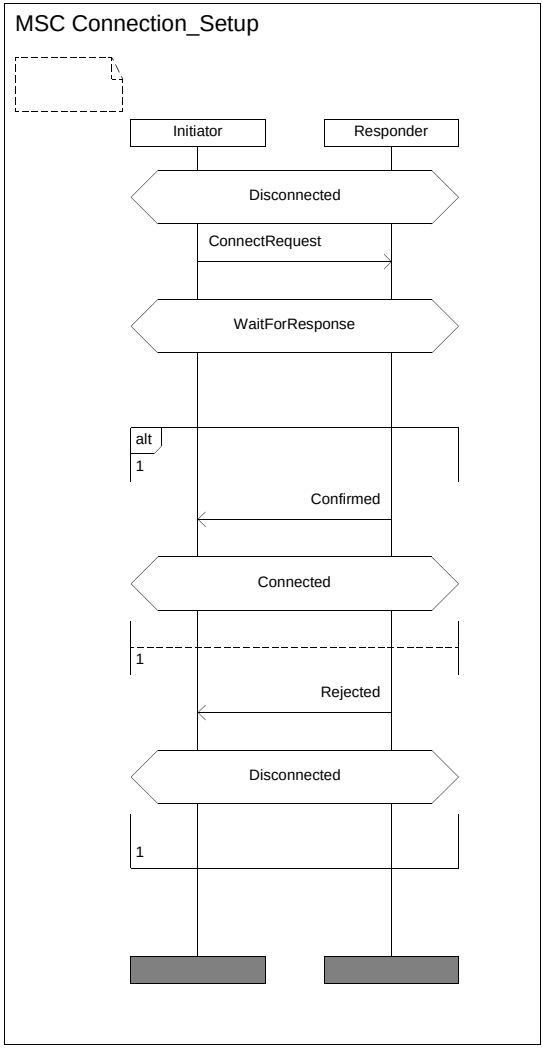


*Figure 619: A plain MSC with inline operator expression*

For information about more plain MSC concepts, like e.g. timers, please consult the MSC standard Z.120.

## HMSC

A high-level MSC (HMSC) provides a means to graphically define how a set of MSCs can be combined. Contrary to plain MSCs, instances and messages are not shown within an HMSC, but it focus completely on the composition aspects. You can get a picture of how an HMSC works in practice by comparing it with a road map. HMSCs, like normal road maps, may easily become quite complex if they are not structured in any way. Fortunately, HMSCs can be hierarchically structured, i.e. it is possible to refine HMSCs by other HMSCs. The power of the MSC language is considerably improved with the new concepts introduced with HMSCs. It is e.g. much easier to specify a main scenario together with all accompanying exceptions.

An HMSC is a directed graph where each node is either (see Figure 620):

- A start symbol which denotes the start of an HMSC (there is exactly one start symbol in each HMSC).
- A stop symbol which denotes the end of an HMSC.
- An MSC reference used to point out another (H)MSC diagram which defines the meaning of the reference. The reference construct in HMSC can thus be seen as a placeholder for an MSC diagram or another HMSC diagram. Like a reference symbol in a plain MSC, an MSC reference symbol in an HMSC can contain references to several (H)MSCs through using MSC reference expressions and the operators alt, par, loop, opt and exc.
- A condition symbol is used to set restrictions on how adjacent referenced MSCs can be constructed. An HMSC condition immediately preceding an MSC reference has to agree with the (global) initial condition of the referenced MSC according to name identification. All conditions on HMSC level are considered to be global, they refer to all instances contained in the MSC. They can be used to guard the composition of MSCs described by HMSCs.
- A connection point which denotes that two crossing lines are actually connected. This symbol has no semantic meaning but is introduced only to simplify the layout of the HMSC.

Flow lines connect the nodes in the HMSC and they indicate the sequencing that is possible among the nodes in the HMSC. If there is more than one outgoing flow line from a node this indicates an alternative.

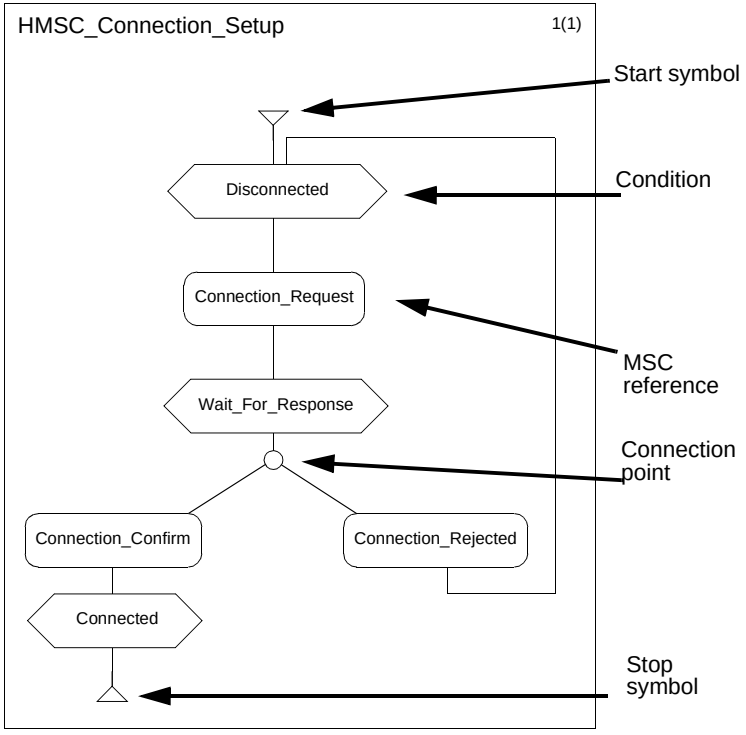


*Figure 620: Example of an HMSC*

# SDL

SDL (Specification and Description Language) is mainly used for specifying behavior of real-time systems. This section provides a small overview of the language. For further reading, we recommend [26], [27] and [28] as useful textbooks about SDL. The SDL language is standardized by ITU-T, as recommendation Z.100 [23].

An SDL system consists of the following components:

- Structure
  - Hierarchical decomposition with *system*, *block*, and *process* as the main building blocks
  - Type hierarchies: inheritance, generalization and specialization are supported for hierarchical building blocks
- Communication
  - Asynchronous *signals* with optional signal parameters
  - *Remote procedure* calls for synchronous communication
- Behavior
  - *Processes*
- Data
  - *Abstract data types* that can be inherited, generalized and specialized
  - ASN.1 data types according to Z.105 [24]. See "ASN.1" on page 3696
- Modularization
  - Components like *block/process types*, *data types*, and *signals* can be placed into *packages* that can be imported into a *system*, enabling separate development.

## Structure

Figure 621 shows the hierarchical levels in SDL: *system*, *block*, *process*, *procedure* and *service*.

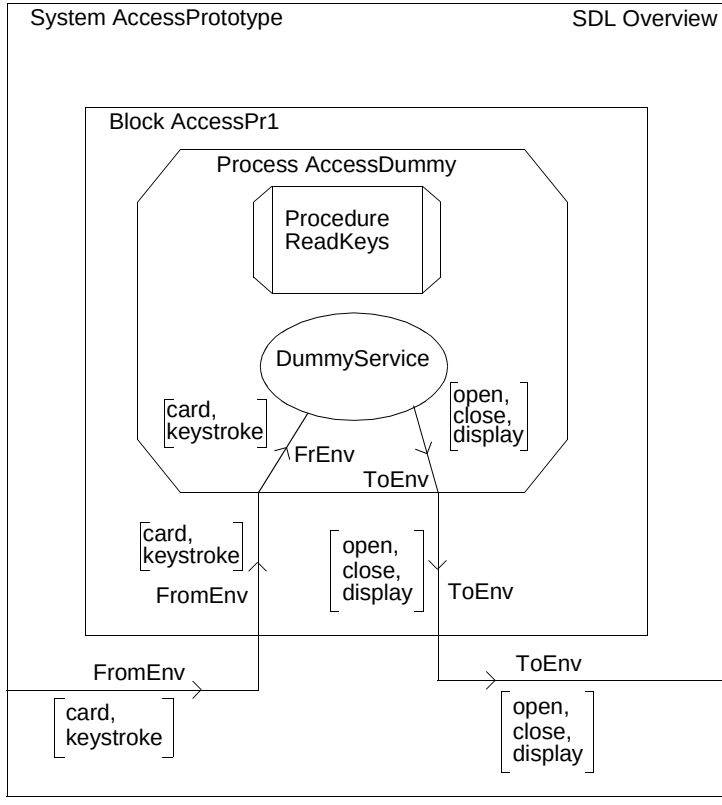


*Figure 621: The hierarchical decomposition of an SDL specification*

A *system* must contain at least one *block* and a *block* must contain at least one *process*. *Services* can exist within a process, being executed one at a time, controlled by the received signals. Thus the incoming signal sets of services in one process must be disjoint. Using *procedures* is a way to structure the information within processes and services. Processes, services and procedures are described by a flow chart-alike notation, see "Behavior" on page 3687.

The static structure of a *system* is defined in terms of *blocks*. Blocks communicate by means of *channels*.

The dynamic structure of an SDL *system* consists of a set of *processes* that run in parallel. A *process* is a finite state machine extended with data. *Processes* are independent of each other and communicate with discrete *signals* by means of *signal routes*.

## Communication

Since SDL does not allow any use of global data, all information that has to be exchanged must be sent along with *signals* between processes, or between processes and environment. Signals are sent asynchronously, i.e. the sending process continues executing without waiting for an acknowledgment from the receiving process.

Signals travel through *channels* between blocks, and from one process to another via *signal routes*. See Figure 621.

Synchronous communication is possible via a shorthand, *remote procedure call*. This shorthand is transformed to signal sending with an extra signal for the acknowledgment. Remote procedures are often used when a process wants to offer services to other processes.

## Behavior

The dynamic behavior in an SDL system is described in the processes. Processes in SDL can be created at system start or created and terminated dynamically at run time. More than one instance of a process can exist. Each instance has a unique *process identifier* (PId). This makes it possible to send signals to individual instances of a process. The concept of processes and process instances that work autonomously and concurrently makes SDL appropriate for distributed applications.

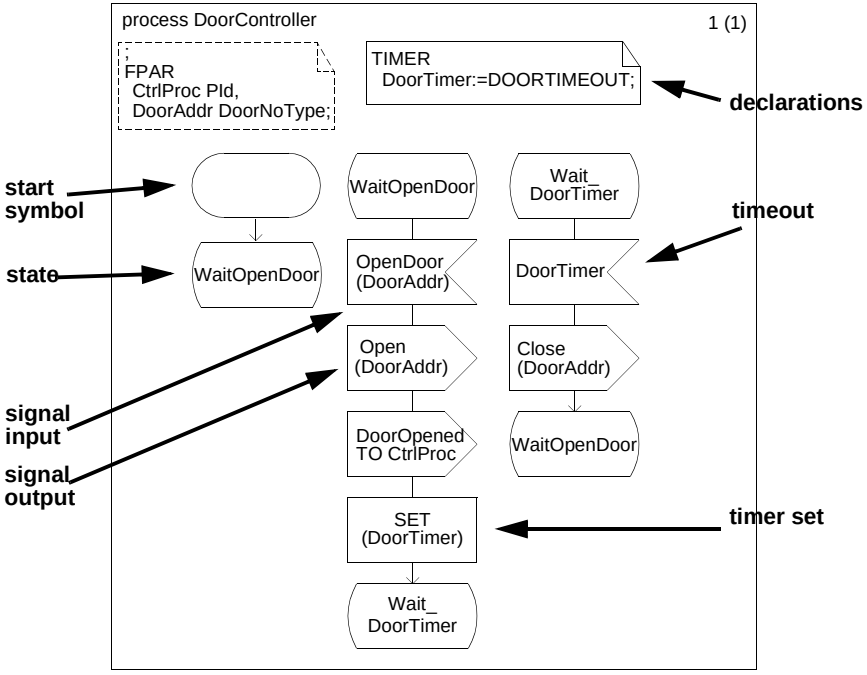


*Figure 622: Behavior described by an SDL process*

A process must have one start symbol. Since a process is a state machine, a transition between two states is made only after a signal has been received. If there are no incoming signals the process is inactive in a state. In SDL a transition takes no time. To be able to model time, and to set time restrictions, there is a timer concept. Each process has its own set of timers that can be set to expire on different durations.

## Data

The set of predefined sorts in SDL makes it possible to work with data in SDL in a traditional way:

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

More complex data sorts can be created by using *arrays*, *strings* and *structs*.

Abstract Data Types in SDL can be used for more than representing data, e.g:

- Hide data manipulation
- Hide algorithmic parts of a specification
- Create an interface to external routines

Data manipulation is hidden in *operators*. For a more thorough description on how to use complex data structures with operators in practice, please see [31].

## Structural Typing Concepts

The object-oriented concepts of SDL give you powerful tools for structuring and reuse. The concept is based on type definitions. All structural building blocks can be typed: *system type*, *block type*, *process type* and *service type*. An exception is the *procedure* that is a type in its original form.

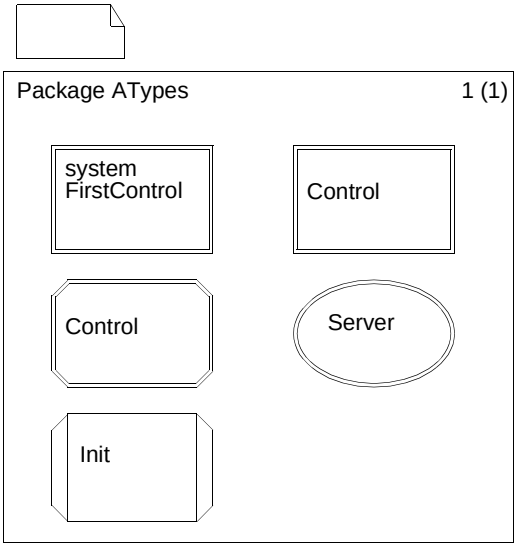


*Figure 623: Package with type definitions*

Type definitions may be placed outside the system in *packages*. *Packages* can be seen as libraries of frequently used functions. The structural typing concepts are shown in Figure 623. All types can inherit from other types of the same kind.

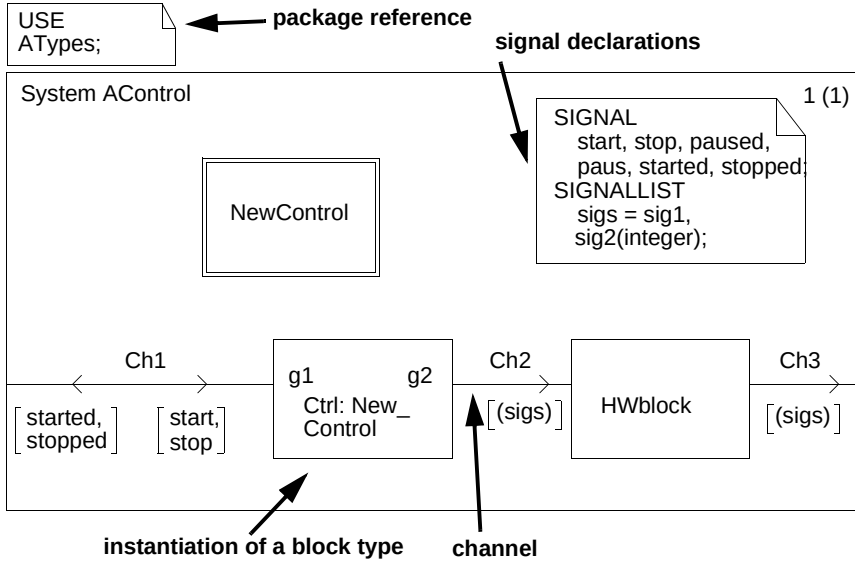


*Figure 624: A system diagram with block instantiation and communication*

One of the major benefits of using an object oriented language is the possibility to create new objects by adding new properties to existing objects, or to redefine properties of existing objects. This is what is commonly referred to as *specialization*.

In SDL-92, specialization of types can be accomplished in two ways:

- A subtype may add properties not defined in the supertype. One may, for example, add new transitions to a process type, add new processes to a block type, etc.
- A subtype may redefine virtual types and virtual transitions defined in the supertype. It is possible to redefine the contents of a transition in a process type, to redefine the contents/structure of a block type, etc.

Figure 624 and Figure 625 describe adding and redefining properties in a system and in a block type while Figure 626 describes the same features in a process type diagram.

To be able to instantiate a type regardless the context (by means of channels), a special concept is needed: *gates*.

Since a channel always has to be connected to a signal route and the connection mechanism lies inside the process, a gate is necessary since it is a way to specify the connection in a transparent manner.

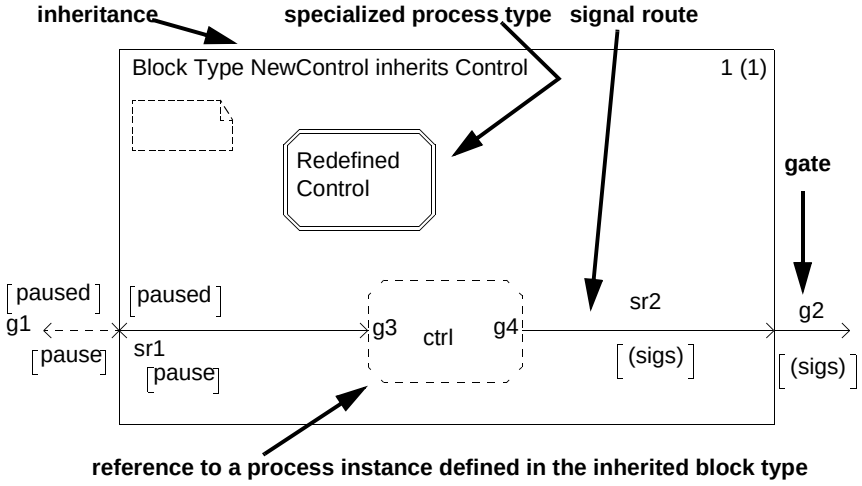


*Figure 625: An inherited block type diagram with process specialization, instantiation and communication*

*In the inherited block type Control, the process ctrl is an instance of the process type Control.*

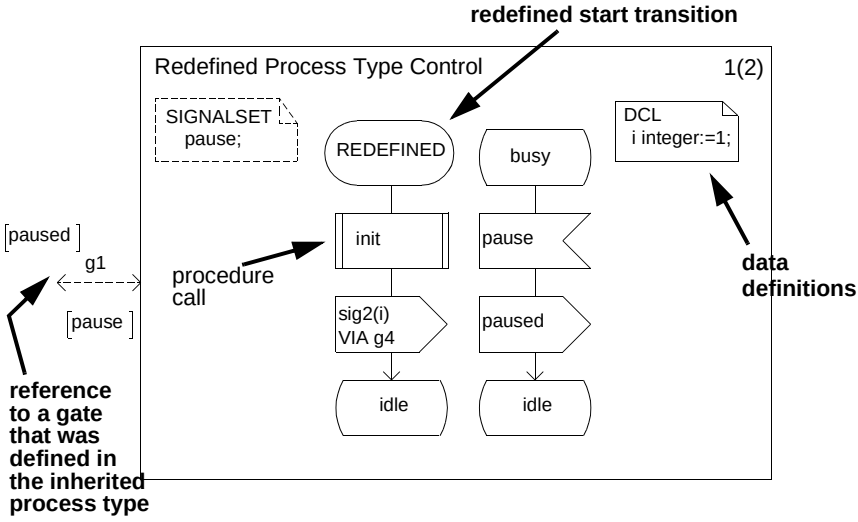


*Figure 626: A specialized process type with added signals, a new transition and a redefined transition*

## Graphical and Textual Notation

The SDL language supports two notations that are equivalent. Beside the graphical representation (SDL/GR), a textual phrase representation (SDL/PR) is standardized.

# TTCN

TTCN (Tree and Tabular Combined Notation) is a special purpose notation to describe test suites [33]. TTCN is a language standardized by ISO for the specification of tests for communicating systems. TTCN has been developed within the framework of standardized conformance testing (ISO/IEC 9646).

With TTCN a test suite is specified. This is a collection of various test cases together with all the declarations and components it needs.

Each test case is described as an event tree. The tree is represented as an indented list in a table. The indentation represents progression with respect to time. See Example 600.

**Example 600: A TTCN test case**

```
     Behavior Description                Constraint        Verdict
1    DuToEnv? Display                    Enter_Card
2     EnvToDu! Card                      Card1
3      DuToEnv? Display                  Enter_Code
4       EnvToDu! Digit                   digit1
5        EnvToDu! Digit                  digit3
6         EnvToDu! Digit                 digit5
7          EnvToDu! Digit                digit7
8           DuToEnv? Unlock              nopar
9            EnvToDu! Open               nopar
10            DuToEnv? Display           Please_Enter
11             EnvToDu! Close            nopar
12              DuToEnv? Lock            nopar
13               DuToEnv? Display        Enter_Card        PASS
14         EnvToDu! Digit                digit1
15          DuToEnv? Display             WrongCode
16           DuToEnv? Display            Enter_Card        PASS
```

Each line consists of a line number, a statement, a constraint reference and a mandatory verdict. A statement can be:

- An event
- An action
- A qualifier

The event statements are statements that can be successful dependent on the occurrence of a certain event, either:

- Receive (represented by a “?”)
- Otherwise
- Timeout

The action statements will always execute and will therefore always be successful:

- send (represented by a “!”)
- implicit_send
- assignment_list
- timer_operation
- goto

A qualifier is simply an expression that must be true if an event should match or an action should be performed.

In TTCN, the communication is asynchronous. The *implementation under test*, IUT, communicates with the environment via *points of control and observation*, PCOs. The interaction occurs at PCOs and are described by *protocol data units*, PDUs, embedded in *abstract service primitives*, ASPs.

At line 1 in s above, the ASP `Display` occurs at the PCO `DuToEnv`. The constraint `Enter_Card` determines exactly which ASP value is to be received.

The leaves in the tree are usually assigned a verdict.

# ASN.1

ASN.1, described in [22], stands for Abstract Syntax Notation One. ASN.1 is a language for the specification of data types and values. ASN.1 is very popular for the specification of data in telecommunication protocols and services, especially in higher (i.e. application oriented) layers. Many telecommunication standards are based on ASN.1. Also TTCN is based on ASN.1. An example of an ASN.1 module is shown below.

**Example 601: An example of an ASN.1 module**

```
ProtocolData DEFINITIONS ::=
BEGIN
- - contains data definitions for an example
protocol

CheckSum ::= INTEGER (0..65535)

DataField ::= OCTET STRING (SIZE (0..56))

PDU ::= SEQUENCE {
        sequenceNr INTEGER (0..255),
        dataField DataField,
        checksum CheckSum OPTIONAL }

END
```

A strong point of ASN.1 is that there are encoding rules that define how an ASN.1 data value is encoded to bits, the most well-known being the Basic Encoding Rules. From an ASN.1 data type definition, functions can be automatically generated that take care of the coding and decoding.

An ASN.1 definition can be imported into SDL as if it was a package. When an ASN.1 data type is imported into SDL, automatically a set of operators that is defined in Z.105 [24] is available for that type.

It is recommended to use ASN.1 for the specification of parameters of signals to/from the environment of the SDL system, especially when encoding rules are to be applied on such signals, or when a TTCN test suite is to be developed. In the latter case the ASN.1 definitions can be directly reused in the TTCN test suite.