SDL Basics

Rolv Bræk[a]

[a]Sintef Delab, N-7034 Trondheim, Norway
E-mail: Rolv.Braek@delab.sintef.no

Abstract

This tutorial is a quick introduction to the basic ideas, concepts and features of SDL. It starts by introducing the class of systems that SDL is intended for and the main features of their behaviour. The correspondence between these features and the basic concepts of SDL are explained. Then the various language elements of SDL are introduced gradually and exemplified by way of a simple system example. Finally, a brief summary of the basic language elements covered by this tutorial is provided. The more advanced features of SDL are explained in the companion tutorial [1].

SDL Basics

1. INTRODUCTION

1.1. The needs

The Specification and Description Language SDL is designed for systems that are:

- reactive;
- concurrent;
- real-time;
- distributed;
- heterogeneous.

The SDL development started early in the era of Stored Programme Control (SPC) switching systems. At that time, in the late 1960s - early 1970s, it became clear to telephone manufacturers and administrations alike that neither programming languages nor hardware description languages provided the basis for the clear and concise communication they needed to successfully develop and maintain the new generations of highly complex switching systems. It is fair to say that the challenges facing those developments were formidable. The technology itself was young and needing to be developed as part of the projects. The applications were highly complex, representing heavy traffic loads and real time demands, and they should be continuously operational with a minimum of malfunctioning. Switching systems were, and probably still are, among the most complex real-time systems in existence. Facing such challenges, it is not surprising that the communication industry became pioneers in the development of systems engineering languages and methods. Nor is it surprising that a language developed to serve such difficult conditions, turned out to be useful far beyond the telecommunication industry for which it was originally intended.

The first issue for the SDL development was to provide a *better* way to describe *behaviour*. Behaviour was in focus because:

- behaviour is vital to the purpose and quality of this kind of systems;
- behaviour is difficult to describe clearly due to its dynamic, invisible and often infinite nature;
- behaviour is often very complex and hard to overview.

Better ways to describe behaviour are still considered to be indispensable from any serious attempt to improve in systems and software engineering. “Better” is here relative to traditional programs and hardware descriptions in supporting:

- human communication and understanding;
- formal analysis and comparison of behaviours;
- alternative implementations and design optimisation.

SDL is intended to be used in several ways:

- for international standards in the communication area: to define unambiguous and

consistent standards;
- for use in tendering: to specify the required behaviour and to compare provided behaviour from different vendors;
- for use in systems development: to specify the required system behaviour, to design and generate an optimal implementation and to document the provided behaviour;
- for verification and validation of behaviour.

Consequently, SDL should be suitable for implementation-independent *specification* of behaviour as well as for *description* of the behaviour actually implemented. Hence the name Specification and Description Language - SDL.

In summary: a language was sought that was both intelligible to human beings, formal enough to support analysis and comparison of behaviours, implementation independent, and realistic in the sense that it could adequately describe real system behaviour. Nothing less!

SDL has developed gradually both in coverage and formality. The first recommendation of the language, appearing in 1976, focused on sequential behaviour, only making a few basic assumptions about concurrency. Later, in the 1984 recommendation, notations for architectural composition were added. Finally, in 1992, came powerful notions of types and inheritance making SDL a full blown object oriented language [5]-[9].

1.2. The concepts

In every language design, assumptions about the problem domain have to be made and mapped to language concepts. Suitability for a given class of problems is largely determined by the conceptual match between the problem domain and the language.

In the case of SDL, the problem domain is reactive behaviour. The need is to say as little as possible about the internal, physical construction of the systems, while telling the full story about how external stimuli and responses are related at the interfaces. The basic concepts of SDL were chosen to match the characteristic features of such interfaces, see Table 1. The choice of basic concepts clearly positioned SDL as an *object-based* language from the very beginning. Also as a concurrent and formal language based on *communicating finite state machines*. The concepts may seem very “physical” at first. Looking closer, however, it is evident that they are not bound to any particular technology. They are neither specific to hardware, nor to software. At the same time they neither exclude implementations in hardware nor in software. In fact, they allow for many alternative ways of implementation giving the designer freedom to optimise for non-functional requirements concerning, for instance, response times and traffic loads. At the same time the underlying theory of communicating state machines provides a firm basis for analysis and comparison as well as an abstraction that enables us to describe control behaviour clearly and realistically.

An important decision was to make the same assumptions about internal interfaces as about external interfaces. Not only does it allow us freedom to place the system boundary where we like, it also provides us with *distribution transparency*. There are no built in assumptions about physical co-localisation of processes. One is free to distribute processes as one sees fit in the

implementation.

Table 1 Characteristic features of the problem domain related to basic SDL.

	Problem domain	SDL concept
Concurrency	External users and other systems operate concurrently and demand to be served concurrently by the system. The system and the environment operate concurrently with respect to each other.	SDL systems are composed from *processes* that behave concurrently with each other. The environment is assumed to be composed in a similar manner.
Communication	The environment interacts with the system through a number of concurrent interfaces. The physical distances vary. The system and the environment interact by signalling, not by direct manipulation.	Processes communicate through concurrent *signal routes* and *channels*. There are no constraints on the distance covered by channels and signal routes. Processes interact (mainly) by *signals* and cannot manipulate the internal state or data of each other.
Sequential behaviour	On each individual interface, the system is expected to behave sequentially in stimuli-response fashion, driven by external events. The responses of the system depend on real-time conditions and on data stored in the system.	Processes are *extended finite state machines* with *timers*. They behave in discrete state transition steps like a finite state machine, but have the additional ability to store and manipulate data, and to supervise time.

While these basic concepts may be sufficient to describe sequential and concurrent behaviour, they are not sufficient for efficient handling of entire system descriptions. The systems we deal with are often very complex and composed from subsystems that may be interconnected, re-used and configurated in many different ways. As any viable language, SDL needs concepts for composition, generalisation and re-use. These are:

composition:
- of *process* behaviours from sub-sequences called *procedures*;
- of *blocks* from processes and *signal routes*;
- of *systems* and blocks from blocks and *channels*;

generalisation and re-use:
- type concepts for signals, procedures, data, services, processes, blocks and systems;
- defining new types by (single) inheritance from other types;
- type libraries as *packages*.

This tutorial will concentrate on the basic concepts and on composition. More advanced use of types and inheritance will be treated in the companion paper [1].

2. THE LANGUAGE

Concepts alone do not make a language. Syntactic form is also needed. Unimportant as it may seem, the syntactic form matters quite a bit for the practical usefulness. SDL has two con-

crete syntaxes:

- SDL/GR which is a graphic representation;
- SDL/PR which is a textual representation.

The graphic form is preferred by most people for actual specification and description work, as it is more intuitive and displays relationships more clearly than the textual form. Fragments in the textual form are embedded in the graphic form where text is more suitable, e.g. for data declarations, signal declarations and operations.

The textual form looks a bit like a programming language. It has the advantage that it does not require any expensive tools, but can be edited using an ordinary text processor. Its main use and purpose is to serve as a vendor independent exchange format between tools. In that role it has the disadvantage that graphic layout information is lost. Therefore a new interchange language, CIF, is currently being developed which also caters for layout information.

SDL makes a rather strict separation between object structure and behaviour. Contrary to some other languages, a process is not just a piece of behaviour. It is an object with a fully defined behaviour that may only be composed in parallel with other objects to form a structure of objects with concurrent behaviour.

SDL deals with structure in a different way than object models in OMT [3], entity-relationship diagrams, or dataflow diagrams, see e.g.[4]. In SDL it is possible to identify instances, and show the architectural structure of a system in terms of its instances, much in the same fashion as a blueprint for a machine or house shows its parts. This feature of the language enables us to model structural distribution and interconnection aspects of a system.

The language makes a clear distinction between types and instances. A type may be defined separately, and instances can be generated in the context of many systems. Unless otherwise stated, we will take the word “process” to mean process instance, and use “process type” to refer to the concept. Similarly for systems, blocks and signals.

3. DESCRIBING STRUCTURE

3.1. System level

In SDL, behaviour is always performed in the context of a *system*. It does not matter if it is a simple behaviour composed from a single process, or a complex behaviour composed from a large number of processes organised in a hierarchy of blocks. The starting point is always a top level description of a *system*. Let us start straight away with a system description. Spend a little time studying Figure 1 before you read on. Are you able to understand anything about the system from reading this top level description?

A system diagram like this, does not tell us very much when studied in isolation. Figure 1 simply tells us that this particular system is composed of one block called *AccessControlUnit*, a set of 100 similar blocks called *ap* of type *AccessPoint* and a block called *op* of type *OperationPoint*.

The (100) *ap* blocks are connected to the environment through (100) channels called *Panel* and (100) channels called *Door,* and to the *AccessControlUnit* through (100) channels called *Validation.* The *op* block is connected to the environment through a channel called *Terminal* and to the *AccessControlUnit* through a channel called *Operation*. The signals passed in each direction over the channels are indicated by signal lists attached to the arrows on the channels.

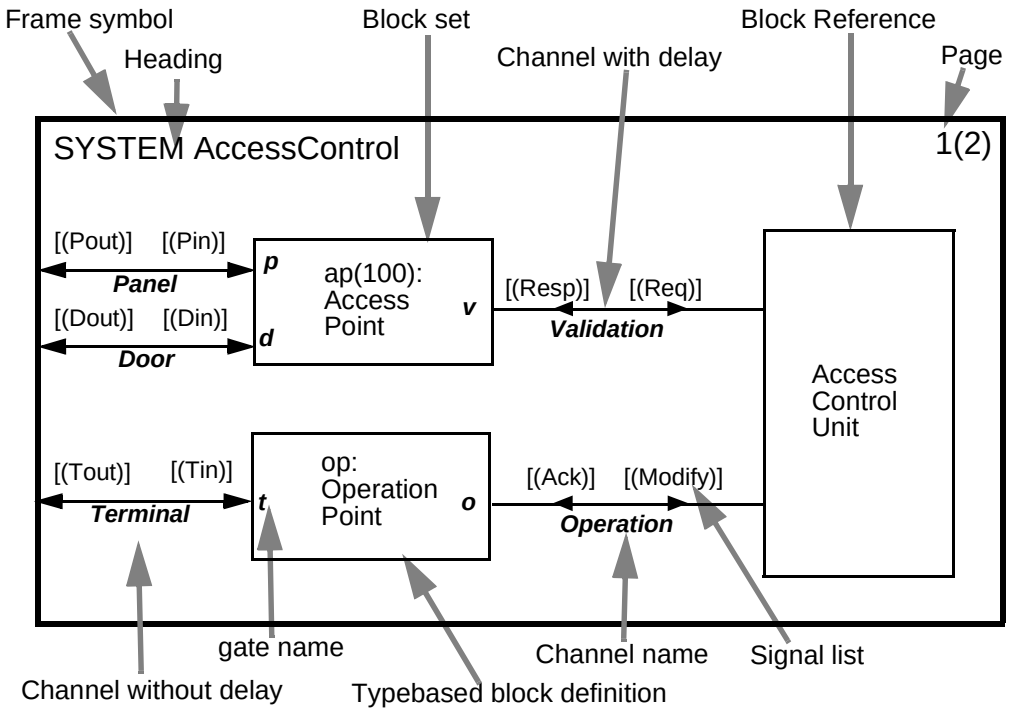


Figure 1. A System diagram example, page 1.

These signals are declared on the second page of the diagram, shown in Figure 2. On this page

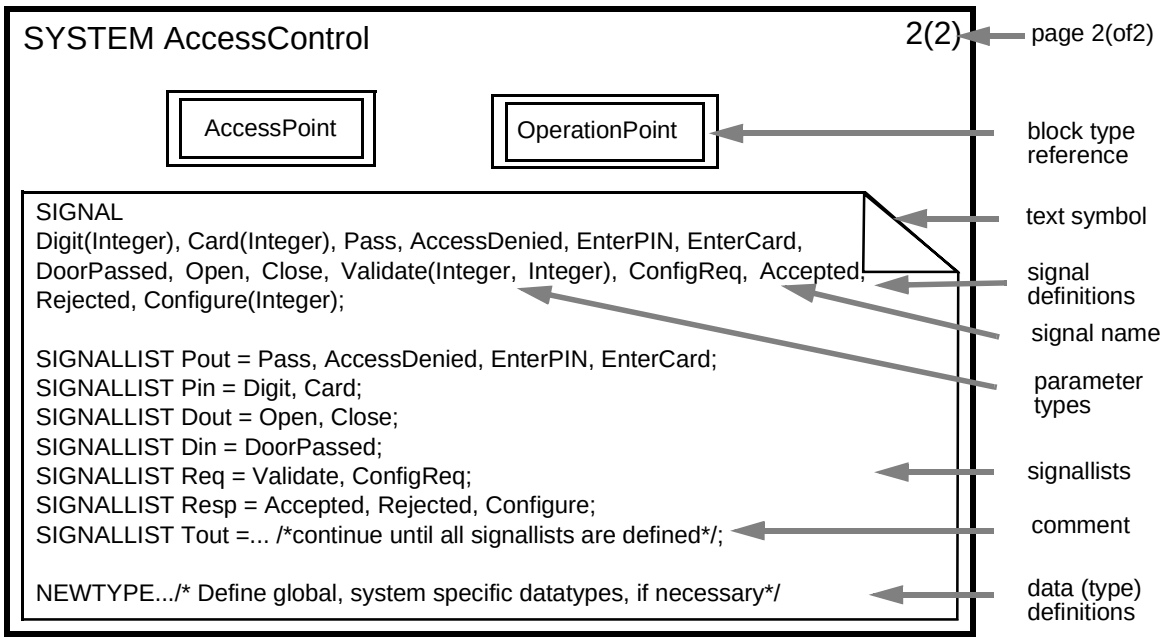


Figure 2. Page 2 of the system diagram.

we also find references to the type definitions for *AccessPoint* and *OperationPoint*.

Altogether this tells us that the system will interact with the environment through many concurrent channels, and that each channel will be served independently by a block. Since each

block will contain at least one process, we know that the system behaviour will contain at least as many concurrent processes as there are blocks. From the system name, block names, channel names and signal names we may guess that the system will provide some kind of access control functionality. Apart from this, the system diagram tells little about the behaviour. The remaining details are hidden inside the block and channel definitions. So, in order to understand more, we must proceed with these. The language elements of system diagrams are summarised in Section 5.1. If you are a newcomer to SDL it may be a good idea to read that section before you continue.

3.2. Block level

Already by presenting the system level, we have indirectly demonstrated some features that are essential to manage complex systems:

- Top down decomposition. We have started to approach the details of the system gradually, in top down fashion. This is a good way to reach understanding of an existing system, although it may not be the way it was developed.
- Bottom up composition. We have seen that the system is composed from user defined components. These components may be arbitrarily complex entities.
- Reusable component types. We have seen that components may be defined separately as types, allowing instances to be instantiated many places with little effort.

There are two important differences between systems and blocks:

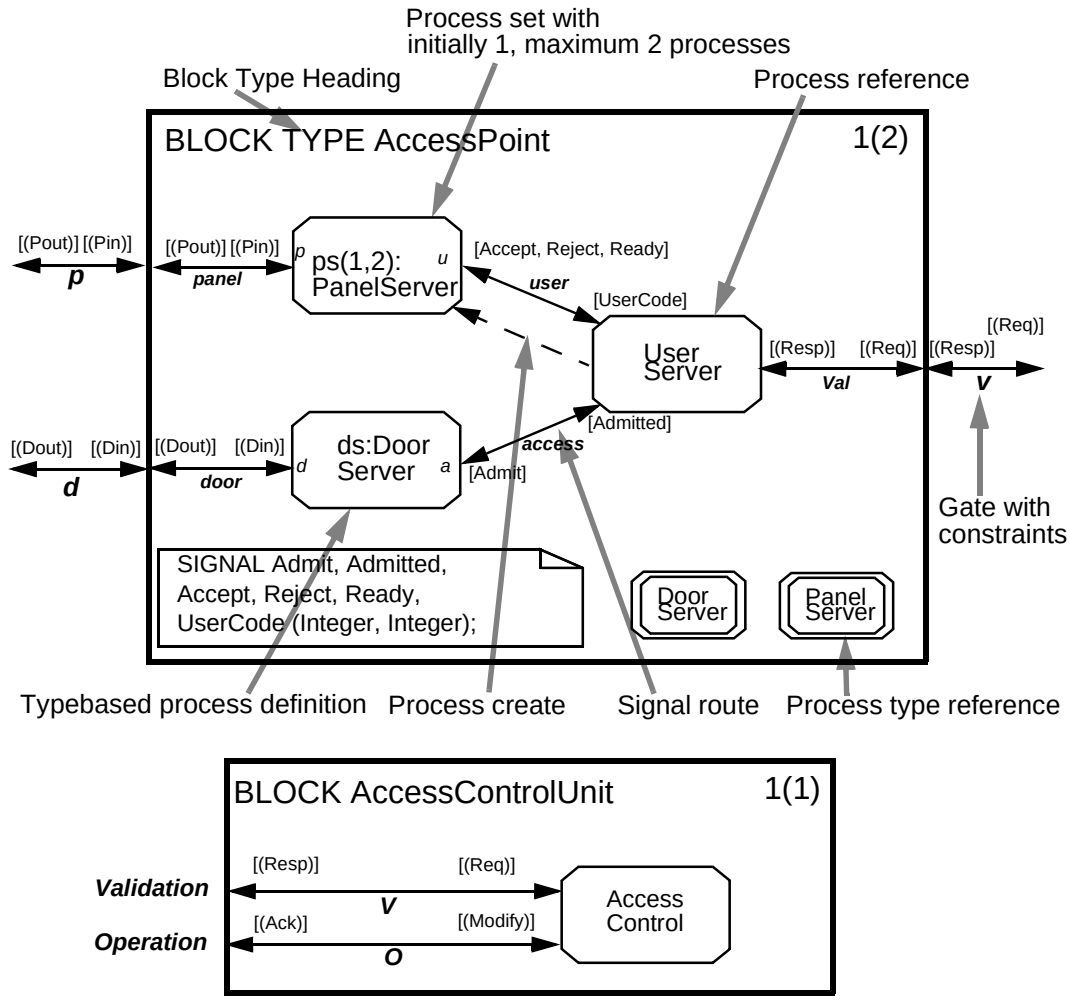
- Blocks can only occur as components inside systems and blocks.
- Systems cannot occur as components neither in systems nor in blocks.

Hence, systems only occur at the top level, while blocks only occur inside. Blocks are containers like systems. They are decomposed into blocks and channels recursively over as many levels as one desires until the basic components, *processes*, are reached. Figure 3 is an example showing a *block type diagram* for *AccessPoint* and a *block diagram* for *AccessControlUnit* (we give no examples of blocks decomposed into blocks here). In these example diagrams we define the blocks in terms of which processes they contain. Again, spend a little time trying to understand these diagrams before reading on.

From the diagrams we learn that each block of type *AccessPoint* contains a process called *UserServer*, a process called *ds* of type *DoorServer*, and one or two processes called *ps* of type *PanelServer*. We also learn that a *PanelServer* may be created dynamically by the *UserServer*. Note that we only need to define local signals here, as those defined on the outer block level are still known.

The *AccessControlUnit* contains a single process called *AccessControl*. In SDL it is not allowed to mix blocks and processes in the same diagram, and it is not allowed to have processes at the system level, only at the block level. This is one reason why the *AccessControlUnit* was represented as a block at the system level, although it contains only one process.

The names we have used so far seem to indicate that we are dealing with a system where the user interface will consist of panels and doors. Indeed, it is a system for access control in buildings, where users identify themselves using a card with a magnetic strip and a personal identification number (PIN). However, this is only informal meaning. Formally, we have only



igure 3. Block type AccessPoint and block AccessControlUnit.

defined some processes, some signal routes and some signals. Only by looking into the process behaviours, we will learn precisely how the system relates to its environment.

Note that the structural decomposition of a system into blocks and processes must be seen primarily as a logical decomposition made in order to precisely define the abstract behaviour of the system. It need not correspond to a physical decomposition of the real system. There are many ways to map an SDL system into a concrete implementation, and some of those will be structured quite differently from the SDL system.

A well known difficulty when describing reactive behaviour is the so-called “state explosion” problem. By decomposing the system into concurrent processes, that are as independent as possible, this problem is considerably reduced.

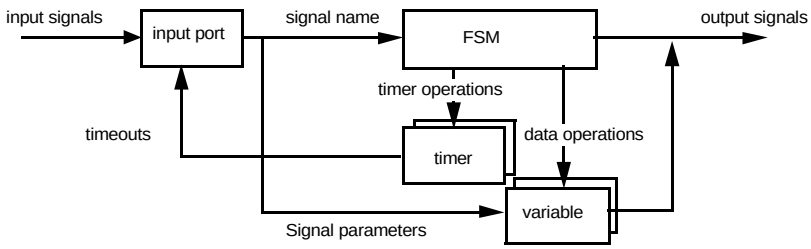
We have now identified processes that behave sequentially and will be able to describe their behaviour without encountering any state explosion. Although the system contains hundreds of independent threads of behaviour, we are now able to concentrate on each thread separately. We are getting closer to the core of SDL now.

4. THE BEHAVIOUR

4.1. The SDL "machine"

SDL processes are described as extended Finite State Machines, FSM. The FSM is excellently suited for the purpose of SDL as it allows stimuli-response behaviours to be described clearly, completely and unambiguously in terms of external stimuli and responses. For this reason the FSM is widely used in such diverse areas as hardware design, protocol design, compiler design and software engineering methods. It has gained a widespread popularity, and is now included in one form or another, in many software engineering methods. SDL is special in the particular form of extended FSM it uses, in its formal definition and in the way concurrent FSMs communicate and may be composed into large systems.

Each process consists of four main parts illustrated in Figure 4: the *input port*, the *FSM*, *timers* and *variables*. The input port contains an unbounded queue of incoming signals and is al-



igure 4. Illustration of an SDL process.

ways ready to accept new signals. A signal arriving at the process is immediately entered into the input port, where it remains until it is *consumed* by the FSM. Signals arriving at a process will be merged into the input port in the order they arrive. If two signals arrive at the same time, the conflict is resolved by selecting an arbitrary sequential order. Signals from independent sources may arrive in any order.The finite state machine consumes signals from the input port in strict FIFO order except when the order is modified by the *save* operator (see the next section). For each signal consumed, it performs one *transition* which will take a short but undefined time. Due to the signal buffering in the input ports, and in delaying channels, communication in SDL is said to be asynchronous.

Signals are discrete messages. Each signal has a name which the FSM uses to select the transition. In addition the signal carries the sender identity and possibly additional data. The FSM will only consume signals while it is in a state. If there are no signals in the input port to consume, it will remain in the state until a signal is entered and consumed. For each signal it consumes the FSM makes a *transition* from one *state* to another state (or to the same state). On each transition it may generate output signals, it may perform operations on the variables and do timer operations. This state-transition behaviour of the FSM is expressed in a *process diagram.*

4.2. The Process diagram

The main content of a *process diagram* is the *process graph.* It specifies precisely how the process will respond to every possible input in every possible state. For each state, a set of transitions is specified. For each transition the *input signals* that may trigger the transition are specified first, then a (possibly empty) sequence of actions is specified before the *next state* is

entered. This is illustrated in Figure 5, Figure 6, Figure 7 and Figure 8 which define the behav-

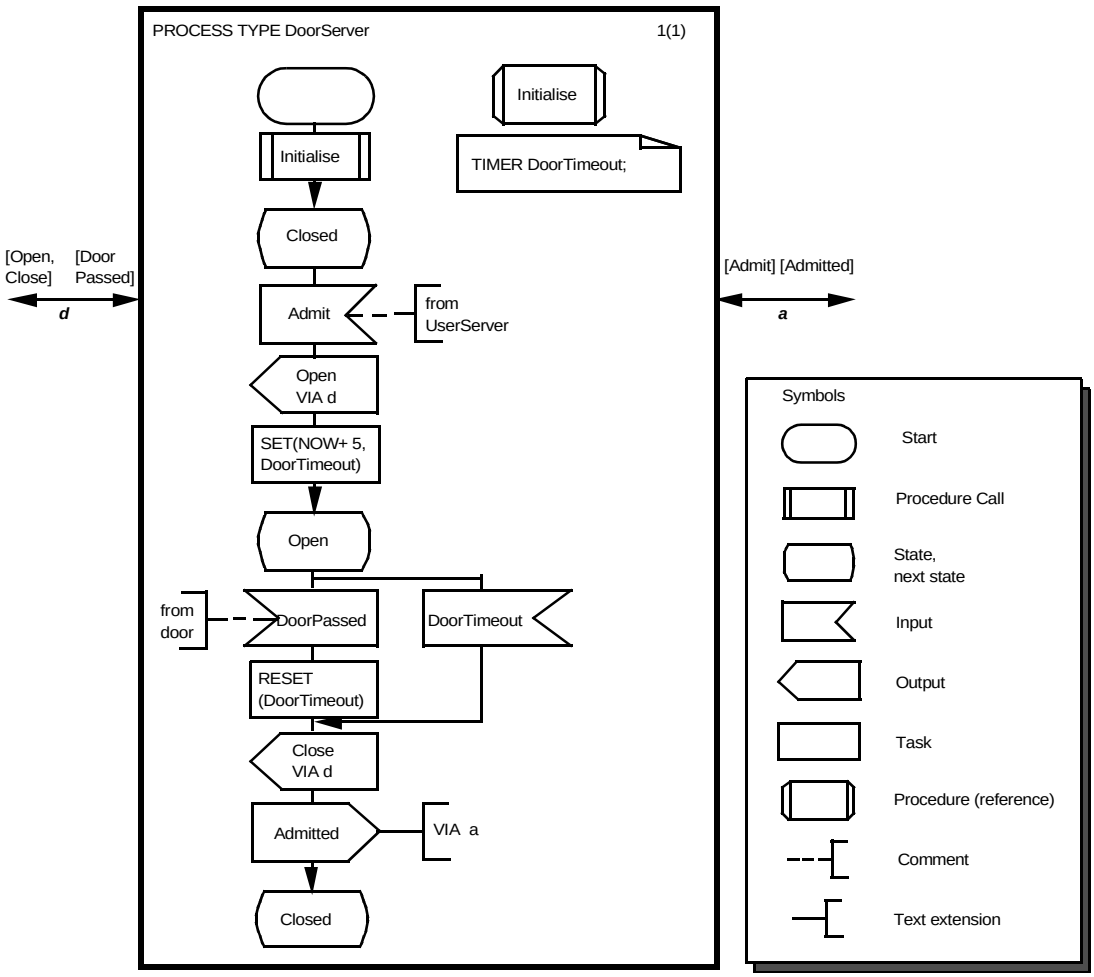


Figure 5. Process diagram for DoorServer.

iour of the three process types in an *AccessPoint*. Try to read and understand these diagrams on your own before reading on. Hopefully your only difficulty will be that you are unfamiliar with some of the symbols. Therefore they are summarised in the figures.

Together these diagrams contain examples of all the most common constructs in process diagrams. First of all there is the usual frame symbol surrounding the process diagram page. This frame separates the process from its environment. The heading consists of the keyword PROCESS or PROCESS TYPE followed by the process name followed by optional parameters.

The normal direction to read a diagram is top to bottom, but deviations from this canonical direction is allowed. Arrow-heads on the *flowlines* between the symbols are optional except when flowlines are merged and when a flowline enters a next-state symbol or an out-connector symbol (not shown here).

The first symbol of the process diagram is the *start symbol*. It is followed by the *start transition* which takes place when the process is generated. A process may be generated either at system start-up, or as a result of a create request from another process. Our *DoorServer* and *UserServer* will be generated at system start-up. So will the first *PanelServer*, but the second will be created dynamically, see the procedure *UInitialise* in Figure 8.

In all these processes initialisation is performed by calling a *procedure*, so as not to clutter the main diagram with initialisation details. Procedures help to achieve top down decomposition as well as bottom up composition of process behaviours. Note that procedure diagrams are referenced using a procedure (reference) symbol in the same manner as block and process types may be referenced.

It is a good idea to concentrate on the stimuli-response behaviour first when reading and making process graphs. What are the states, and what are the main sequences of inputs and outputs specified by the diagram? The *DoorServer* process is the simplest of the three process types in the example, with the purest state transition behaviour. It has no data, but uses a timer called *DoorTimeout*, which is declared inside the text symbol. It has two states: *Closed* and *Open*. After initialisation it will remain in the *Closed* state until it receives an *Admit* signal from the *UserServer*. This signal causes a transition to the *Open* state while performing two actions:

1. The signal *Open* is output *via* gate *d*. This means that the signal will be transmitted through the channel connected to gate *d* and passed on to a process that is able to receive the signal. In this case the receiver will be somewhere in the environment.
2. The timer *DoorTimeout* is *SET*. If the timer has not been reset when the specified point in time, *NOW+5*, is reached, it will enter a signal with the same name as the timer into the input port. NOW is a pre-defined operation that returns the current real-time value.

In the state *Open*, two transitions are specified:

1. The signal *DoorPassed* is consumed causing the process to *RESET* (that is to stop) the timer, output the signals *Close* and *Admitted* and enter state *Closed*.
2. The *DoorTimeout* is consumed causing a similar transition as above, except that the timer is not reset (because it has already expired).

What if the process consumes a signal which is unspecified in the current state? In that case, the signal is simply discarded and the process remains in the same state. This mechanism is referred to as non-persistent input, and is important to bear in mind.

In these process diagrams the destination of output signals has been specified either inside the output symbols, or in *text extension symbols* attached to them. Text extensions can be used with any of the symbols to extend the area available for text. The sender of input signals has been indicated too, in *comment symbols* attached to the input symbols. The fine difference between text extension symbols and comment symbols should be noted: for text extensions the association line is solid while it is dashed for comments. Note that there is no way in SDL to formally specify the sender of input signals. In these diagrams we have added this information as informal comments only as an aid to the reader. Normally when reading process graphs, we like to look up the sender of input signals and the receiver of output signals to increase our understanding of the overall behaviour. From a testing point of view the added information of which process should be the sender is of considerable value.

Apparently what the *DoorControl* process does is to open a door (somewhere in the environment) on command from the *UserServer*, and to close the door after it has been passed or the maximum time has elapsed, and inform the *UserServer* when this has happened.

The *PanelServer* process is slightly more complex than the *DoorServer*. It uses two variables *Pin* and *CardNo* which are both *Integers*. SDL supports the normal data types: *Boolean, Char-*

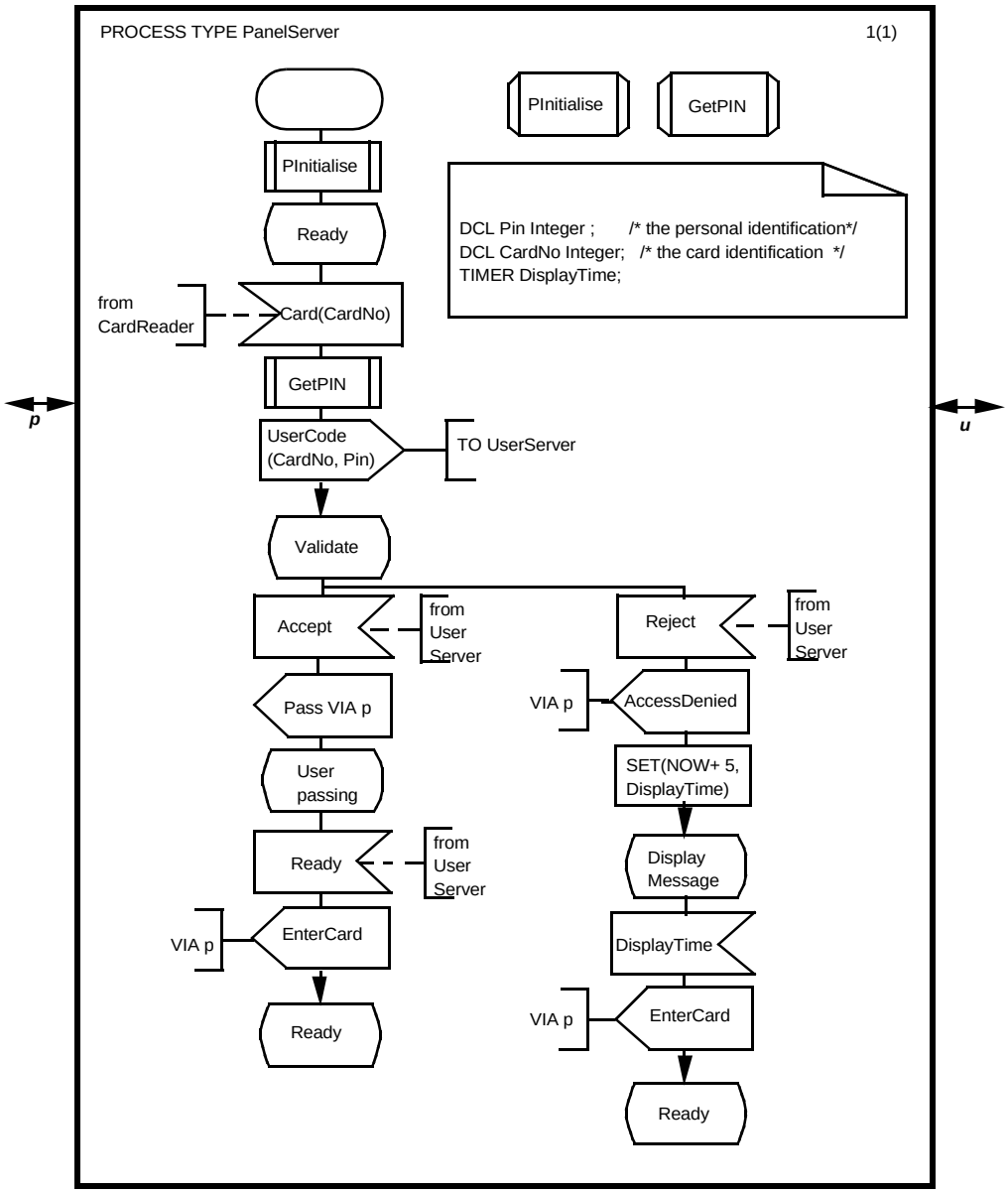


Figure 6. Process diagram for PanelServer.

acter, String, Charstring, Integer, Natural, Real, Array, Powerset. In addition it has a *Struct* concept for structures and user-defined types. An important special datatype is the Process Identifier, the *PId*, which is used to identify processes. There are also types for *Time* and *Duration*.

When the *PanelServer* is in state *Ready*, it expects the input signal *Card* (from a card-reader somewhere in the environment). Note here that the *Card* signal, according to the signal definition, has one integer of data attached to it. This integer is assigned to the local variable *CardNo* upon consumption of the *Card* signal, by the expression *Card(CardNo)* inside the input symbol.

After having received the *Card* signal, the procedure *GetPIN* is invoked. This procedure is not described here. It actually collects *Digit* signals from a keyboard in the panel and constructs

a PIN value which it assigns to the *Pin* variable. The value of the *CardNo* and the *Pin* variables are then communicated to the *UserServer* in the output *UserCode(CardNo, Pin)*. We have here illustrated an important principle: signal parameters are copied into local variables upon consumption, and from local variables into signals upon sending. There is no other way to access the parameters of input signals than to copy them into local variables.

In the *UserServer* we see how the *UserCode* signal is treated at the receiving end. Upon con-

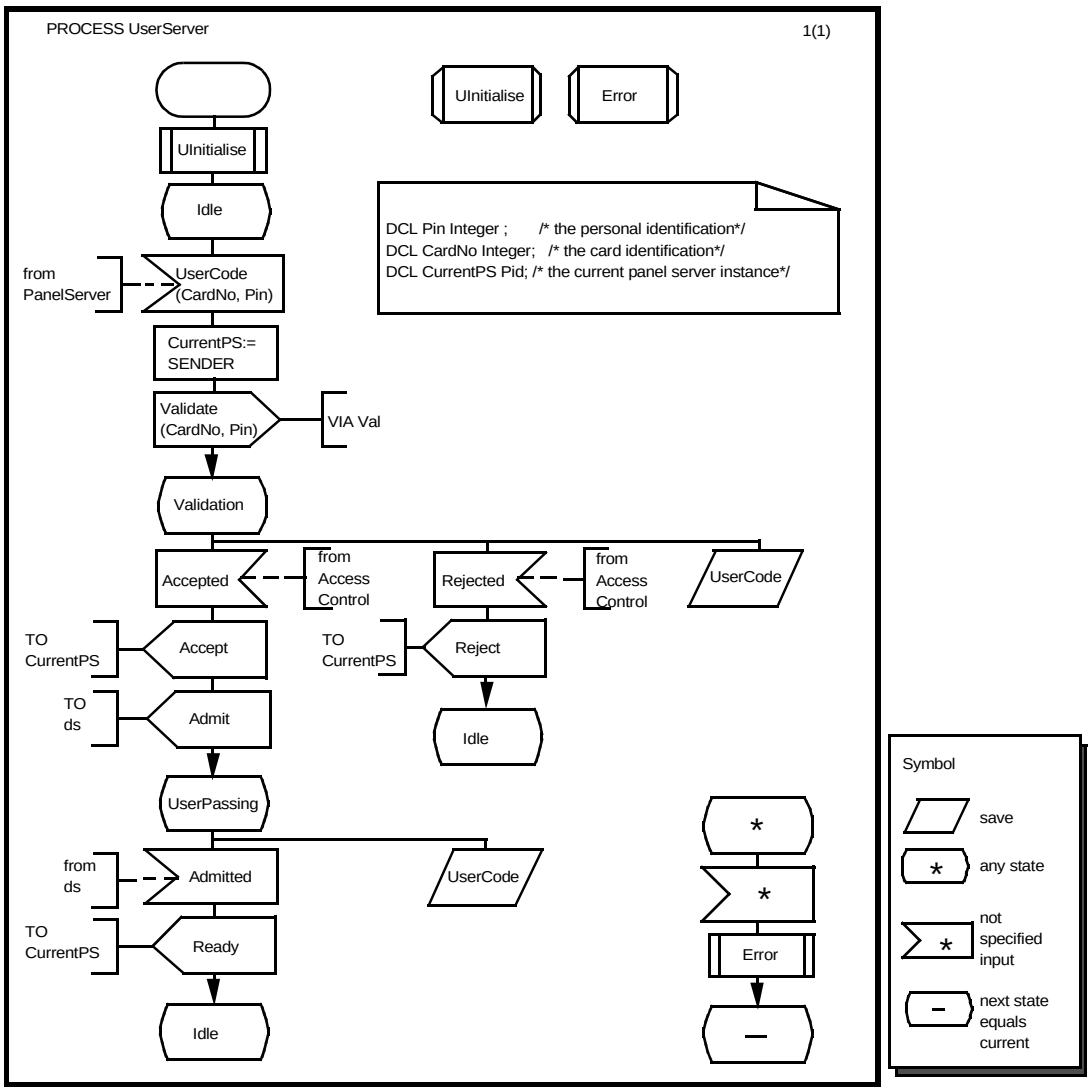


Figure 7. Process diagram for UserServer.

sumption the parameter values are copied into local variables, which happen to have the same name as in the *PanelServer*. They are then sent on to the *Validation* process in the *Validate* signal issued via the signalroute *Val*. Since there may be two panels in an *AccessPoint*, the identity of the current panel is stored by the expression *CurrentPS := SENDER*. (Note that *CurrentPS* is declared as a *PId* variable). *SENDER* is a built-in operation that returns the *PId* of the sender of the input signal just consumed. This value is used to direct the response back to the correct panel later on using the expression *TO CurrentPS* which, in this diagram, is placed in a *text ex-*

tension symbol attached to the output symbols.

The initialisation procedure *UInitialise* is shown in Figure 8. This also illustrates the use of

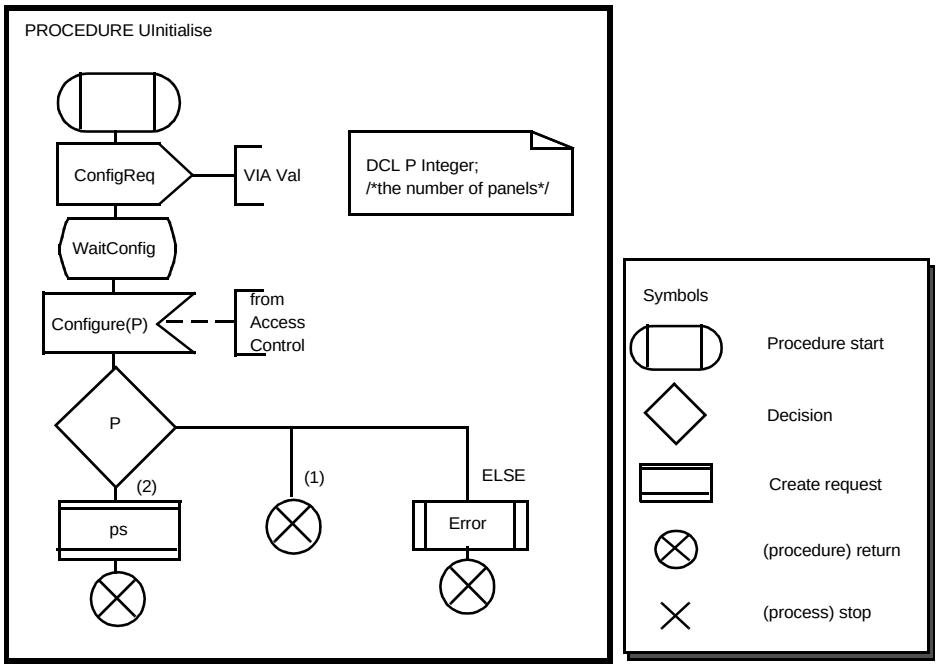


Figure 8. Procedure UInitialise.

decisions and process create. If the configuration parameter *P* is equal to 2, a second *PanelServer* is created.

Since there may be two panels, it is necessary to specify what shall happen if they are both serving users at the same time. In this case we have chosen to only handle one panel at a time in the *UserServer*. Since we have non-persistent input in SDL, we must explicitly specify signals we do not want to consume. This is done by specifying the signals in a *save symbol*. Signals that are saved will be kept in the input port in the order of their arrival until a new state is reached. They will then be treated as normal signals first in the input port, and may either be consumed or saved again. The *UserServer* saves any *UserCode* signals that might be (first) in the input port in all states except *Idle*, and in this way it manages to treat one panel at the time without suffering from state explosion.

It is often desirable to have a well defined action to perform when unspecified signals are consumed. This can be done by adding a transition for "*" (asterisk) input, i.e. every input not explicitly specified in every state. If the treatment is the same in every state, this can be done as simply as illustrated in Figure 7 which also illustrates two important and useful special state names: asterisk "*" and dash "-". The asterisk means "all states" and may be followed by an exception list of state names in parentheses. The dash state name means "the most recent state". Thus, for every state in *UserServer*, if unspecified consumption occurs, the procedure *Error* is invoked and the next state remains the same as the current state.

We have now covered most of the symbols in the process diagrams, and you are now able to read and understand processes described in SDL. More details may be fond in Section 5.3. For more advanced use of SDL, see [1]. Methodology guidelines can be found in [2] and [10].

5. LANGUAGE SUMMARY

5.1. System diagrams

A system diagram is illustrated in Figure 1 and Figure 2.

Frame Symbol

The system itself is represented by a frame symbol which represents the boundary of the system. Outside the frame is the *environment*, which is not described. Similar frame symbols are used in all the different SDL diagrams to delimit the entity being defined from its environment.

Heading

In the upper left corner inside the frame, we find a *heading*. It consists of a text which starts with the keyword SYSTEM followed by the name of the system, here *AccessControl*. The heading is also a common feature of all diagrams, but the keyword will depend on the actual kind of entity being defined.

Page numbers

In the upper right corner there are two numbers. These are *page numbers*. The first number is the number of the current page, i.e. 1 for the first page, and the second number is the number of pages this diagram consists of, i.e. 2 pages in this case. SDL diagrams may in general, take up several pages and the page numbering is always done in the same way.

Block interaction area

Inside the frame, we find the body of the system defined in terms of *blocks* and *channels*. It also defines the channels connecting the system to the environment.

- *Block symbols.* Blocks are represented by rectangular symbols, each representing either a single block or a set of similar blocks. The content of the symbol may take three different forms:
  i. *Block reference*. This is simply a name. The name refers to a separate block diagram which contains the actual definition of the block. This is always a singular block.
  ii. *Typebased block definition*. In this case the symbol contains a block name and a block type name separated by a colon, and an optional number of instances. It also contains one or more *gate names*. The block type which is referred to may either be defined locally, as is the case here, or in a surrounding scope unit. Since block types are defined independently of how instances are interconnected, they have gates to which external channels may be connected.
  iii. *Block diagram.* It is possible to define the block directly nested inside the block symbol. This is not shown in our example, and is not very practical for larger systems.
- *Channel symbols.* The only way to pass signals between blocks are through channels. Channels are represented by lines with arrowheads indicating the direction of signal flow through the channel. The signals of a channel are denoted by a list of signals (and signallists) in brackets. The flow may be either uni-directional or bi-directional. There are two kinds of channels:
  i. *Channels with no delay*. In this case the arrowheads are placed at the ends.
  ii. *Channels with delay*. In this case the arrowheads are not placed at the ends, but somewhere along the lines. These channels have an implicit FIFO queue in each direction, and may delay a signal for an arbitrary time.

  Both kinds of channel will deliver the signals at the destination endpoint in the same

order as they were entered into the channel. If two signals are presented simultaneously to the channel they will be entered into the channel in arbitrary order. A channel connected to a block set will actually represent a set of channel instances, thus there are actually 100 instances of the *Validation* channel in our system.
Every channel (or set of channels) must have a name, and the channel endpoints are attached to the blocks that the channel connects to or to the frame symbol in case the channel is a link to the environment.

Type in system area.

An example of this area is found at the top of Figure 2. Here we define block types, process types, service types and procedures which are local to the system. It is possible to define types either directly, or by reference. In Figure 2 the type definitions for *AccessPoint* and *OperationPoint* are referenced using a double sided rectangle containing the type name. The purpose of these symbols is to indicate where the type definitions are localised without having to provide the full definition at that point.

Text symbols

Text symbols contain declarations and informal text. They are not specific for system diagrams, but occur in all kinds of SDL diagrams. In system and block diagrams text symbols will contain:

- *Signal definitions.* In SDL it is necessary to declare all signals such that they are visible to the processes which handle them. Signals are declared inside text symbols. The declaration gives the signal name and the type of its parameters, if any.
- *Signallists.* Often the list of signals associated with a channel is quite comprehensive and the diagram becomes crowded. *Signallists* help to remedy this. A signallist is a list of signals which has been given a name. The list may also include timer signals or other signallist names. If a signallist contains other signallists, the signallist names will appear in parentheses.
- *Data definitions*. Definitions of local data types.
- *Notes*. A note is an explanatory text embraced by /* ... */.

5.2. Block diagrams

Block diagrams are illustrated in Figure 3. Block diagrams are very similar to system diagrams. There is a frame symbol, an interaction area, a type definition area and text symbols. The differences are the following:

- *Heading.* The keyword is either BLOCK or BLOCK TYPE. If it is “block type”, the name may be followed by *formal context parameters* and a *specialisation*. This will be explained in the paper [1].
- *Gate definitions.* Gates are identified by a gate name and a gate symbol. These are like signal route symbols, but attached outside the frame symbol. Gates can be compared with “plugs” of household appliances. They are used to “plug” up instances correctly. The (optional) signallists help to ensure that the instances of the block type are connected properly to their surroundings. Note that gates are used only with types. The symbols outside the frame of a singular block like *AccessControlUnit*, represent the actual channels connected to that block.
- *Block substructure area*. Here the body of the block is defined in terms of blocks and channels in the same way as the body of a system diagram. This alternative is not

shown in our example, but is used in more complex cases where we need to decompose blocks over several levels before reaching the processes. A block contains either a process interaction area (see below) or a substructure area. It may also contain both, in which case they are considered as alternative definitions of the block.

- *Process interaction area*. Here the body of the block is defined in terms of *processes* and *signal routes*. Signal routes look exactly like channels without delay. They are defined in the same way and behave in the same way. Process symbols differ slightly from block symbols, so it is easy to see that it is a different kind of entity. In similar fashion as blocks, the processes appear in three forms:
  i. *Process reference*. This is a process symbol containing a name which refers to a process definition in a separate diagram. It may optionally specify a minimum and maximum number of instances, where the minimum number is created when the system is created, and the remaining may be created dynamically.
  ii. *Typebased process definition*. This is a process symbol containing a process name and a type name separated by a colon, and optionally a range of instances in the same way as above.
  iii. *Process diagram*. This is a diagram, defining the process behaviour, directly in the process interaction area. This option is only useful for very small diagrams and is not illustrated here.
- *Type in block area*. As in system diagrams, this is the place for local definition of block types, process types, service types and procedures.
- *Data definitions*. Block types may contain data type definitions, but no variable declarations.

5.3. Process diagrams

Process diagrams follow the same overall principles that are used in all SDL diagrams: there is a frame symbol and a heading. Outside the frame are gates, if the diagram defines a process type. Inside we find an area containing the process graph itself, we find text symbols containing declarations, and we find (procedure) reference symbols. The following is a brief summary of the main elements that are special to process diagrams.

Heading

The keyword is either PROCESS or PROCESS TYPE. Processes may have parameters. These are local variables that will receive initial values when the process is created.

Process graph

- *Start.* There is one and only one start symbol per process diagram. It is followed by the start transition which is triggered when the process is created.
- *State, nextstate.* States are designated by a state name. State symbols with the same name represent the same state.
- *Input*. Input symbols specify the signal name and also to which local variables the signal parameters shall be assigned, if any.
- *Output*. Output symbols specify the signal name and the value of signal parameters, if any. In addition they may specify the destination process by a *TO-clause* and/or a *VIA-clause*. The TO-clause specifies the destination process either by name (e.g. *TO ds* in Figure 7) or by a PId value (e.g. *TO CurrentPS* in Figure 7). The VIA-clause is used to list the path of signalroutes and channels which the signal will be sent through (e.g. *VIA Val* in Figure 7). The VIA-clause may also specify a gate. If the TO- and VIA-

clauses are omitted, there must be a unique destination for the signal based on its signal name.

- *Save*. The save symbol lists the names of signals that shall be saved, and not consumed, in a state.
- *Task*. A task is used to set the values of data by *assignments.* A task symbol may contain a list of assignments separated by commas. Alternatively a task symbol may contain *informal text*, where formal assignments are not considered appropriate. The right hand side of the assignment operator symbol represents the expression. Variable occurrences on the right of an assignment operator mean extracting the value from the variable. Variables on the left of the assignment operator are modified to become the expression value of the right hand side.
- *Timer operations.* Timer operations are placed inside task symbols. When a timer has not been SET, it is inactive. When it is SET, it becomes active. A timer is set with a TIME value. If an active timer is SET it continues to be active with the new TIME value. RESET causes the timer to become inactive and ensures that the timeout signal will not be consumed by the process.
- *Decision*. The decision symbol is used to choose between different alternative courses of action upon a question. The question is written within the decision symbol and may be either a formal expression (such as the one in our example) or informal text. On each of the flowlines from the decision symbol there is an answer. The course of action will follow the line where the correct answer is associated. Decisions change the control flow according to values of internal variables, while state/input constructs change control flow according to external stimuli. Often it is a matter of design whether we use state and signals or decisions.
- *Procedure call*. The procedure call symbols resemble the procedure (reference) symbols, but their corners are different. Note that a procedure call symbol has one and only one entrance and one and only one exit.
- *Create request.* This causes a process instance to be created and given initial parameter values. Only members of process sets where the maximum number of instances have not been reached may be created. Note that blocks cannot be created dynamically. In block diagrams (see Figure 3) the creation may be shown by a dashed line from the *parent* process to the *offspring* process.
- *Stop*. While processes are created by their parent (or implicitly at start-up time), processes may not kill each other. Processes terminate if they reach a Stop symbol. When a process has terminated, any attempt to reference that process will result in an error.

Text area.

- *Variables*. Variables in SDL are declared very much the same way as in programming languages. *DCL* is the keyword which precedes data declarations. A list of names follows that is terminated by the sort name. *Sort* is the SDL technical term for what is normally called data type.
- *Timers and Time*. A *timer* is declared similarly to a variable. *Time* is a special data type and is mainly used in connection with timers. The expression *"NOW+5"* is a time value, and it adds the time-expression *NOW* and the *Duration 5* (here seconds). NOW is an operator of the time data type and it returns the current real-time. *Duration* is another special data type and it is also mainly used in connection with timers. Timers are just like alarm clocks. They will issue time-out signals when their time is reached.

There may very well be several different timers active at the same time.

- *Process Identifier, PId*. Each SDL process has a unique identification which is a value of the data type *PId*. PId expressions are used mainly as the destination of output. PId values are obtained from the following PId expressions which are predefined in all processes:
  i. *SELF* – this process itself.
  ii. *OFFSPRING* – the most recent process instance created by this process.
  iii. *PARENT* – the creating process, if this process was dynamically created.
  iv. *SENDER* – the sender of the signal most recently consumed by this process.
- *Data types.* SDL has predefined types which are quite similar to those we know from programming languages. In addition it is possible to define new data types using generators and structures. It is also possible to define entirely new types, using an axiomatic approach.

Comment and text extension symbols

These are very similar, but they are distinguishable by the association line. The association line for a comment symbol is dashed, while for text extension it is solid. As the name suggests, a text extension is an extension of the text within the symbol.

Type in process area.

This area contains service type diagrams and/or procedure diagrams, or references to such diagrams. Only procedure (reference) symbols have been shown in the examples here.

Procedure diagrams.

Procedures are *scope units* and they may have their own variable declarations. Procedures contain the same mechanisms for describing behaviour as are found in processes, but procedures are not actors of their own. The interpretation is performed by the process in which they are declared. Special features of procedure diagrams are:

- *Parameters*. Procedures are defined with formal parameters that receive actual values when the procedure is invoked. Parameters may be passed either by value, which is the default, or by reference.
- *Procedure start and Return symbols*. Two special symbols represent the start of the procedure and the return from the procedure.

5.4. Additional features

SDL offers a number of features in addition to those presented here. A few of them are briefly mentioned below.

Shared variables

We have emphasised that data is not shared among the processes. However, there are situations where several processes need to access the same variables. In such cases the variables are still owned by one and only one process. Other processes may gain access to the values, either by normal signal exchange, or by two special constructs:

- *Reveal/view*. By declaring a variable as *revealed*, it is possible for other processes in the same block to *view* (but not change) the value.
- *Export/import*. By declaring a value as *exported*, it is possible for processes even in other blocks to see the value by *importing* it. This export/import construct is actually a shorthand for queries using normal signals.

Services

Sometimes the behaviour of a process is composed from sub-functions or services that are

quite independent, and possible to describe in separate process graphs. As an alternative to describing the behaviour of such a process as one finite state machine in one process diagram, SDL offers the opportunity to decompose the process into several finite state machines, called *services*, which are described by separate process graphs. The services share the input port of the process, and may also share variables. Only one service at a time is allowed to perform a transition, so there is no danger of interference on shared variables. Services cannot be addressed from the outside, so input signals are directed to the correct services on the basis of the signal names only. Consequently the services must be uniquely identified by the signal name.

Remote procedures

SDL offers a remote procedure construct that allows processes to call procedures which belong to other processes. Like export/import, this is not a basic mechanism in SDL, but a shorthand for an underlying communication using normal SDL signals.

Enabling conditions

It is possible to make signal consumption dependent on so called *enabling conditions*, which are boolean expressions placed in a special symbol on transitions. Such transitions will only be performed if the expression evaluates to true.

6. ACKNOWLEDGEMENT

The example presented in this tutorial has been inspired by a similar example in the SDL methodology book by Bræk and Haugen [2]. The author is grateful to Øystein Haugen and Richard Sanders for thoroughly reviewing this tutorial at an early stage and for giving many valuable suggestions.

7. REFERENCES

1. Sarma, A. (1995) *SDL-92*. Tutorial at the 7th SDL Forum. Ibid.
2. Bræk, R. and Haugen, Ø. (1993) *Engineering Real Time Systems. An object-oriented methodology using SDL.* Hemel Hempstead: Prentice Hall. ISBN 0-13-034448-6.
3. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W (1991) *Object-Oriented Modelling and Design.* Englewood Cliffs: Prentice-Hall.ISBN 0-13-630054-5.
4. Yourdon E. (1989) *Modern structured analysis*. Englewood Cliffs: Prentice-Hall. ISBN 0-13-598632-X.
5. Z.100 (1994), *CCITT Specification and description language (SDL)*, ITU-T.
6. Z.100 C (1994), *Initial algebra model. Annex C to Z.100*, ITU-T.
7. Z.100 D (1994), *SDL predefined data. Annex D to Z.100*, ITU-T.
8. Z.100 F2 (1994), *Specification and description language (SDL) - SDL formal definition: Static semantics*, ITU-T.
9. Z.100 F3 (1993), *Specification and description language (SDL) - SDL formal definition: Dynamic semantics*, ITU-T Apr. 94.
10. Reed, R. *Methodology for Real Time Systems*.Tutorial at the 7th SDL Forum. Ibid.