13 Tutorial on SDL

Introduction **2**
Overview of SDL **3**
SDL as an object oriented language **22**
SDL by example **36**
List of figures **67**
List of definitions **68**

SDL Tutorial

Introduction

This SDL tutorial provides a number of different approaches to learning SDL:

- Overview of SDL (p.13-3)
  - follow this if you would like an overview of the language elements of SDL, with an example as illustration; this is organised according to what instances an SDL system consists of, how they are defined, how SDL systems are structured, how processes communicate, how subtypes can be defined and how data types may be defined.
- SDL by example
  - follow this if you would rather see a complete example right away and have the language explained through this; this is organised as a top-down introduction where you start from a system diagram, through block diagrams, to process, service and procedure diagrams.
- SDL as an object oriented language
  - follow this if you would like to learn how SDL elements correspond to the various elements of object orientation; this requires a minimal knowledge of object orientation, and you will learn what corresponds to objects, attributes, classes, subclasses, etc.

Behind all these perspectives on SDL lies a number of *definitions* of the various language concepts and most of the figures in the electronic form of this chapter will be sensitive for mouse clicks and provide the definitions of the language element you click at at in the diagrams.

In addition the Chapter provides you with a subset of the more formal definition of SDL. Whenever you in the electronic form encounter a text like this “Z.100”, clicking on this will bring you part of the Z.100 correspondence to the topic you are reading.

TIMe Electronic Textbook v 4.0 © SINTEF Modified: 1999-07-16

Overview of SDL

Introduction

SDL is used for specification of systems. This is done by making SDL systems that are models of existing or potential systems. SDL systems are specified by SDL system specifications.

As part of the modeling process, components of systems are identified and modeled by *instances* as parts of the corresponding SDL system. Categories and subcategories of components are in SDL represented by types and subtypes of instances.

An *SDL* system *consists* of a set of *instances*. Instances may be of different kinds:

- blocks containing other blocks (which in turn may contain other blocks) or processes;
- processes characterised by attributes in terms of variables and procedures and which exhibit behaviour in terms of Extended Finite State Machines;
- services being parts of processes, with the same properties as processes, but being executed as part of the containing process execution.

SDL system can be *structured* by various means. A system consists of a number of blocks connected by channels, each block may contain a *substructure* of blocks (to any depth) or it may contain process sets connected by signal routes.

Processes execute concurrently with other processes and communicate by exchanging signals; or by remote procedure calls. Reception of signals and requests for remote procedures are the events that t rigger state transitions in the behaviour of processes. Services as part of processes execute one at a time like co-routines.

Variables are defined by means of data types:

- abstract data types that may be both predefined and user defined
- ASN.1 data types (according to a separate standard Z.105).

SDL specifications can be modularised by means of packages. A package is a collection of type definitions. Packages can be used in the definition of new packages and the definition of systems.

Processes and process types

The primary instances of an SDL system are processes. A process may have attributes in terms of variables, it may have procedures, and it may have a certain behaviour.

A process type defines the properties of a category of process instances.

The process type diagram in Figure 13-1 (p.13-4) is an example of a definition of a process type. Each Controller process will have:

- the variables cur_panel, cid and PIN as declared in the text frame;
- a procedure OpenDoor;

a behaviour defined by states and transitions.
The process type is the central controlling part of an access control system, based on card codes and PIN codes. The behaviour of the Controller takes care of the communication with the user (via a panel), with the central unit (that does the actual validation), and it eventually opens the door (by means of the OpenDoor procedure).

Specifying behaviour: states and transitions

The behaviour of a process is described as an Extended Finite State Machine: When started, a process executes its start transition and enters the first state. The reception of a signal triggers a transition from one state to a *next state*. In transitions, a process may execute *actions*. Actions can assign values to variable attributes of the process, branch on values of expressions, call procedures, create new process instances and send signals to other processes.

Communication by means of sending signals is asynchronous: the sending process does not wait until the signal is handled by the receiver, and the receiving process will keep signals in a queue until it reaches a state in which it is prepared to handle it.

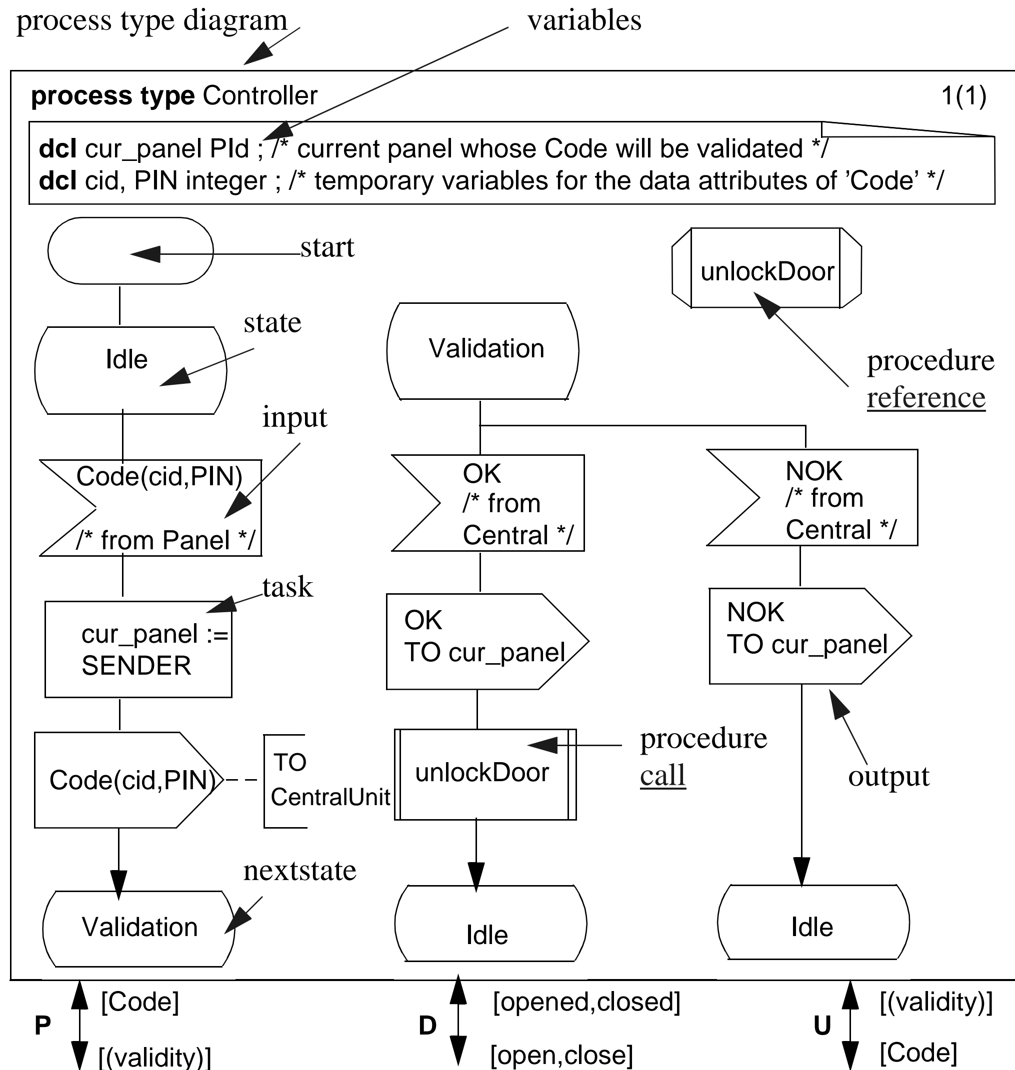
Figure 13-1 (p.13-4) is an example on behaviour specification.

In a state (e.g. Idle) the process takes from the queue the first signal that is of one of the types indicated in the input symbols (here Code, containing information about the card id and PIN from the Panel). The Idle state is followed by one input symbol which describes the consumption of the signal Code. In the transition following the reception of the Code signal, it will use the variable cur_panel to remember from which panel the signal came from and then send the Code to the central unit for validation. The next state is Validation. In state Validation the Controller will only accept OK or NOK. If it gets OK it will open the door by calling the procedure OpenDoor.

Figure 13-1: Behaviour Specification

TIMe Electronic Textbook v 4.0 © SINTEF Modified: 1999-07-16

Open figure



P, D and U on the frame are gates: they define possible connection points for signal routes (see below) that connect specific process sets of this type. The signals (e.g Code) and signal lists (e.g. validity) define which signals may enter/exit through the connection point.

For more details on the constituents of a process type diagram see Virtual process type Controller (p.-59).

Variables

Variables are declared according to data types. Each variable has a name and a data type. Controller processes as defined in Figure 13-1 (p.13-4) have a variable cur_panel of type PId and two Integer variables cid and PIN.

The data type defines possible values, behaviour and operators that can be applied to values of the type. It is possible to define data types.

Predefined types include Character, Boolean, Integer, Natural, PId (Process Instance Identifier), and Real. Templates for defining arrays, strings and powersets are also provided.

Variables of type PId denote process instances, so cur_panel is a variable that denotes a process instance representing the panel that sent the Code signal.

Procedures

Procedures as part of processes define patterns of behaviour that the process may execute at several places or several times during its life-time. The behaviour of a procedure is defined in the same way as for processes (that is by means of states and transitions), a process may have (local) variables, and in addition it may have IN, OUT, IN/OUT parameters.

Procedures are defined by procedure diagrams. The unlockDoor in Figure 13-1 (p.13-4) is thus only a reference to a separate diagram defining the properties of unlockDoor.

For an example on a procedure diagram see Procedure diagram, GetPIN (p.-74).

Communication by means of signal exchange

Processes execute concurrently and communicate asynchronously by sending signals. Each process has a queue of signals. The reception of a signal is the event that may get a processes to perform a transition from one state to another state.

In addition to signals, processes may also communicate by means of remote procedures. On the server side they are treated like signals (e.g. only accepted in states with an input of the procedure), while the client side will be blocked until the remote procedure has been executed.

Grouping of process sets by means of blocks

Types and instances of types correspond to the notion of classes and objects of class in object oriented languages. In addition SDL supports the what would correspond to the grouping of objects into larger units.

A *block* is a container for either sets of processes connected by signal routes, or for a substructure of blocks connected by channels. Each of these blocks may in turn consist of either process sets or a substructure of blocks. This decomposition may be applied to any depth.

There is no specific behaviour associated with a block, and blocks cannot have attributes in terms of variables or procedures. Therefore, the behaviour of a block is simply the combined behaviour of its processes.

A block type defines the common properties for a category of blocks. In Figure 13-2 (p.13-7) a block type AccessPoint is defined by means of the process type Controller (and two other processes: Panel and Door, taking care of the communication with the actual physical panel and door).

TIMe Electronic Textbook v 4.0 © SINTEF Modified: 1999-07-16

Panel and DOOR are defined directly (there will be process diagrams for each of them and the process symbols with Panel and DOOR in Figure 13-2 (p.13-7) are just references to these), while lsc is defined by the process type Controller. The symbol in Figure 13-2 (p.13-7) with the name Controller is a reference to the corresponding process type diagram.

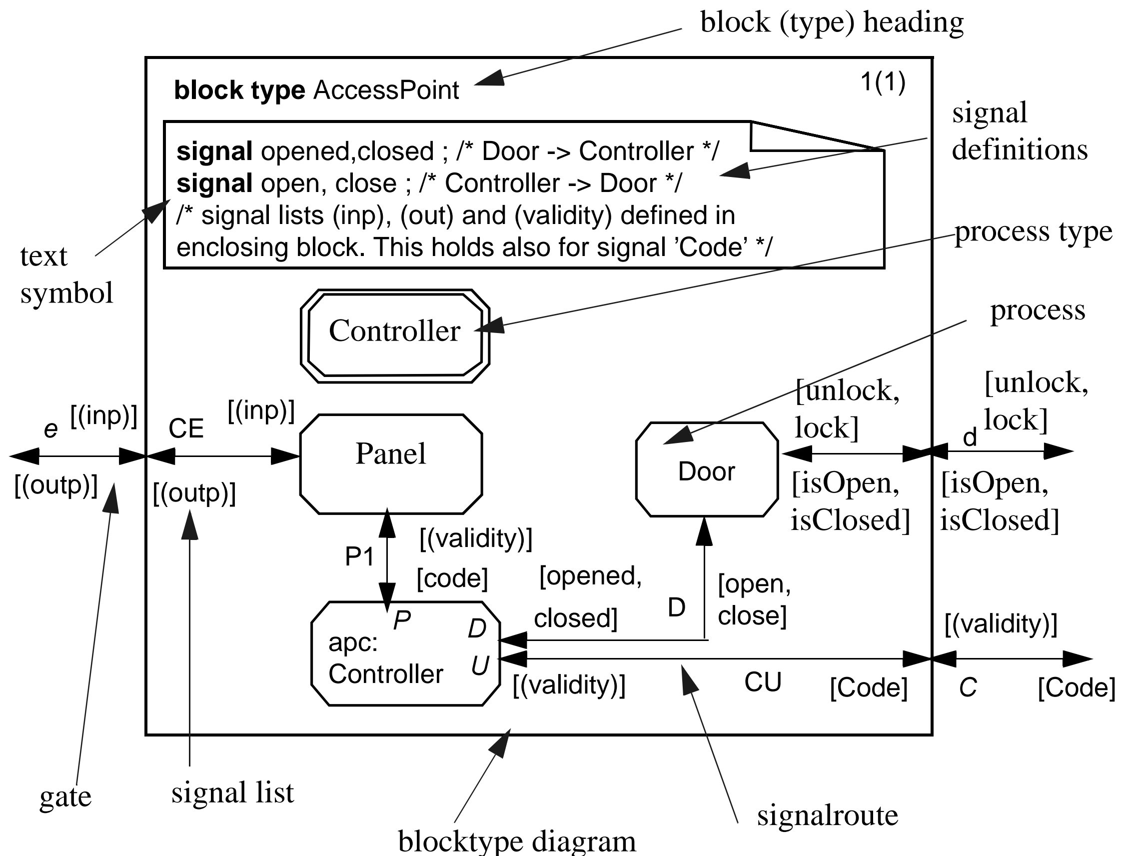
Processes are parts of process sets

Process instances are part of process sets. The specification of a process set includes the *name* of the set, the number of instances (*initial* number and the *maximum* number of instances), and possibly the name of a process *type*. If no process type name is used, then the properties of the processes in the set are defined directly (in the corresponding process diagram). Omitting the number of instances implies that initially there is 1 element, and that the number of instances is unbounded. In Figure 13-2 (p.13-7) there are three process sets of the special kind that initially has just one member:

Panel and Door are defined directly (that is the Panel and Door are references to separate process diagrams), while lsc is the name of a process set according to the Controller process type defined in Figure 13-1 (p.13-4).

Figure 13-2: Block type AccessPoint with processes

Open figure



Process sets are connected by signal routes

In order for processes to interact by means of signals, their sets have to be connected by signal routes. The signal routes and the associated signals (e.g. Code) and signal lists (e.g. validity) only specify possible signal exchanges.

When signal routes connect process sets according to process types, they connect to the gates defined in the types. In Figure 13-2 (p.13-7) the gates P,D,U are the gates defined in Figure 13-1 (p.13-4).

The interaction between processes is specified on the signal routes connecting them, whereas a process type defines gates as connection points for signal routes. The process type Controller defines e.g. a gate P for the connection to Panels, with ingoing signal Code and outgoing signals defined by the signal list validity (OK, NOK). The constraints on the gates (in terms of ingoing and outgoing signals) allows the specification of the behaviour of process types without knowing in which context the instances of the type will be and how they are connected. Gates can only be connected by signal routes that carry the signals of the constraint, in the right directions.

The signal lists are defined below, see Figure 13-4 (p.13-10).

Local definitions in blocks

In addition to containing process sets or blocks, a block may have data type definitions and signal definitions. Signals being used in the interaction between processes in a block may therefore be defined locally to this block (providing a local name space) - here exemplified by opened, closed, open, close.

Blocks as part of blocks

As described in Figure 13-2 (p.13-7) an AccessPoint will have three concurrent processes, each taking care of different roles of the access point. If each of these roles would require more than one process, then they would be represented by blocks which in turn would contain the necessary processes. This is illustrated in Figure 13-3 (p.13-9), but is not used in the following.

The symbols with the names Panel, Door and Controller are block symbols, specifying that each AccessPoint block has three blocks as part of it, connected by channels.

As for signal routes, channels have associated signals (e.g. Code) and signal lists (e.g. validity) in order to specify possible signal exchanges in the corresponding direction.

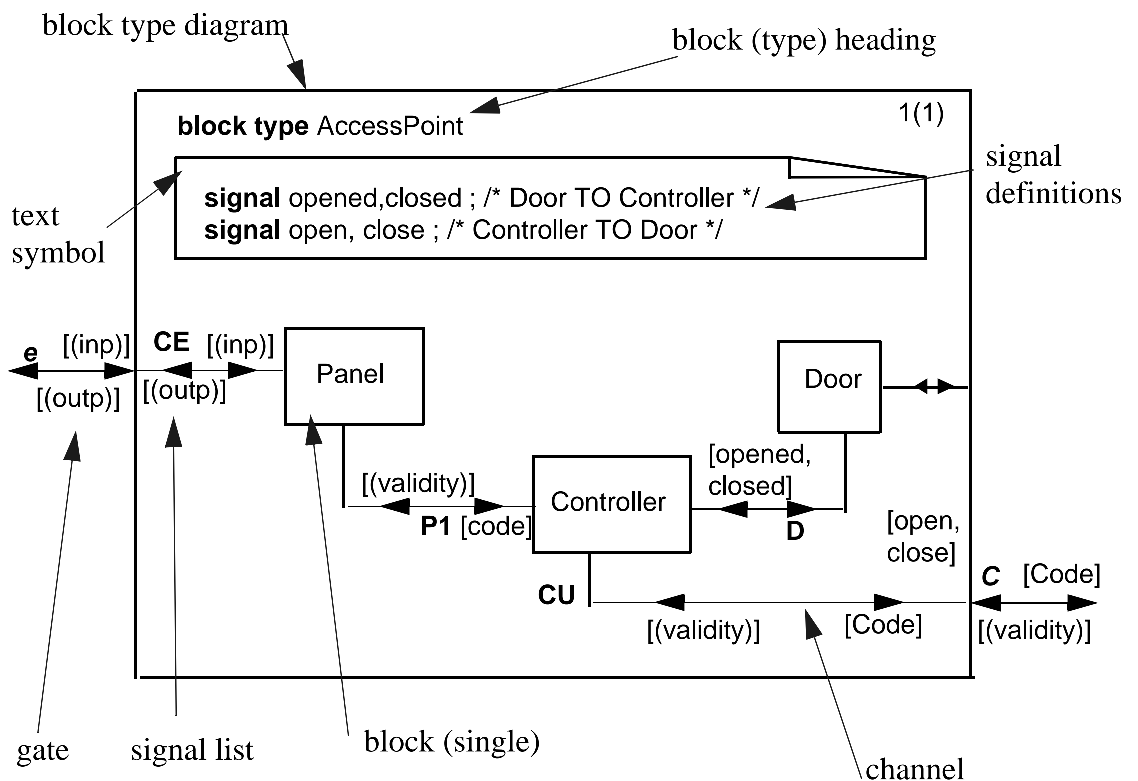
TIMe Electronic Textbook v 4.0 © SINTEF Modified: 1999-07-16

Types, sets and instances

Note the distinction between process *types*, process *sets* and process *instances*. Process types only define the common properties of instances, while process sets have a number of instances. Signal routes connect process sets and not process instances. Process instances are denoted by values of type PId. Process instances have variable attributes and behaviour.

Figure 13-3: Block diagram of AccessPoint with block substructure

Open figure



Systems: set of blocks connected by channels

In order to provide a complete specification of a given access control system with a single central unit and a number (100) of access points according to the block type AccessPoint, a system diagram as in Figure 13-4 (p.13-10) is specified.

A system consists of a set of blocks connected with each other and with the environment by channels. Note that channels connected to a block set according to a block type connect to the gates (e and C) defined in the block type.

The system specified in Figure 13-4 (p.13-10) has interaction with its environment. The signals used for this purpose can be defined as part of the system (as in Figure 13-4 (p.13-10)) or as part of a package used in the system. The system assumes that the environment has processes which may receive signals from the system and send signals to the system.

For more details on constituents of system diagrams see System diagram, Access Control System (p.-45).

Figure 13-4: System design in SDL

Open figure

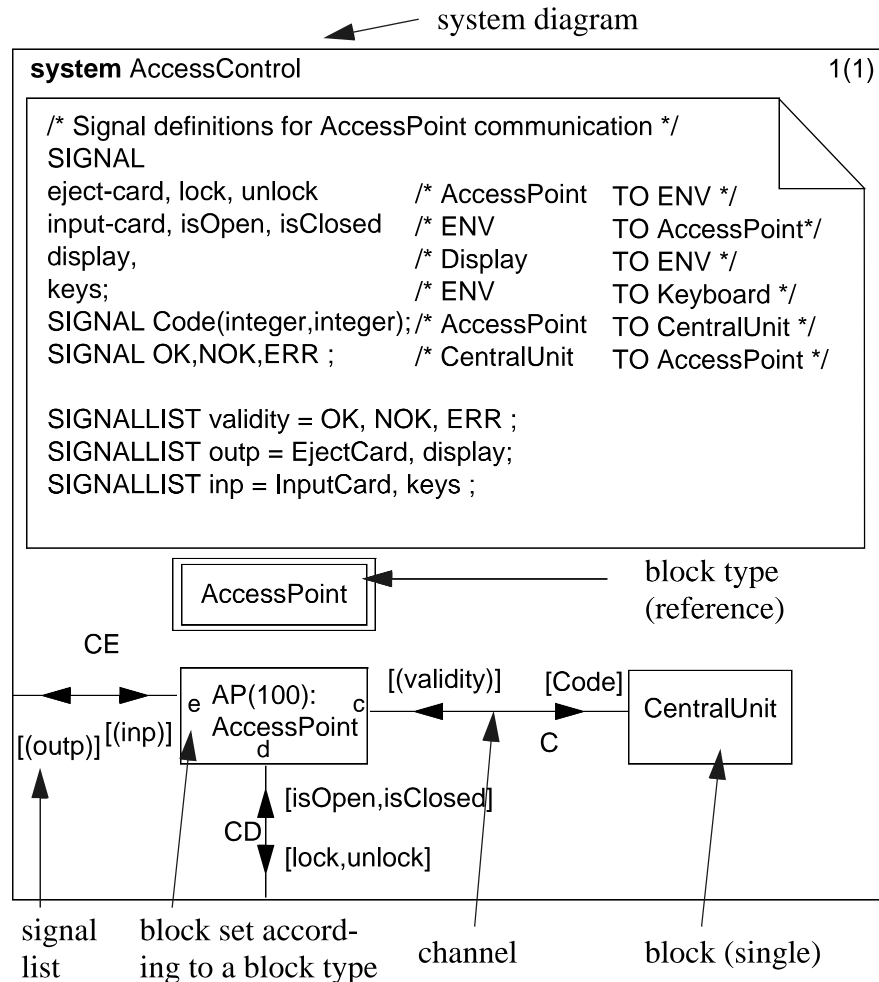
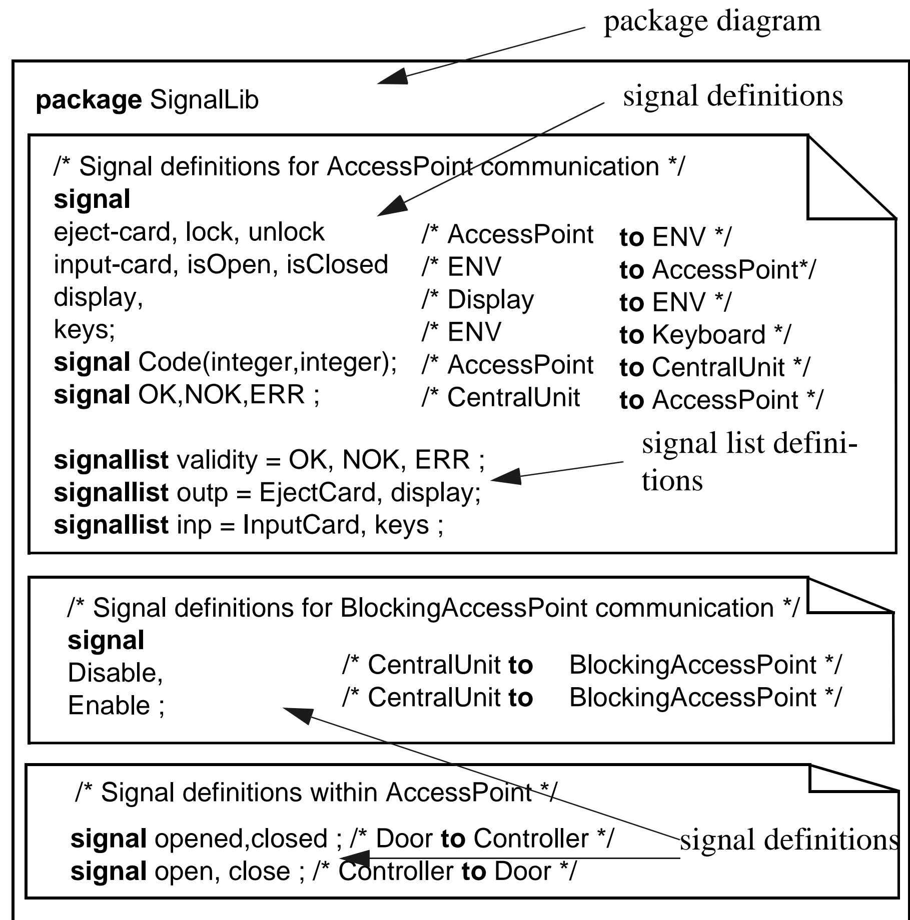


Figure 13-5: Package diagram SignalLib

TIMe Electronic Textbook v 4.0 © SINTEF Modified: 1999-07-16

Open figure



Packages: collections of related types and definitions

Sets of related types may come as a result of a domain analysis (the types representing application specific concepts) or as a result of a specific system specification where the types are needed.

A package is a set of types. Types that are only used in one system will normally be defined as part of the system specification, but for convenience they may be collected and defined in a package and then used by the system. If a set of related types are to be used in many systems within a specific application domain, then a package is the right place to define the types.

In Figure 13-5 (p.13-10) the signal types for the access control domain has been collected in the package SignalLib. Signals can be defined with parameters, as e.g. Code with two Integer parameters. This means that each Code signal carries two values of type Integer. The package also defines the signal list validity, inp and outp.

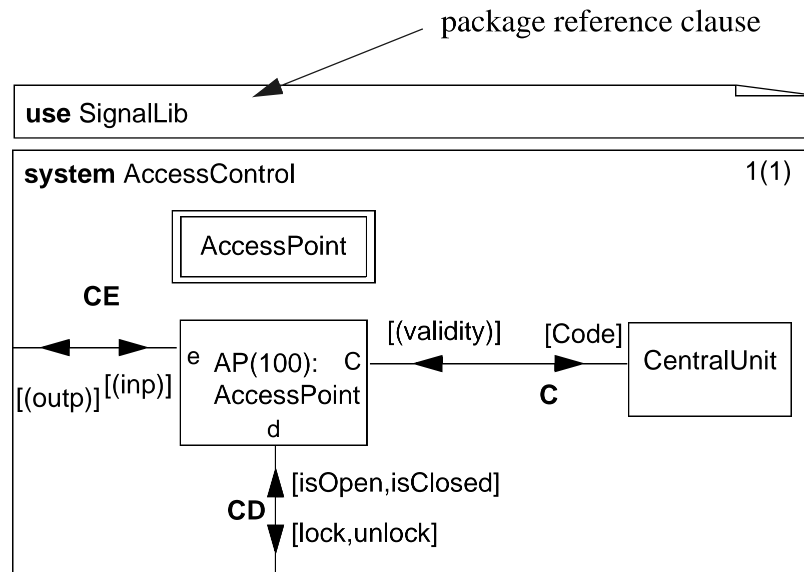
In Figure 13-6 (p.13-12) the signal definition of SignalLib are made available by a use clause as part of a system diagram.

In a similar way, the block type AccessPoint in Figure 13-6 (p.13-12) may defined in a package, possibly together with other types, instead of being defined in the system diagram.

For more details on constituents of packages see Package diagram, SignalLib (p.-48).

Figure 13-6: System using a package of type definition

Open figure



Subtypes

In general a type in SDL can be defined as a *subtype* of another type (the *supertype*) and thereby inherit the properties specified for the supertype. This holds for system, block process and service types, and for signals and procedures.

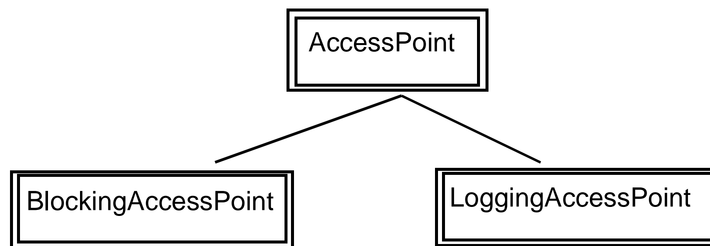
A general type intended to act as a supertype will often have some properties that should be defined differently in different subtypes, while other properties should remain the same for all subtypes. This is supported by virtual types and virtual transitions: these types and transitions can be redefined in subtypes. The behaviour of an instance of a subtype will follow the pattern given by redefinitions.

A subtype

- *inherits* all definitions of the supertype and can *add* own definitions; these include definitions of variables, procedures, signals and type definitions;
- *redefine* virtual types and procedures defined as virtual types and procedures in the supertype;
- *inherits* states and transitions (for those types where this applies) and may *redefine* virtual transitions;

TIMe Electronic Textbook v 4.0 © SINTEF Modified: 1999-07-16

As an example on this, we define two new block types (BlockingAccessPoint and LoggingAccessPoint) as subtypes of the block type AccessPoint, as illustrated below.



They will inherit all properties of AccessPoint, but it is essential that they can redefine the Controller part:

- BlockingAccessPoint so that it can be blocked even for people with validated card and PIN codes, and
- LoggingAccessPoint so that these access points will log what is going on at the point.

Therefore the Controller process type in AccessPoint is specified as a *virtual* (process) type in Figure 13-7 (p.13-13) and defined in the process type diagram in Figure 13-22 (p.13-51). Some of the input transitions are also defined to be virtual so that they can be redefined in redefinitions of the virtual process type.

Figure 13-7: Block type AccessPoint with virtual Controller process type

Open figure

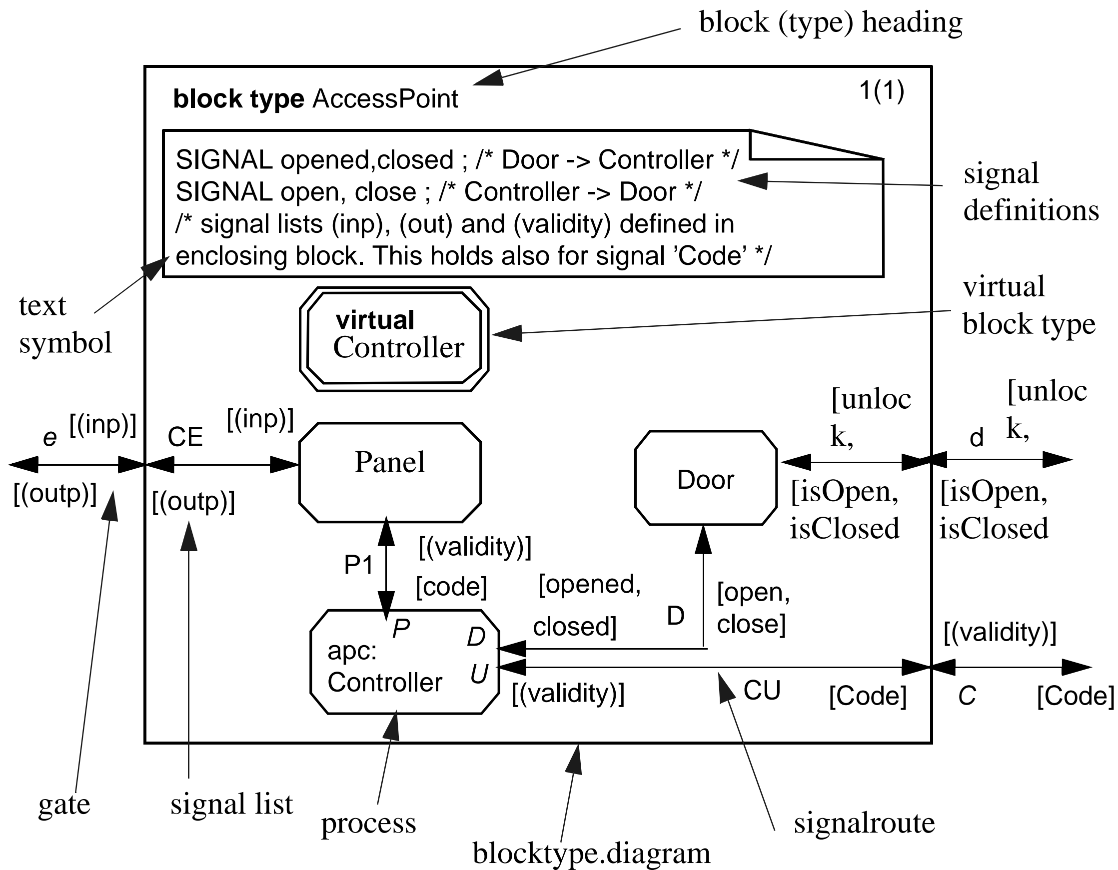
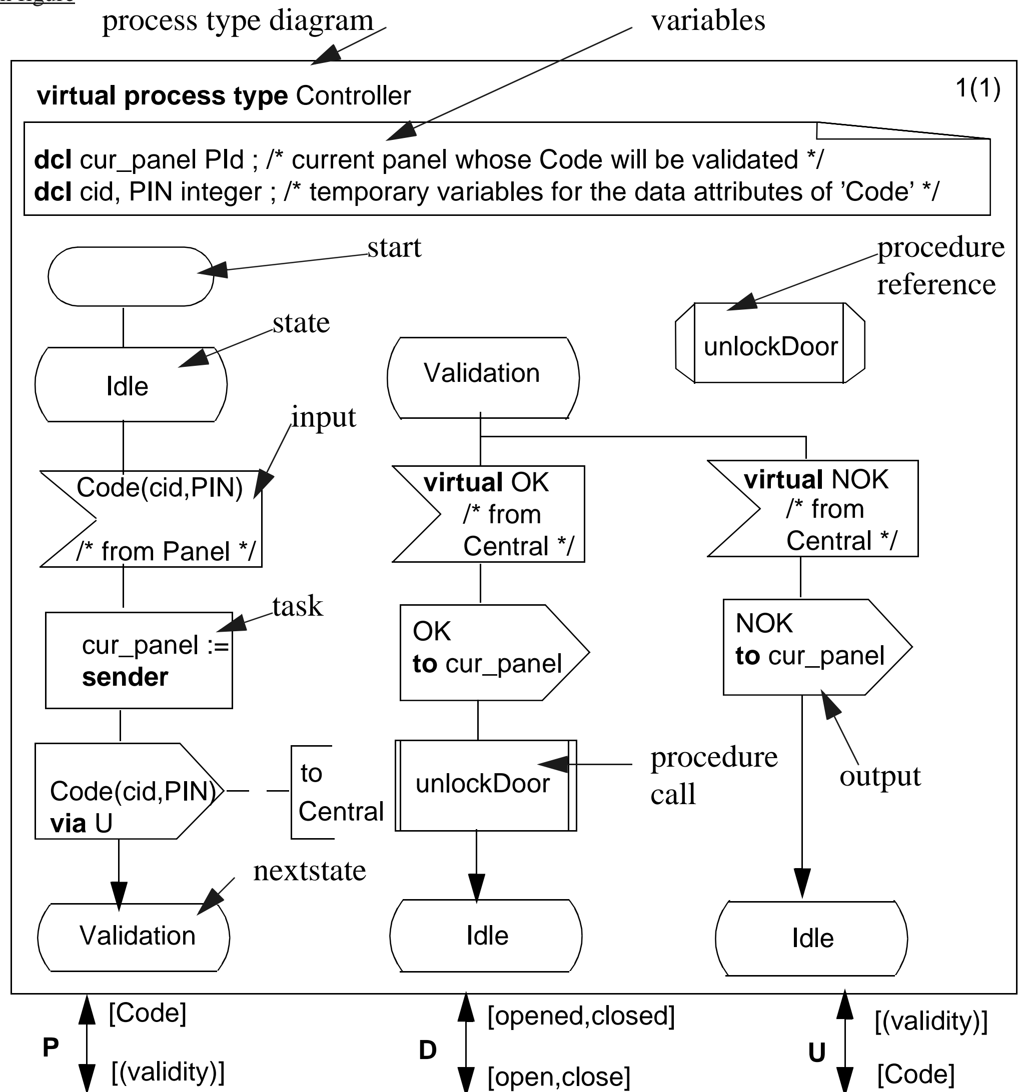


Figure 13-8: Virtual process type Controller

Open figure



For more details on constituents of block types see Block type diagram, AccessPoint (p.-51).

For BlockingAccessPoint the virtual process type is *redefined* as indicated in Figure 13-9 (p.13-15) and for LoggingAccessPoint as indicated in Figure 13-21 (p.13-49).

Figure 13-9: Block type BlockingAccessPoint as a subtype of AccessPoint

Open figure

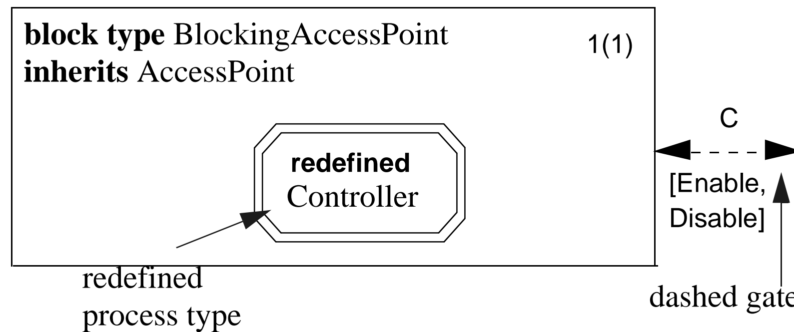
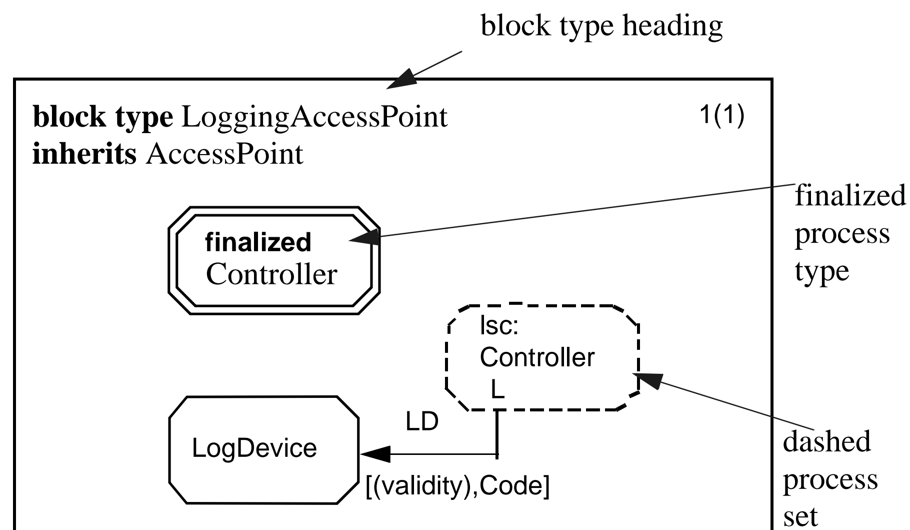


Figure 13-10: LoggingAccessPoint as a subtype of AccessPoint

Open figure



For more details on constituents of process type diagrams see Process type diagram, Controller (p.-59).

The redefined virtual process type Controller in Figure 13-9 (p.13-15) is given in Figure 13-11 (p.13-16) . This also illustrates inheritance for process types and thereby inheritance of behaviour: the redefined process type inherits the properties specified in the virtual process type (Figure 13-22 (p.13-51)) and the redefined process type adds the state “blocked” (with the corresponding input transition) and the input of “Disable” is added to all states (the * in the state symbol means all states).

The redefinition of the virtual process type Controller in Figure 13-21 (p.13-49) is given in Figure 13-12 (p.13-17). The redefinition is finalised, so that it can not be further redefined. The two virtual transitions are also given finalised redefinitions.

Figure 13-11: Redefined process type with added states and transitions

TIMe Electronic Textbook v 4.0 © SINTEF Modified: 1999-07-16

Open figure

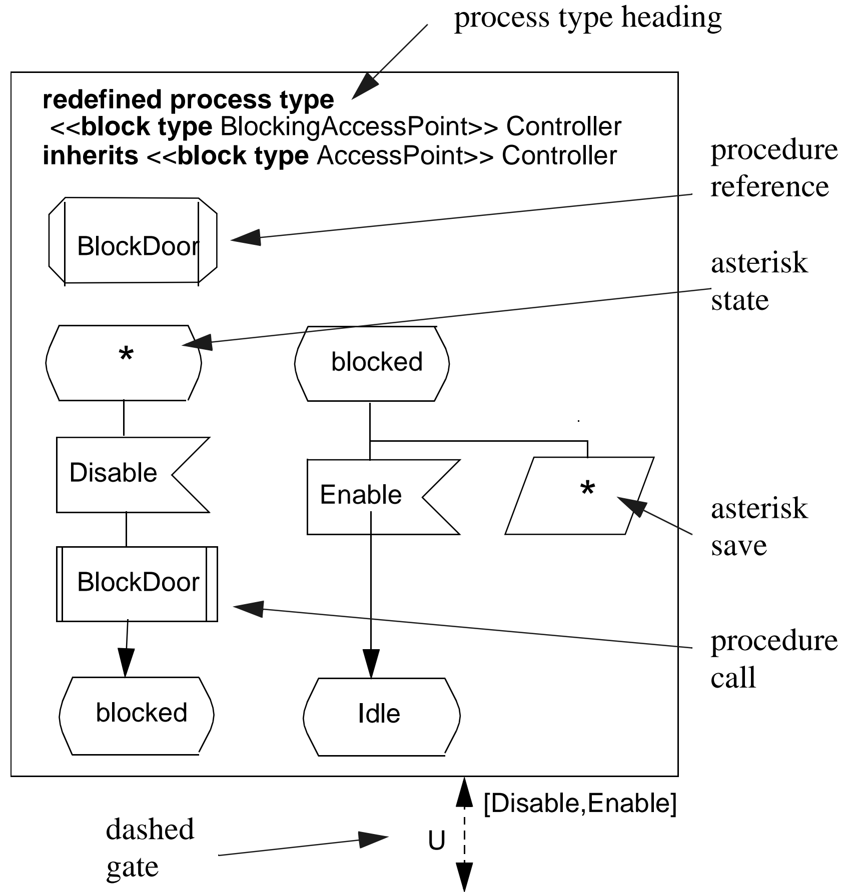
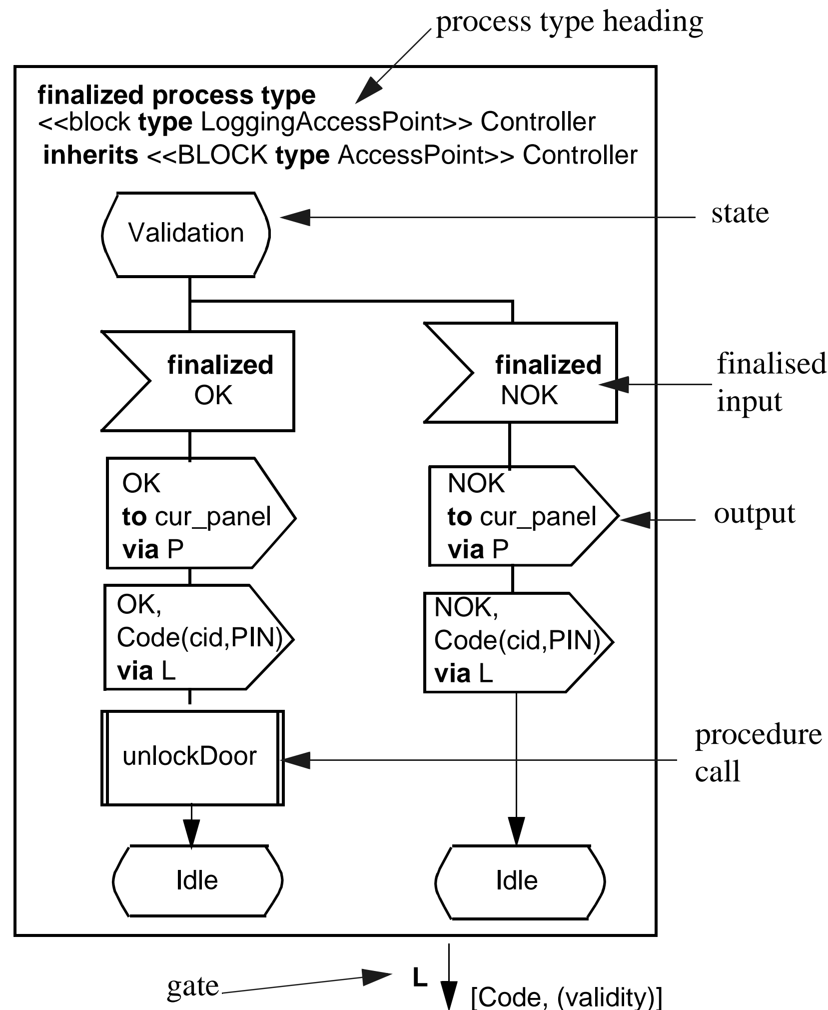


Figure 13-12: Finalised process type

Open figure



Composing behaviour of processes by means of services

If it is known that a component of the system has separate activities, each with their behaviour and possibly with local variables like processes, and if they should *not* execute concurrently, then the component may in SDL by represented by a process that consists of *services* representing the activities.

As an example consider the Panel process, with CardReader, Keyboard, Display, and PanelControl as separate activities but not executed concurrently. This is specified in Figure 13-13 (p.13-19).

In addition to services, the combined process may have variables. Services in one process instance do not execute concurrently with each other; only one executes at a time. The next service to execute is determined by the incoming signal or by signals sent from one service to another. Services share the input queue and the variables of the enclosing process.

TIMe Electronic Textbook v 4.0 © SINTEF Modified: 1999-07-16

Note that the connection points CE and P1 on the frame in Figure 13-13 (p.13-19) are not gates (the service diagram does not define a type of processes), but simply the names of the signal routes that connect Panel with the environment (CE) and with the Controller (P1), see Figure 13-2 (p.13-7).

The PanelControl service referenced in Figure 13-13 (p.13-19) is defined by the service diagram in Figure 13-14 (p.13-19).

Figure 13-13: Process in terms of services

Open figure

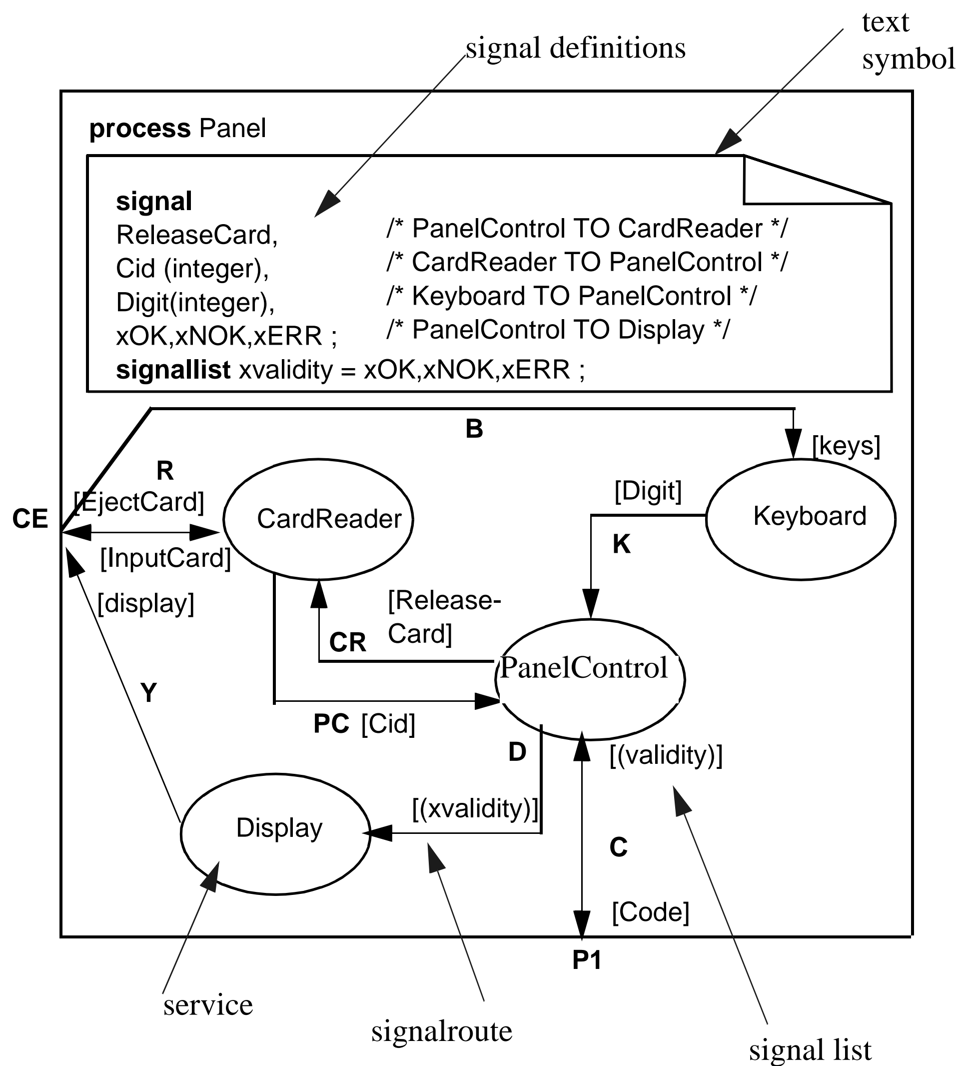
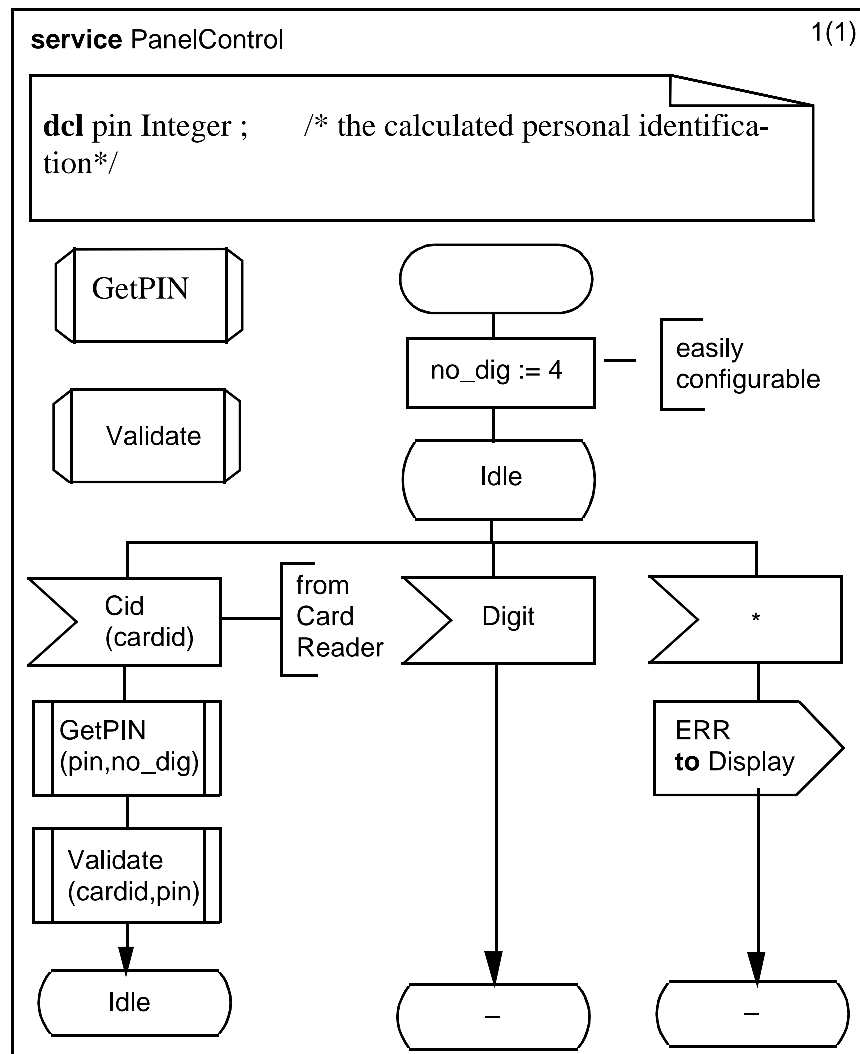


Figure 13-14: Service diagram, PanelControl

Open figure



Specifying properties of variables: data types

As mentioned already processes and services have attributes in terms of variables, and procedures may have local variables. Variables are declared according to data types. Signals may carry values of data types.

SDL supports data types by means of *abstract data type*. A data type defines possible values, behaviour and operators that can be applied to values of the type. The properties of data types are defined by means of axioms. SDL supports predefined types including Character, Boolean, Integer, Natural, PId (Process Instance Identifier), and Real.

It is possible to define data types. This is done by defining the literals and the operators of the type. Abstract data types defines these by means of axioms. Operators may instead be defined constructively, that is more or less as a procedures.

TIMe Electronic Textbook v 4.0 © SINTEF Modified: 1999-07-16

As an example of a simple user-defined data type, consider the Code signal that contains a card identification and a personal identification number. Suppose that we find it desirable to collect those two pieces of data in one structured type. This is done by the STRUCT concept:

```
NEWTYPE AccessCode
STRUCT
cardid, pin Integer ;
```

If AC is a variable of sort AccessCode then we may have the following assignments:

```
AC!cardid := 1234 ;
temp_pin := AC!pin ;
```

Templates for defining arrays, strings and powersets are also provided.

SDL as an object oriented language

This part of the tutorial provides you with a presentation of SDL as seen from an object orientation perspective. The idea is that you have some kind of understanding of what object orientation is and want to know what the correspondence is in SDL.

The entry to the material is made by means of terms that are more or less well-established as terms within object orientation. Popup menus will lead you to the corresponding elements in SDL. In case there are several alternatives, you either know what you are looking for, or you consult the “in general” entry, which will give you our understanding of the object orientation term and then the mapping to SDL.

- object
- attributes (instance variables)
- methods
- behaviour
- object interaction
- interfaces
- class
- subclass/inheritance
- aggregation/part-whole
- localisation of definitions
- class libraries
- parameterised classes

The different concepts are illustrated by SDL examples. When reading this interactively, it is recommended, when looking up the example for the first time, to open it in a *separate* window (shift click) - subsequent references to examples will then simply show the examples in this second window. You will have two windows: one with this text and one with the example.

This part of the tutorial may also be used in cases where you have made an object oriented analysis in some other language or notation and now want to make the more formal specification or design in SDL.

If you want some background information on the object oriented approach behind SDL, look at Object oriented approach behind SDL (p.13-34).

Objects

As part of object modeling, components of the real world systems (or anticipated such) are identified and selected properties are described. These components are modeled by objects. Objects may be classified into categories of objects (modelled by classes) and into subcategories (modelled by subclasses).

TIMe Electronic Textbook v 4.0 © SINTEF Modified: 1999-07-16

The corresponding is done in SDL, just in other terms: SDL systems consist of instances. The classification of components into categories and subcategories is in the SDL system represented by types and subtypes of instances (see Subclass/inheritance (p.13-29)).

If you are interested in the approach to object orientation behind SDL, have a look at Object oriented approach behind SDL (p.13-34).

An SDL system consists of a set of instances. Instances may be of different kinds, and their properties may either be directly defined or they may be defined by means of a type. If it is important to express that a system has only one instance with a given set of properties, then the instance is specified directly, without introducing any type in addition. If the system has several instances with the same set of properties, then a type is defined, and instances are created according to this type.

If you have an object that should act concurrently with other objects, then you are looking for a Process (p.13-23) in SDL, and if you have classes of these you should define a Process type (p.13-29).

If you have objects that each contain several concurrent activities, then represent these activities by processes, and each container object by a Block (p.13-23). If you have classes of these define a Block type (p.13-29).

Process

Example

Processes are objects that perform concurrently with other processes and communicate by sending signals or by remote procedure calls. Processes are the main objects of SDL. Classes correspond to process types.

A process instance is part of a process set, which in turn is part of a block. The fact that a process set is part of block is described by a process reference. Properties of processes are either described directly by a process diagram, defining a set a processes, or by means of process type diagram, defining a type of processes, see Process type (p.13-29).

Service

Example

A service object is an instance that is an integral part of a process. Services alternate executing, depending on incoming signals to the container process.

The properties of a service are either described directly in a service diagram, describing on single service as part of a process, or they are defined by a service type diagram, defining a Service type (p.13-29).

Block

Example

A block is an object that contains other blocks or processes; in addition it may contain local definitions of types, e.g. signal types, data types, procedures, process types and block types. A block does not have variables, so it is not possible to represent state-carrying objects by means of blocks.

A block is either described directly in a block diagram or a type of blocks is described by a block type diagram, see Block type (p.13-29).

System

Example

A system object is the outermost object containing the blocks of the system connected by channels. In addition the system may have definitions of types, but it may not have instance variables - global, shared variables have to be global variables of shared processes.

A system is either described directly in a system diagram or a type of systems is described by a system type diagram, System type (p.13-29).

Variable

A variable is the association of a name and a value of some data type. Variables are only part of processes and services, and as local variables of procedure instances, but they are not part of blocks and systems.

Data types defines operations that may be performed on values (objects) of these types.

Attributes

In addition to methods, objects have data item attributes. These are often just variables of some predefined types (having values, that represent part of the state of the object) and object references, that is variables that denote other objects. Some languages also provides variables of user-defined classes.

Most object oriented languages follow the approach that instance variables should not be accessed directly from other objects, but only via methods. Variables used in this way thereby become part of the implementation of the class, while the interface of the object is represented by the methods.

Methods may also have local variables that are used as auxiliary variables.

In SDL process and service instances may have variables, while systems and blocks can not have variables. It is possible to access variables of processes from other processes, but it is recommended to access them either through Exchanging signals (p.13-28) or through Calling remote procedures (p.13-28). Shared variables must therefore be variables of some shared processes.

Procedures may also define local variables.

Example

Variables are declared in text symbols.

Types may be user-defined, see Specifying properties of variables: data types (p.-24) for a short introduction.

Predefined types

In addition SDL has predefined types which are quite similar to those we are used to from programming languages. The common predefined types are: Boolean, Character, Charstring, Integer, Natural and Real.

PId

Object references are in SDL supported by variables of the predefined type PId. PId variables may denote processes of any process type, so object references in SDL are not typed.

TIMe Electronic Textbook v 4.0 © SINTEF Modified: 1999-07-16

Methods

Methods of objects are properties that define possible behaviour patterns that the object may apply to the data item attributes and thereby change the state information of these.

The main kind of SDL entity corresponding to an object is a process. A process may have procedures and functions (that is value returning procedures) defined as part of its definition. Such procedures may be executed by the process itself, and if exported they may be requested by other processes (see Calling remote procedures (p.13-28)).

Procedures may also be defined in block and in systems (see Globally defined procedures (p.13-26)) ; they may then be executed by those processes that are defined within these blocks/systems. Procedures may also be defined in a service; they may only be executed by the service itself.

Procedures

Example

Procedures defined locally to a process or service or globally in a block, system or package are intended for decomposition of the behaviour specification into partial action sequences. In order for a procedure to represent a property of a process, so that other processes may request its execution, the procedure must be exported by the process, see Remote procedures (p.13-26).

Procedures work much the same way procedures in programming languages. They have value parameters (in, out, in/out) and a procedure defined locally to a process or service may manipulate variables of the enclosing process/service. and thereby have side-effects.

Procedures defined in types and inherited in subtypes can only be redefined if they are defined as virtual procedures (see Virtual procedures/functions (p.13-25)), while ordinary procedures are guarantied to have the same effect for all subtypes.

A procedure is a type in itself, and as such can be based on another more general procedure by specialization. This holds for procedures in general, that is for both ordinary procedures, exported, virtual, and value returning procedures. See Inheritance of behaviour (p.13-31) for the details on how this works.

Functions - i.e. value returning procedures

Value returning procedures are procedures that can be called as integral parts of expressions. Value returning procedures can be used very much like an operator, but they may contain states just like an ordinary procedure (while operator diagrams cannot).

Virtual procedures/functions

Most object-oriented languages have virtuals in terms of virtual procedures. BETA has in addition virtual classes. Some languages, e.g Smalltalk, do not distinguish between virtual and non-virtual procedures (all methods are virtual and may be redefined in subclasses), while e.g. BETA, C++, Eiffel, and SIMULA distinguish. SDL dis.fmtinguishes between virtual and non-virtual procedures (and types in general - see Virtual classes/types (p.13-33)). The rationale for the distinction is that the designer of a general

(super)class may want to ensure (in order for it to work) that some of its procedure attributes should not be redefined in subclasses. Distinguishing between virtuals and non-virtuals is the most general approach, as a special case of this is simply to specify all to be virtual.

A virtual procedure is defined by a procedure diagram, where the procedure heading starts with the keyword **virtual** and optionally has a virtuality constraint specified. A virtuality constraint for a procedure is the name of another procedure. The procedure diagram is otherwise as an ordinary procedure diagram.

A virtual procedure is redefined in a subtype of the type containing the virtual procedure by a procedure diagram with a procedure heading starting with the keyword **redefined** or **finalized** and with the same name as the virtual procedure. If the virtual procedure has a virtuality constraint, then then the redefinition must be a procedure that is specialisation of the constraint procedure.

Globally defined procedures

Procedures may be defined in packages, systems and blocks, even though they have to be executed by processes, services or other procedures.

Remote procedures

Remote procedures are supported by server processes *exporting a procedures,* client processes *importing procedures* (their signatures) and *calling remote procedures*.. In addition the procedures are specified in a context enclosing both client and server.

The exporting process can control in which states it will accept the remote request. It may also specify to save the request to other states.

The calling of the remote procedure is indistinguishable from local procedure calls unless the caller explicitly states the client process.

Remote procedures may be value returning (as in our example above) and they may be virtual.

For details on remote procedures, see remote procedures.

Behavior

Most object oriented languages assume that behaviour is only associated with methods of objects and that a method of an object may be executed whenever some client object needs the effect of the method. In cases where this is not really the case, the methods are described in order to take this into account. Special methods are executed as part of the construction/deletion of objects, but apart from this the object itself has no specified behaviour.

A few languages support objects with individual behaviour. This is especially languages that also supports concurrency and where e.g. the synchronisation between objects are described as part of the behaviour of the objects and not of the methods.

TIMe Electronic Textbook v 4.0 © SINTEF Modified: 1999-07-16

Some object oriented analysis methods recommend that the behaviour of important objects are described by state/transitions diagrams, where important states are identified and events that cause transitions between states and corresponding method execution are described.

SDL (by objects of kind process and service) belongs to the class of languages where objects have individual behaviour.

Process behaviour by Finite State Machine

Example

Process behaviour in SDL is defined by means of Extended Finite State Machines (EFSM). As such it fits with analysis methods that recommend the state/transition diagrams for important objects.

Process behaviour is described in the so-called process graph, with states and transitions. In a given state a process may input a number of signals, and the consumption of a signal leads to the execution of the following transition and entering the next state.

Process behaviour by service composition

Example

Sometimes it can be useful to describe the behaviour of a process as a number of partial behaviours. Instead of specifying the complete behaviour of a process type, it is possible to define partial behaviours by means of service types. A process type can then be defined as a composition of service instances according to these service types. In addition to services, the combined process may have variables. Services in one process instance do not execute concurrently with each other; only one executes at a time. The next service to execute is determined by the incoming signal or by signals sent from one service to another. Services share the input queue and the variables of the enclosing process.

Object interaction

Most object oriented languages support only one thread of action and have method call and direct instance variable access as the only kinds of object interaction. Some languages provide mechanisms for concurrent objects (with several threads) and corresponding mechanisms for either non-synchronised message passing or synchronised (remote) procedure call.

SDL belongs to the second class of language. Processes execute concurrently. The behaviour of each process is represented by a Finite State Machine (see Behavior (p.13-26)). Processes interact either by Exchanging signals (p.13-28) or by Calling remote procedures (p.13-28).

SDL models independent behaviours as (the behaviour of) concurrent processes.

The essential information one wants to convey in SDL models, is not the independence, however, but the dependency between systems. It is mutual dependencies that give systems purpose and meaning. Hence, a precise and unambiguous definition of mutual dependency is the prime concern. For this reason, all dependencies are modelled explicitly as signals interchange between the processes and their environments. There is

basically no way a process and its environment may influence each other apart from sending signals through the signalroutes/channels that link the process and its environment together.

Exchanging signals

Processes in the system and the environment communicate with each other by sending signals through the signalroutes and channels. There are no shared data to be found outside the processes, so signals are the only means for processes to communicate. There is no way for one process to directly manipulate another process.

There is no priority among signals; signals arriving at a process will be merged into one single queue in the order in which they arrive. There is one and only one signal input queue associated with each process. This queue is called the .i.input port;. 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.

Calling remote procedures

In addition to the sending of signals, processes can interact by means of remote procedure calls. Such procedures must be defined as remote procedures: the server processes must export the procedures, the client must import them. In addition the procedures are specified in a context enclosing both client and server. This makes the signatures of the procedures known to both server and client.

The calling of the remote procedure is indistinguishable from local procedure calls unless the caller explicitly states the client process.

For details on remote procedures, see remote procedures.

Gates

Object interaction is often based on objects having an interface or a set of interface in terms of signatures of methods.

Example

The corresponding mechanism in SDL is the gate. In order for processes in process sets to exchange signals, the process sets must be connected by signal routes, and the enclosing blocks and block sets must be connected by channels. Services as part of processes are also connected by signal routes.

When defining types of blocks, processes and services, the possible connection points for channels/signal routes are defined as gates. A gate can be specified to be one-way or two-way gates, and for each of the directions it can be specified which signals may be accepted/sent.

Class

The classification of system components into categories of components is in object orientation modelled by classes. The corresponding notion in SDL is a type (of instances). A type defines the common properties of a category of instances. Each instance has its

TIMe Electronic Textbook v 4.0 © SINTEF Modified: 1999-07-16

own identity and its own set of properties, e.g variables with different values. A type is not a set of instances: SDL has separate constructs for defining sets of instances (see process sets, block sets).

Most object oriented languages (and also analysis methods) provide only one kind of object, and correspondingly only one kind of class. This is not the case n SDL. Look at Objects (p.13-22) to get a description of which kinds of objects that are supported by SDL. Depending upon the kind of the category of component you are supposed to model by a corresponding kind of SDL type of instance.

Process type

Example

A class of concurrent, message passing objects is in SDL represented by a process type.

Service type

A class of alternating components within a concurrent object (process) is in SDL represented by a service type.

Block type

Example

A class of container objects is in SDL represented by a block type.

System type

Example

A class of whole application-objects is in SDL represented by a system type.

Abstract Data Type

Classes or types defining the properties of attributes/instance variables are in SDL represented by Abstract Data Types. This mechanism allows you to define types by means of values (literals), operation signature and behaviour by means of axioms. For a short introduction see Specifying properties of variables: data types (p.-24).

Subclass/inheritance

Classes allow to model concepts from the application domain and to represent the classification of similar objects. Specialisation of general concepts into new more specialised concepts is in most object oriented languages represented by subclasses. Subclasses are said to inherit the properties specified in the superclass.

The language mechanisms for this in SDL are specialisation of types by means of inheritance, virtual types and virtual transitions.

A (sub)type may be defined as a specialisation of another (super)type. A subtype inherits all the properties defined in the supertype definition, it may *add* properties and it may *redefine* virtual types and virtual transitions. Added properties must not define entities with the same name as defined in the supertype (within the same entity class).

A parameterised type can also be specialised. All properties of the super type, including formal parameters and context parameters,are inherited. A subtype definition may add formal parameters, context parameters, and other properties.

Only types and parameterised types can be used as supertypes, including procedures. It is *not* possible to inherit from a single block definition, from a process set definition, or from a service definition.

Inheritance

Below follows a list of possible ways of inheritance and redefinition of properties defined in the superclass.

- Inheritance of attributes

  Most language support simple inheritance of attributes (data items, part objects). Some languages allow additional attributes defined in a subclass to override attributes defined in the superclass (redeclaration of attributes), while other languages do not allow this.

  SDL doe.fms not allow redeclaration of attributes. Only type attributes defined as virtuals can be redefined.

  For the details on this in SDL look at Adding properties (p.13-31).
- Redefinition of methods

  Some languages (like Smalltalk) allow that all methods may be redefined in subclasses, while other languages (like C++ and Eiffel) require that these methods shall be defined as virtual entities. The rationale behind this is that the specifier of a superclass may want to assure that some crucial methods are not redefined, because other methods may depend on them.

  SDL requires that types, procedures, and transitions shall be defined as virtual entities in order to be redefinable in subtypes.
- Inheritance of actions

  Most languages do not support inheritance of actions. The prime reason for this is that most languages support only objects with attributes, so that all actions are associated with procedure/method attributes. And inheritance for procedures/methods are usually not supported.

  Some concurrent languages support objects with actions (that is process objects where each object has its associated sequence of actions that is executed concurrently with the action sequence of other objects). They support inheritance of attributes, but not of actions.

  Specialisation of actions may be done in two different ways:
- specializing the effect of an action, or

TIMe Electronic Textbook v 4.0 © SINTEF Modified: 1999-07-16

- specializing the ordering of partial action sequences comprising an action.

SDL provides the second alternative, see Inheritance of behaviour (p.13-31).

Adding properties

A subtype may define properties that come in addition to those inherited from the supertype. Such added properties are not allowed to have the same names as an inherited properties of the same kind.

Example

When adding block sets, process sets or services, these must be connected to block sets, process sets or services that are inherited. In order to distinguish between these in the graphical representation, the inherited elements are dashed. The same holds for dashed gates: it is not only possible to connect new elements to them, but is it is also possible to add to the constraint of the inherited gate.

Redefining virtuals

Example

A virtual type (that is block-, process-, service- type or a procedure) in an enclosing type can be redefined in a subtype of the enclosing type. As part of the virtual type definition, a virtuality constraint can be defined: any redefinition must then be a subtype of this constraint. This allows for analysis of type with virtual types, even though the virtual types can be redefined in subtypes.

Inheritance of behaviour

Example

A subtype of kind process, service or procedure inherits all the transitions of the supertype, except the virtual transitions that are redefined.

Class libraries

As part of analysis and specification, sets of application specific concepts will often be identified, and the corresponding classes defined. A common strategy is to collect related classes in class libraries.

Example

The corresponding element in SDL is a package of type definitions. Types that are only used in one system will normally be defined as part of the system specification. If a set of related types are to be used in many systems within a specific application area, then this set can be represented by a package.

Note that a package is simply a collection of type definitions and as such not existing when the system is operating. It is a means for organising descriptions and not for structuring systems.

Aggregation/part-whole/containment

Some object oriented languages support the notion of objects being contained in other objects. Words being used are also aggregation, part-objects.

Two major approaches to this are:

1. contained objects are constituent parts of the containing object, that is they are created as part of the containing object. A given object can only be part of one object,
2. containment is rather a special relation between two separate objects and a given object can be part of more than one object. This is e.g. the approach followed in OMT.

SDL has support for (a slight modification of) the first approach, and relations in general are not supported:

Example

- Blocks are constituent parts of a system or a block. Block are created as part of the creation of the system and con not be created dynamically

Example

- Processes can only be part of one process set that is part of a block - processes in the set can, however, be created dynamically,

Example

- Services are constituent parts of processes.

Localisation of definitions

Some phenomena and concepts are only meaningful within the context of a specific phenomenon or concept. Localisation of definitions supports this and gives rise to nesting of definitions.

Most object oriented languages only provide flat name spaces, with one large set of class definitions, with locally defined methods. Only few languages provides class definitions within class definitions, and few languages provides any other mechanism for enclosing a set of related object and class definitions.

SDL def.fminitions may be nested and thereby support localisation. Type definitions may, be located where it is most convenient, as long as they are visible from they are supposed to be used. If identified types in an early stage should be specified as part of the system specification (and not yet as part of a package), they may simply be defined at the system level, without considering where they in fact belong. General types of e.g processes and procedures can be defined at system level, in order to be used in several blocks of the system. More special types should be defined where they are used.

A package of types is the ultimate example on non-localised type definitions, while exported procedures will most often be defined locally to the process (type) that exports it. Signals are often defined in the nearest enclosing block in which they are used between processes. Context parameters (see Parameterised classes (p.13-33))provide the mechanism to make a type definition independent on the enclosing scope.

System structure in terms of instances implies relations between instances, while localisation implies a relation (is-local-to) between definitions. Localisation is in SDL supported by nesting of definitions, and it forms the basis for scope-rules and visibility rules.

An SDL specification consists of definitions of entities of the different entity kinds (for a list of entity kinds, see entity kinds).

Some definitions may contain definitions of other entities (nesting) and will therefore form the scope units for these entities. (for a list of scope units, see scope units).

TIMe Electronic Textbook v 4.0 © SINTEF Modified: 1999-07-16

As part of the definition of an entity, the name of the entity is defined. Entities defined in the same scope unit and belonging to the same entity kind must have different names, while entities of different kinds may have the same name. As an example a procedure and signal defined in the same scope unit may have the same name. While it is sometimes convenient to be able to reuse a name in this way, it should not be done too much--readers may otherwise easily be confused.

Entities defined in a scope unit are visible in this scope unit and in all nested scope units. A signal defined in a block is e.g. visible in the block definition itself (where it can be used in the specification of channels), and it is visible in an enclosed process type definition (where it can be used in outputs).

When several definitions in nested scope units have the same name, the name will refer to the definition in the innermost scope unit (starting with the one containing the use of the name). In order to refer to one of the other definitions with the same name, a qualified identifier must be used.

Parameterised classes

This is not covered in this version, since it is not supported by tools.

One way out

Some object oriented languages support classes/types as parameters to classes. The class/type parameter may be used almost as an ordinary class/type. Depending upon the language it is possible to perform independent analysis of such a parameterised class.

The notion of context parameters provide one kind of parameterisation of types in SDL, but it does not cover block- and process types as type parameters to block types. This, is however, covered by a special application of virtual block- and process types.

A block type BT that has to have a process type PT as parameter is defined so that PT is a virtual process type in BT, with a constraint C that matches the use of PT in BT. When an actual process type APT should be provided to BT, this is done by defining a subtype SBT of BT and redefining the virtual process type PT to a subtype of APT. If just the parameter binding is desired, then this subtype adds nothing to APT, but in general it is possible to add properties in the subtype of APT. The requirement on APT is that it is a subtype of C.

Virtual classes/types

Most languages only support virtual procedures, or methods that may be redefined in subclasses. If a class shall be parameterised by a type or class, then a separate notion of type parameters is introduced.

SDL provides the notion of virtual types in general, and not just virtual procedures.

The general rule is that if a type of a certain kind can have definitions of types of different kinds, then it can have definitions of virtual types of the same kinds. This means that you can define

- system types with virtual block-, process-, service types and virtual procedures,
- block types with virtual block-, process-, service types and virtual procedures,
- process types with virtual service types and virtual procedures,
- service types with virtual procedures, and
- procedures with virtual procedures.

A virtual type has in addition to the definition (e.g. in terms of behaviour in terms of a process graph) also a virtuality constraint. This will be a type of the same kind as the virtual type, and the simple rule is that *any redefinition shall be a subtype of the constraint.*

The constraint of a virtual type is used in the analysis of the use of the virtual type in the enclosing definition, and the idea is that a virtual type can only be used according to its constraint. In general a constraint type is a general type and can as such contain whatever a type may contain, but the normal cases are

- for virtual block types: gates, so that block sets of the virtual block type can be correctly connected;
- for virtual process types: gates, so that process sets of the virtual process type can be correctly connected, and formal parameters, so that instances can be created with the correct set of actual parameters (in fact a virtual process type redefinition cannot add formal parameters for the same reason);
- for virtual service types: gates, so that instances can be connected correctly;
- for virtual procedures: formal parameters, so that the virtual procedure can be called correctly, and for the same reason a redefinition of a virtual procedure cannot add formal parameters.

For details on the the redefinition of virtual types, see Redefining virtuals (p.13-31).

Object oriented approach behind SDL

The benefits of object orientation range from the underlying philosophy of modelling the phenomena in the form of objects, to the compactness of descriptions achieved by the use of the inheritance and specialisation mechanisms. Hence, there are two separate ideas that go under the name of object orientation and both are part of the object orientation presented here:

1. The notion of objects. It conceives each object as being characterised by data items carrying state information, by local patterns of action sequences (procedures, methods) that the object may apply to these data items and by an individual sequence of actions that the object may execute on its own.
2. The objects are active objects and not just passive data structures with associated operations. In order to directly model the different kinds of action sequencing found in a large class of application areas, the approach includes the **execution of objects as part of other objects** (as is the case for procedures and methods), as **alternating**

TIMe Electronic Textbook v 4.0 © SINTEF Modified: 1999-07-16

(one at a time) with other objects and as **concurrent with other objects**. An essential property is that objects have a well-defined interface that hides the internal structure of data items and action sequences from the environment.

3. The notion of hierarchical types. The approach makes a sharp distinction between classes and objects. (Other words commonly used are types and instances.) Objects are carriers of state information and behaviour, while classes are patterns defining common structure and properties of objects. **A class is not regarded as a set of objects, but as a definition of a category of objects.** Classes do not contribute to the total state of a system, but help in organizing objects in type hierarchies. Objects model the phenomena of the application area, while classes model the types. The importance of this aspect is that it provides effective support to reuse.

Reuse of components requires language mechanisms to support composition and adaptation of reusable components. Object-oriented concepts give answers to both of these: composition by clean interfaces between classes of objects and adaptation by inheritance and specialisation. The notion of objects and type hierarchies also promotes the definition of general classes that may be reused in many different applications.

SDL by example

This part of the SDL tutorial leads you through SDL by means of an example. You will learn about the various elements of SDL by clicking on the desired elements in the diagrams. In case parts of a diagram reference other diagrams, e.g. process references, clicking the name (in underlined blue or red) will follow the reference and bring you to the referenced diagram.

If you want to be lead through the example top down, start with System diagram, Access Control System (p.13-38) and follow the diagram references from there.You may alternatively choose to look at the kind of diagram you want to learn about.

Introduction to the example (p.13-36) gives a short, informal introduction to the example being used throughout.

- Package diagram, SignalLib (p.13-40)
- Package diagram, AccessPointLib (p.13-42)
- Block type diagram, AccessPoint (p.13-44)
- Block type diagram, BlockingAccessPoint (p.13-47)
- Block type diagram, LoggingAccessPoint (p.13-49)
- Process type diagram, Controller (p.13-51)
- Process type diagram, redefined Controller in BlockingAccessPoint (p.13-57)
- Process type diagram, finalised Controller in LoggingAccessPoint (p.13-59)
- Process diagram, Panel in terms of services (p.13-60)
- Service diagram, PanelControl (p.13-62)
- Procedure diagram, GetPIN (p.13-63)

Introduction to the example

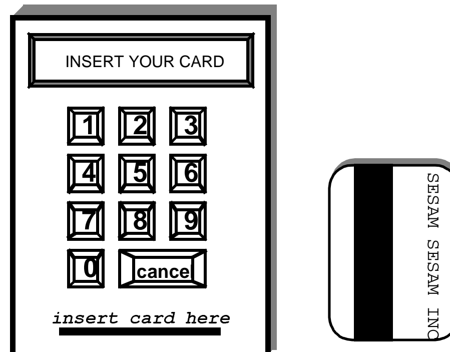
The purpose of access control systems is in general to control the access to some service to people with known identity, represented by cards and personal codes. In this specific example the system shall control access to access zones by controlling the opening of doors.

Each card holds a unique Card-code that identifies the card. To grant access the system will read the Card-code and then check the corresponding access right. For additional authentication, the user will be asked to enter the secret personal number (PIN).

Figure 13-15: Panel and card of an access control system

TIMe Electronic Textbook v 4.0 © SINTEF Modified: 1999-07-16

Open figure



The card is a plastic card with a magnetic strip holding a card code and possibly an encrypted PIN code. The physical appearance of the panel and the card is shown in Figure 13-15 "Panel and card of an access control system" (p.13-36). Each panel represents an Access Point.

The main service demanded by the user is to gain access when the card and code is presented to the system, and to deny access if an attempt is made to enter at an access point where the user is not authorised to pass.

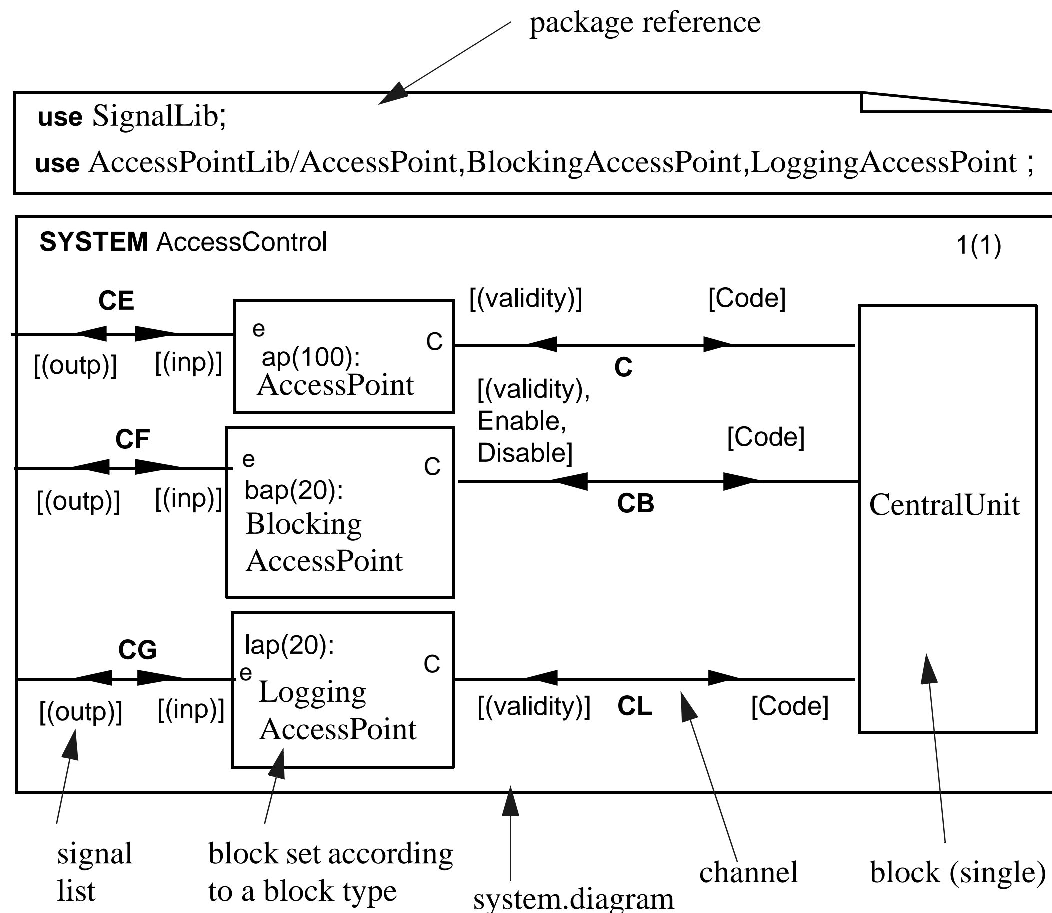
A typical access control system will consist of a number of access points and a central unit where validation is performed. Some access points are so-called blocking access points, that is access points that may be blocked by an operator, so that access is denied even with a valid card and code, until the access point is enabled again. Other access point may have the property that they log what is going on at the point.

In order to illustrate as many mechanism of SDL as possible, the example system will consist of three sets of access points, each of a different type. In a real access control system one may choose to give all access points the possibility of being blocked and of logging.

System diagram, Access Control System

Figure 13-16: System diagram for access control system with three types of access points

Open figure



In SDL a system is defined by means of a system diagram. By making a system diagram it has been decided what is part of the system and what is part of the environment of the system.We choose to design the access control system such that the access terminals (called AccessPoints) are *within* the system, while the users actually getting access are *outside* the system. The CentralUnit containing the access rights is within the system, while for our current purpose, how the access rights information got into the CentralUnit is not described.

Before drawing this border between the system and the environment and thereby deciding what should be part of the system, a domain analysis will normally have taken place, different solutions will have been considered and different sketches of the system will have been tried out.

In this presentation, the final system description is presented top-down, in order to present the various SDL language elements.

TIMe Electronic Textbook v 4.0 © SINTEF Modified: 1999-07-16

This Access Control system consists of one single block (CentralUnit) and three block sets, that is sets of blocks according to block types, connected by channels. It communicates with the environment that is supposed to behave like processes representing the users of the system, the operators and the controlled physical panels and doors at the access points.

System

A system is in general a set of blocks, block sets and channels. Blocks and block sets are connected with each other or with the environment of the system by means of channels. This means e.g that there may not be processes directly as part of the system and systems will not have global variables.

Environment

For the system the environment consists of a set of SDL processes that may send signals to the system and which may receive signals from the system. The signals for this purpose are defined in the system or, as here, in a package (Package diagram, SignalLib (p.13-40)) used by the system. The users of the system are thus regarded as processes in the environment.

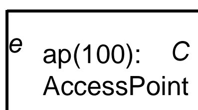
Block

A block is created as part of the creation of the enclosing block or system. All blocks are created as part of the system creation, that is there is no dynamic creation of blocks.

The CentralUnit block is specified directly (singular block), while the other blocks of the system are parts of block sets according to block types. The symbol with CentralUnit is also a *reference* to a block diagram that describes the properties of the block.

Note that the block reference is merely a graphical shorthand for diagrams. Block references may be substituted by block diagrams, but the surrounding diagrams would be very crowded and illegible if diagrams could not be remotely referenced by block references. The reference defines the scope of the name.

block set



Type-defined blocks are contained in block sets. A block set is a fixed number of blocks with properties according to a block type. The set of AccessPoints is called ap and the number (100) designates the cardinality of the set. A channel connected to a block set (via the gates e or C) will actually represent a set of channel instances.

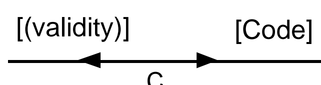
A block set is *not* a reference (as CentralUnit). It defines a set of block instances, but it relies on the definition of the block type AccessPoint. This block type definition is not part of the system, but part of the Package diagram, AccessPointLib (p.13-42) and defined in Block type diagram, AccessPoint (p.13-44).

channel

Blocks and block sets are connected with each other and with the environment by means of channels. A channel is a one-way or two-way directed connection. It is characterised by the signals that it may carry. A channel has a signal list for each direction.

If there is no channel between two blocks, then processes in these two blocks cannot communicate by signal exchange. Processes may, however, communicate by means of remote procedure calls without channels connecting the enclosing blocks.

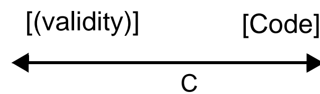
delaying channel



A *delaying* channel is specified by a channel symbol with the arrows at the middle of the channel.

The delay of signals is non-deterministic, but the order of signals is maintained.

non-delaying channel



A *non-delaying* channel is specified as folows, that is with the arrows at the endpoints. Associated with each direction of a channel are the types of signals that may be conveyed by the channel. The list enclosed by the signal list symbol can be signals (as e.g. Code) or signal lists (as e.g. validity) enclosed in ().

Channels connected to the frame symbol represent the connections to the environment.

package reference clause

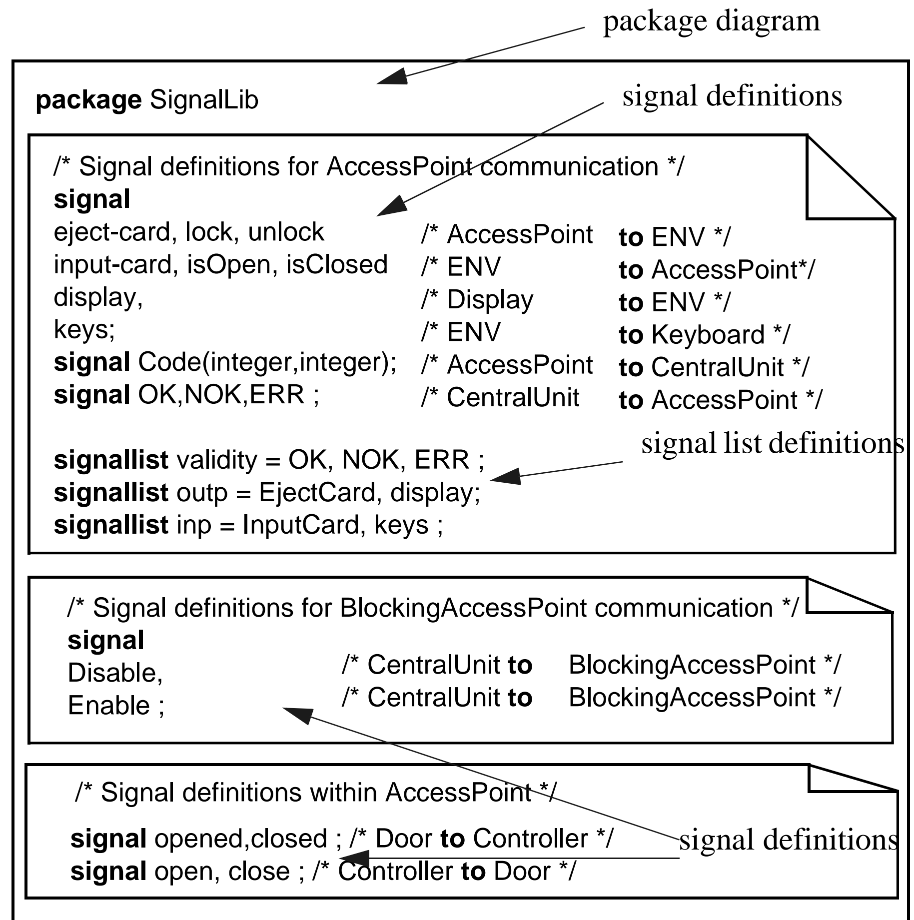
A package reference clause specifies that a system diagram or package diagram use the definitions of other packages. The names following the "/" after the package name denotes the subset of the definitions that are used.

The system uses the types defined in the packages SignalLib and the denoted types (AccessPoint, BlockingAccessPoint and LoggingAccessPoint) from the AccessPointLib package.

Package diagram, SignalLib

Figure 13-17: Package diagram SignalLib

Open figure



This package defines all the signals being used in the access control system.

Defining a package SignalLib makes all the signal type definitions become globally defined, and they may be used by more than one system (without "copy-paste"). It is of course possible to let additional signals be defined locally in order to restrict the contexts in which they will be used.

package

A package is a collection of types, defined by a package diagram. A package may in general contain definitions of types, data generators, signal lists, remote specifications and synonyms. Definitions within a package are made visible to a system definition or other package definitions by a package-reference-clause (use clause).

The package in Figure 13-17 (p.13-40) only contains definitions of signals.

signal definition

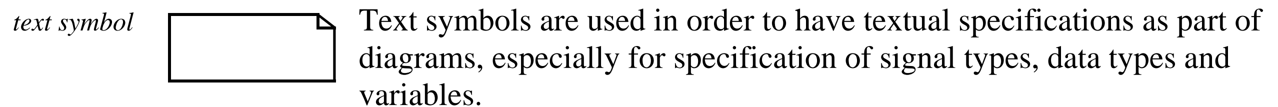
A signal definition defines a set of types of signals. A signal instance is a flow of information between processes, and is an instantiation of a signal type defined by a signal definition. A signal instance can be sent by either the environment or a process.

Signals may carry data values. The types of the values are specified as parameters of the signal definition. The signal Code defined in Figure 13-17 (p.13-40) is defined to carry two integer values.

Signals may be defined in system and block diagrams, and these may then be used for communication between the blocks of the system or the processes of the block. Signals may also be defined in process (type) diagrams, but then they can only be used for communication between processes of the same set. Often signal definitions are collected in packages.

signal list Often the lists of signals associated with channels and signal routes are quite comprehensive and diagrams become crowded. The notion of signallist helps on this. A signallist is a list of signals which has been given a name. Validity, inp and outp are signallists defined in the package and used in the system diagram.

text symbol



Text symbols are used in order to have textual specifications as part of diagrams, especially for specification of signal types, data types and variables.

There is no limit to the number of text symbols that may occur in a diagram. Text symbols are not connected to other symbols by flow lines.

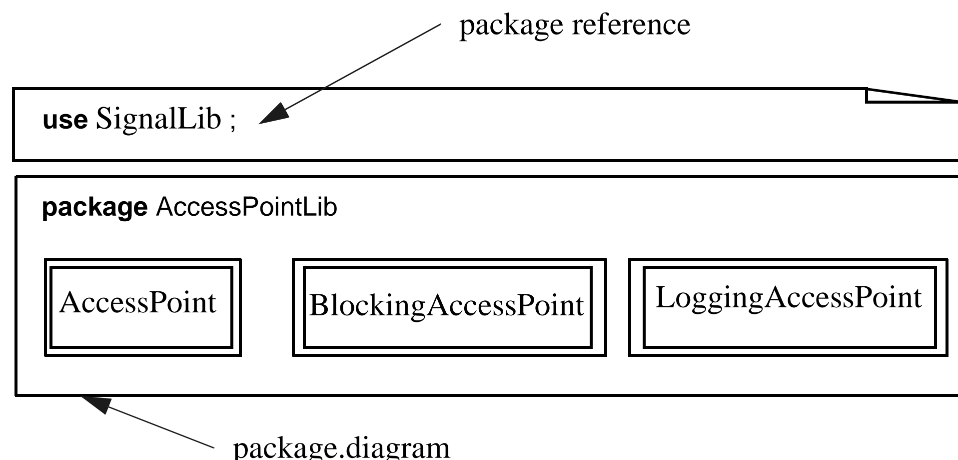
The text symbol is also used for the graphical representation of a use clause, see Figure 13-17 (p.13-40).

Package diagram, AccessPointLib

The AccessPointLib package uses the signals defined in the package SignalLib (by the use clause) and defines three block types.

Figure 13-18: Package diagram AccessPointLib

Open figure



TIMe Electronic Textbook v 4.0 © SINTEF Modified: 1999-07-16

block type reference

Block types are referenced by means of block type references. Block types are defined in block type diagrams, and they are referenced by means of block type references. The block type reference indicates in which block or system scope unit the block type is defined. The three block type references in Package diagram AccessPointLib (p.13-42) indicates that the scope of these are the package and not a specific system.

Note that the block type reference (as for block references) is merely a graphical shorthand for diagrams. Block type references may be substituted by block type diagrams.

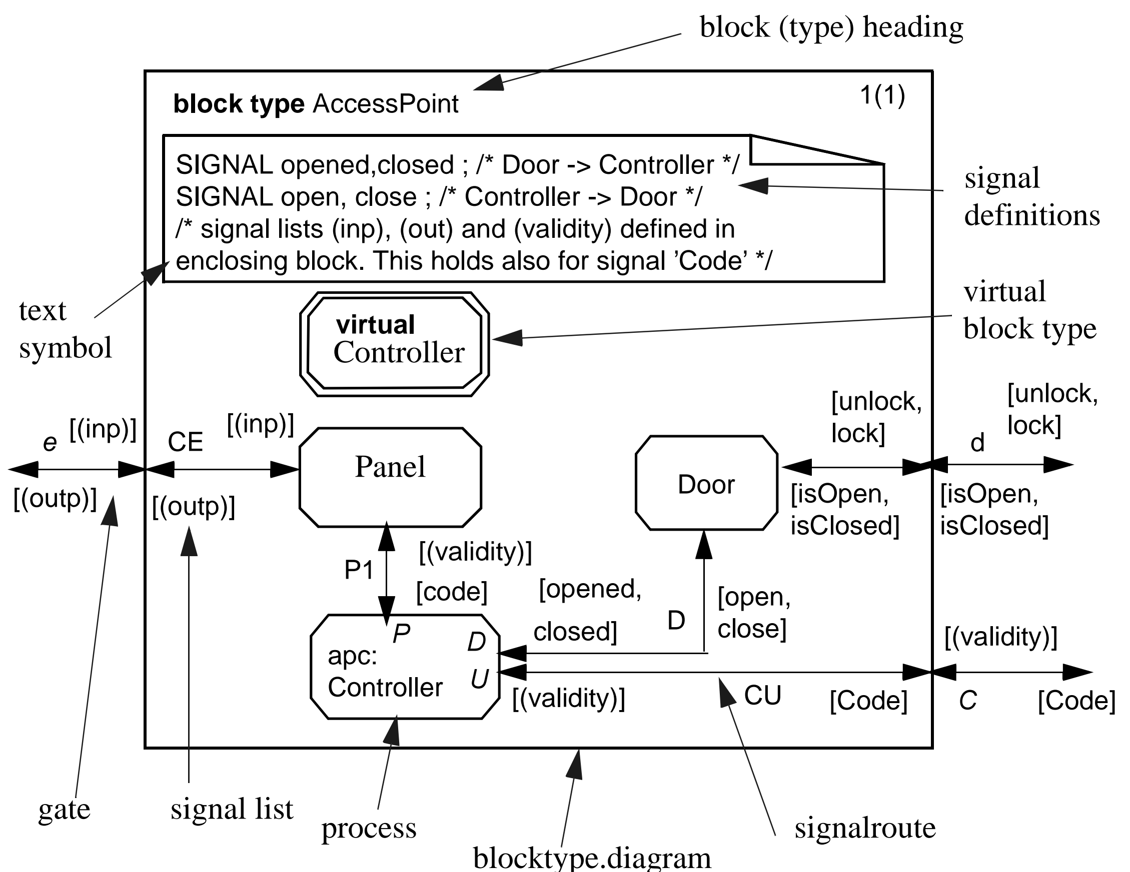
Block type diagram, AccessPoint

The block type AccessPoint defines the properties of a general type of access point in the system. The other types of access points (blocking and logging access points) are defined a subtypes of this.

Each access point shall handle the interaction with the user via a panel, communicate with the central unit and control the door.

Figure 13-19: Block type AccessPoint with virtual Controller process type

Open figure



This block type diagram defines the block type with name AccessPoint in the Access-PointLib package. Each block instance of this type will consist of three process sets (Panel, Door, apc). The first two are defined in corresponding process diagrams (they are really just process references), while apc is a set instances of process type Controller. The process type Controller is defined as a virtual process type, with the keyword VIR-TUAL, so that specialisations of AccessPoint may replace that definition with their own definition.

The Panel takes of the physical panel, the Door process takes care of controlling the physical door, while the Controller process handles the communication with the Cen-tralUnit in order to validate users of the access point.

Note the identifiers e and C which in the system diagram occurs inside the block set ap. These identifiers designate gates. Gates are used to indicate which channels of the block type are supposed to connect to which channel connecting an instance of the type. The gate names are defined by the type and visible wherever the type name is visible. Note also that the gate symbols have arrows at the ends and that signal lists are associated with the arrows. The signallists are constraints on the gates and will ensure that the instances of the block type are connected properly to their surroundings.

block type

A block type defines the common properties for a category of blocks. All block of the same type will have the same properties, as specified in the block type diagram.

Block types may contain a connectivity graph of block instances connected by channels. This makes up a structure of nested blocks. At the leaves of this structure there are blocks which contain processes. Blocks cannot contain both blocks and processes at the same level.

In addition to containing structures of blocks or structures of processes, block types may contain other type definitions. This makes up the scoping hierarchy of SDL. Names in enclosing type definitions are the only names visible.

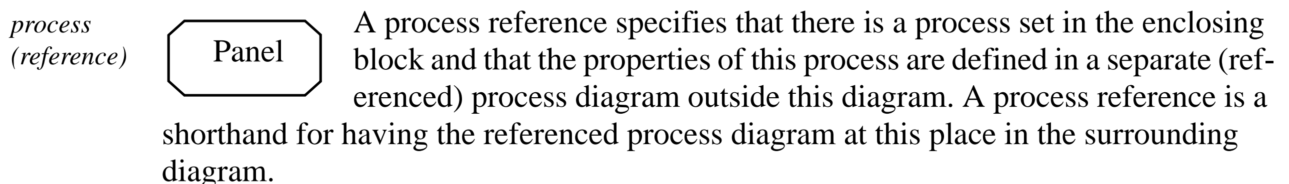
Block types may contain data type definitions, but no variable declarations. This follows from the fact that processes in SDL do .fmnot share data other than signal queues. They share a signal queue in the way that one process appends (output) signals to the queue (the input port), while the other process consumes (input) signals from the same queue. Appending and consuming signals are atomic, non-interruptible operations. The input port is the basic synchronisation mechanism of SDL.

Block types may contain process types, service types and procedures as well as block types and data types.

block (type) heading

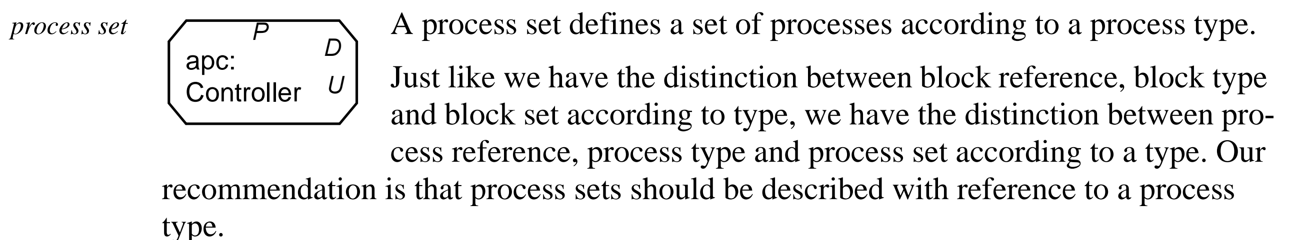
The heading of block type diagrams defines the name of the block type, possible formal context parameters, whether the block type is virtual or not and if it inherits from another block type. The block type in Figure 13-19 (p.13-44) does not have any context parameters and it is not virtual.

process (reference)



A process reference specifies that there is a process set in the enclosing block and that the properties of this process are defined in a separate (referenced) process diagram outside this diagram. A process reference is a shorthand for having the referenced process diagram at this place in the surrounding diagram.

process set

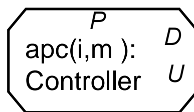


A process set defines a set of processes according to a process type.

Just like we have the distinction between block reference, block type and block set according to type, we have the distinction between process reference, process type and process set according to a type. Our recommendation is that process sets should be described with reference to a process type.

While Panel above is a process reference, and thereby a process set without any associated type, apc is a process set according to the process type Controller and therefore not a process reference.

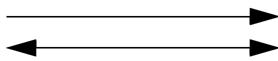
number of instances



In general process sets may have specified the number of instances in the set.

The numbers in parentheses after the process set name specifies the number of instances in the process set. As defined in above, there are initially no processes, and there is no limit on the number of instances that may be created.

signal route



A signal route represents a communication path between process sets and between process sets and the environment of the enclosing block/block type.

process type

A process type defines the common properties of a category of process instances. A process type is defined by a process type diagram.

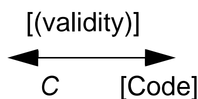
virtual process type

A virtual process type is a process type that can be redefined in a subtype of the enclosing block type.

The virtuality is specified in the process type heading or by <virtuality> in the corresponding process type reference symbol, as is done here for the process type Controller.

A redefinition of the process type must be a subtype of the type identified in the virtuality constraint. As specified here the process type reference symbol has *no* explicit virtuality constraint, which means that any redefinition will *extend* the given definition of Controller (the Controller is its own constraint).

gate



A gate is a potential connection point for channels/signal routes when connecting sets of blocks/processes/services. The same symbol is used in all cases.

Gates are defined in block/process/service types and used when connecting sets or instances of these with channels/signal routes.

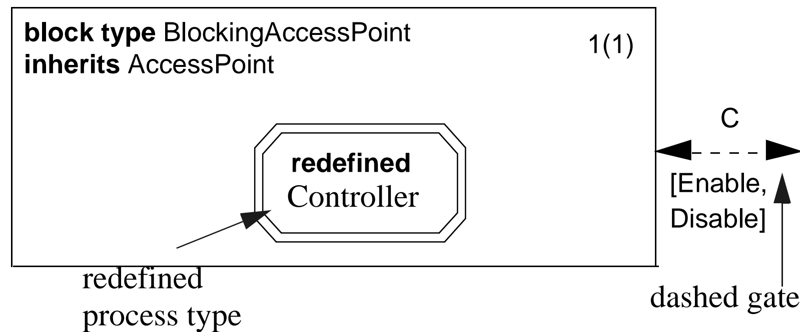
The signal list associated with the endpoints represents constraints (on incoming/outgoing signals) the gate.

TIMe Electronic Textbook v 4.0 © SINTEF Modified: 1999-07-16

Block type diagram, BlockingAccessPoint

Figure 13-20: Block type BlockingAccessPoint as a subtype of AccessPoint

Open figure



This block type defines a block type with name BlockingAccessPoint as a subtype of block type AccessPoint. It represents access points that may be blocked by some operator.

BlockingAccessPoints are quite similar to the plain AccessPoints.The only difference is that the BlockingAccessPoints shall be able to react to signals from the CentralUnit that plain AccessPoints will not recognise. BlockingAccessPoint will have a Door (which should not have a new definition), a Panel (which could have a new definition, but need not have a new definition) and a control process Controller which should be able to do the extended controlling.

A BlockingAccessPoint is a specialised AccessPoint where Controller is extended. This is expressed by the INHERITS clause of the block type heading.

The block type diagram specifies that BlockingAccessPoint inherits everything from AccessPoint, but it adds a redefinition of Controller and it adds two signal types on the inherited gate C: Enable and Disable. The fact the the gate is inherited is indicated by it being dashed.

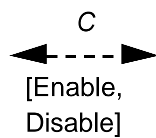
In general, entities defined in supertype, inherited in subtypes and for which some additional properties have to be specified in the subtype, are called existing entities, and in the graphical syntax they are dashed entities.

redefined process type

A redefined process type is a redefinition of the corresponding virtual process type in the super block type, and it is virtual, so that it can be redefined in further subtypes of this block type.

A redefinition of the process type must be a subtype of the type identified in the virtuality constraint. In this case the constraint is not explicitly specified; this implies that the definition of the virtual process type is its own constraint: the redefinition thereby defines an extension (a subtype) of the virtual process type.

dashed entity



A dashed entity is the graphical way of representing an entity that is inherited from a supertype and which needs to be used in the definition of the subtype. There are dashed block sets, process sets, services and gates.

The Z.100 terminology is existing entity.

An existing block set/block may be connected by channel, and these will then be there in addition to those specified in the super type.

An existing process set/service may be connected by signal routes, and these will then be there in addition to those specified in the super type.

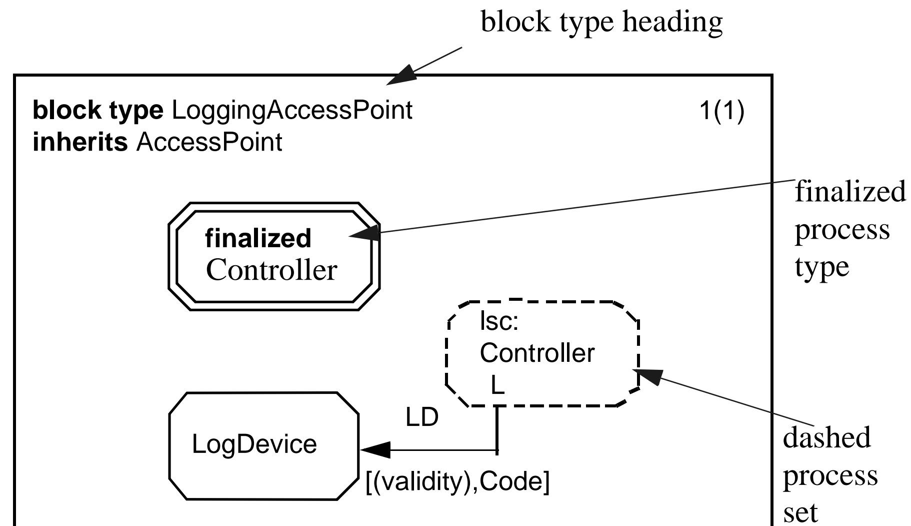
An existing gate can have constraints in terms of signals on the endpoints of the gate specified, and these are then added to the inherited gate and will then apply in addition to those of the inherited gate.

In the textual version of a specification, inherited entities are simply identified by name.

Block type diagram, LoggingAccessPoint

Figure 13-21: LoggingAccessPoint as a subtype of AccessPoint

Open figure



This block type defines a block type with name LoggingAccessPoint as a subtype of block type AccessPoint, *adding* the process LogDevice.

With LoggingAccessPoint it is not sufficient to only modify the Controller, since there is an addition to the block, namely the LogDevice. The LogDevice must be connected to the Controller along a signalroute (which is added compared with the supertype AccessPoint). lsc has been defined in the AccessPoint definition and is dashed here.

We notice the keyword FINALIZED in the process type reference. This has a slightly different meaning than REDEFINED.

finalised process type

A finalised process type is a redefinition of the corresponding virtual process type in the super block type, and it is **not** virtual, so that it can **not** be redefined in further subtypes of this block type.

A final redefinition of the process type must be a subtype of the type identified in the virtuality constraint.

A redefined type can be redefined again in yet another specialisation. A finalised type cannot be redefined. There is a subtle point to making this distinction. Virtual and redefined types are very flexible, but analysis becomes more uncertain since some components may not be entirely known. Finalised types are not flexible any more, they are completely known and, therefore, analysis can be certain.

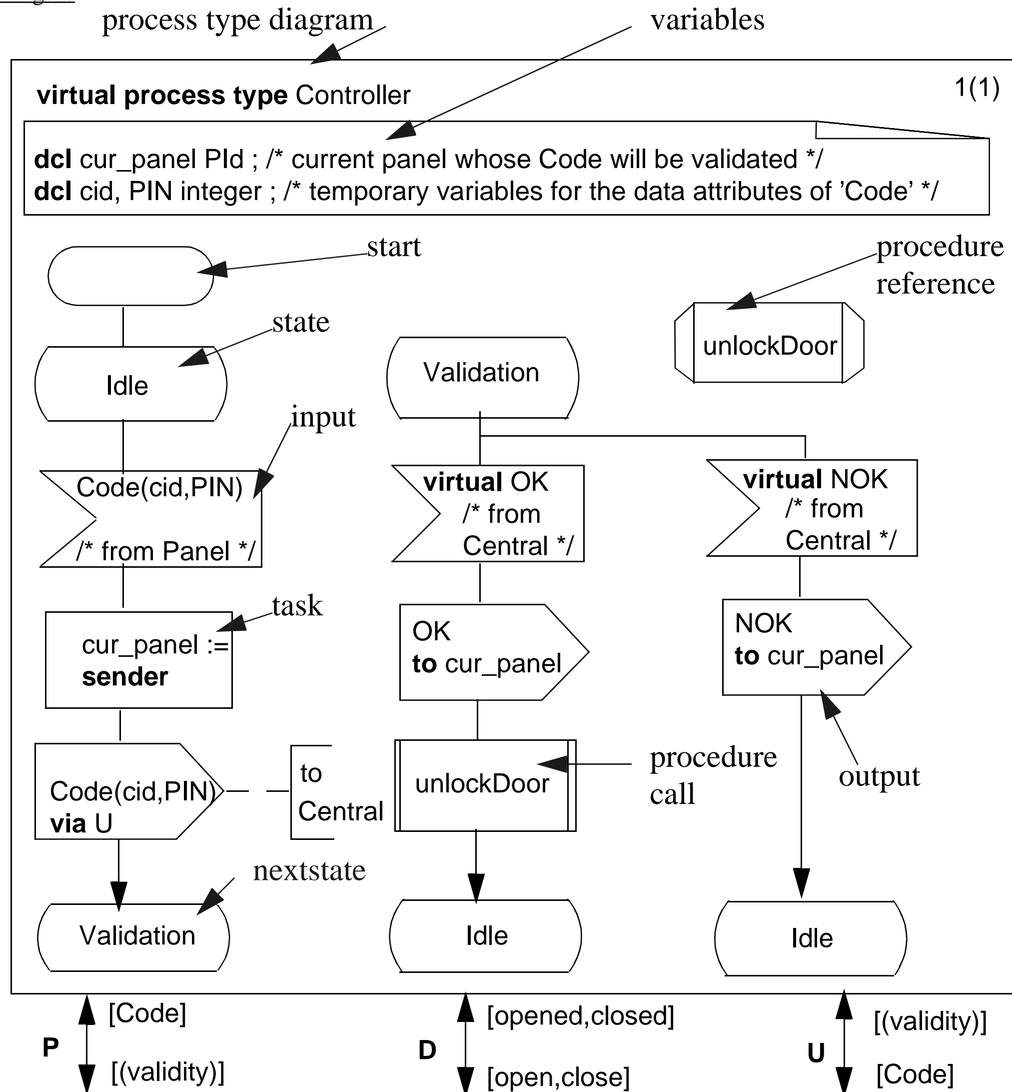
The new signalroute LD indicates that it is not be possible to derive the finalised Controller by only adding a number of new transitions to the basic Controller. In order to get new transitions, we need either new input signals or new states. The Controller of LoggingAccessPoint has neither new signals, which can be seen from the channels to the *lap*

set of logging access points, nor new states. In fact the LogDevice should be invoked for most transitions since the requirement was to trace the transactions. Then our need is to modify (redefine) some of the existing transitions.

Process type diagram, Controller

Figure 13-22: Virtual process type Controller

Open figure



This process type heading defines the process type Controller as a virtual process type. This means that the process type can be redefined in a subtype of the enclosing block type.

Plain AccessPoints have their own (default) definitions of Controller.

A Controller process will start executing the start transition. In this case the start transition is empty and simply leads to the Idle state. The process will remain in the Idle state until it receives an input signal. It expects to receive a Code signal containing information about the card id and personal identity number from the Panel. It may, however, be

prepared to receive other signals as well. The Idle state is followed by one input symbol which describes the consumption of the signal Code. If the process is in the Idle state and signals other than Code are received, they will be discarded.

We have defined three process gates P, D and U with associated process gate constraints. We note that the enclosing AccessPoint definition uses these gates in connection with the instance lsc of Controller.

Within the process type diagrams, the gates appear as identifiers in the VIA-clause of the output symbols.

When we want to analyse the type enclosing the virtual type (here, block type AccessPoint) we wish to know something about the instances of the virtual types even though we know they may be redefined in subtypes. At least we must know the static interface, i.e. the gates. Very often we would like to know more about the type and, therefore, the header of a virtual type may include a virtuality constraint. The virtuality constraint is of the form “atleast type-identifier”. All “matches” (redefinitions and finalisations) of the virtual must be specialisations of the type referred to by the type-identifier of the constraint.

process type diagram

A process type diagram defines the properties of a process type. A process type defines the common properties of a category of process instances. A process type is defined by a process type diagram.

process type heading

The heading of process type diagrams defines the name of the process type, its virtuality (and constraint), its formal context parameters and if it inherits from another process type. The heading in Figure 13-22 (p.13-51) defines a virtual process types without any context parameters and without any parameters.

variables in processes

Variables can be defined in processes, services and procedures. They are defined in text symbols.

SDL supports predefined types including Character, Boolean, Integer, Natural, Real and PId (Process Instance Identifier). The variables cid and PIN in Figure 13-22 (p.13-51) are defined to be of type Integer, while the variable cur_panel is of type PId, which means that it denotes a process instance.

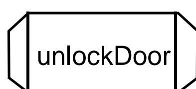
For a short introduction to the definition of user-defined types see Specifying properties of variables: data types (p.-24).

Variables of process are created as part of the creation of the process instance.

Variables will get default initial values if nothing else is specified.

The following elements of SDL are used in the definition of Controller behaviour.

procedure reference



A procedure reference specifies that there is a procedure in the enclosing entity and that the properties of this procedure are defined in a separate (referenced) procedure diagram outside this diagram.

In the example here, unlockDoor is a procedure defined locally to Controller, and it is referenced by the symbol containing “unlockDoor” - that is there is a procedure diagram defining the properties of unlockDoor.

TIMe Electronic Textbook v 4.0 © SINTEF Modified: 1999-07-16

start

There is only one start symbol for a process. The transition from the start 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.

The start transition in the Controller process is empty, that is there are no actions, so the process just enters the Idle state upon start.

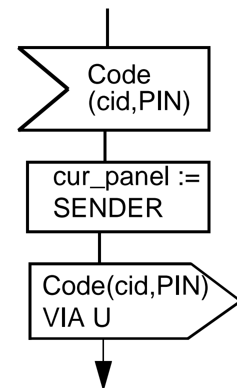
transition

A transition performs a sequence of actions. During a transition, the data of a process may be manipulated and signals may be output.

Actions may be

- task,
- output,
- set,
- reset,
- export
- create request,
- procedure call, or
- remote procedure call.

Example of a tran-tion from process type Controller, with a task followed by an output.

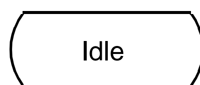


The transition will end with the process entering a

- next state,
- with a stop,
- with a return or
- with the transfer of control to another transition.

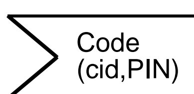
The controller process has three transitions: one starting in the state Idle and two in the state Validation. They are all input transitions, that is they are triggered by the consumption of a signal from the input queue of the process.

state



A state represents a particular condition in which a process may consume a signal resulting in a transition. If the state has neither spontaneous transitions nor continuous signals, and there are no signal instances in the input port, otherwise than those mentioned in a save, then the process waits in the state until a signal instance is received.

input



An input allows the consumption of the specified input signal instance (here of type Code).The variables associated with the input (here cid and PIN) are assigned the values conveyed by the consumed signal.

The values will be assigned to the variables from left to right. If there is no variable associated with the input for a sort specified in the signal, the value of this sort is discarded. If there is no value associated with a sort specified in the signal, the corresponding variable becomes “undefined”.

The sender expression of the consuming process is given the PId value of the originating process, carried by the signal instance.

virtual (input) transition

A virtual input transition specifies that subtypes of type with this transition may redefine it, that is it must input the signal in the state, but the following transition may be redefined

A virtual input transition is a special case of a general notion of virtual transition:

- virtual priority input,
- virtual start,
- virtual spontaneous transition.

In addition a save may be specified as a virtual save.

Redefinition of virtual transitions/saves corresponds closely to redefinition of virtual types:

- A virtual start transition can be redefined to a new start transition.
- A virtual priority input or input transition can be redefined to a new priority input or input transition or to a save.
- A virtual save can be redefined to a priority input, an input transition or a save.
- A virtual spontaneous transition can be redefined to a new spontaneous transition.

task

A task may contain a sequence of assignment statements or behaviour specified in informal text.

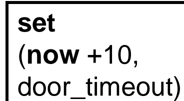
The example here is an assignment of (the predefined) SENDER, that is the sender of the signal triggering the transition of which this task is a part, to a PId variable cur_panel.

timer

In addition to assignments, task may specify the setting and resetting of timers. Timers are just like alarm clocks. The process waiting for a timer is passively waiting since the process needs not sample them. Timers will issue time-out signals when their time is reached.

A timer is declared similarly to a variable.

set timer



Timers are **set** and **reset** in tasks. 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. **time** is a special data type and is mainly used in connection with timers. The expression “**now**+10” is a **time** value and it adds the **time** expression **now** and the duration 10 (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. You may add or subtract duration to **time** and get **time**. You may divide or multiply duration by a real and get duration. You may subtract a **time** value from another **time** value and get duration.

TIMe Electronic Textbook v 4.0 © SINTEF Modified: 1999-07-16

The timer signal can be input in the same way as ordinary signals:

The semantics of timers is this: a time value is **set** in a timer and it becomes active. When the time is reached, a signal with the same name as the timer itself will be sent to the process itself. Then the timer becomes inactive.

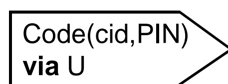
A timer may be **reset** and it then becomes inactive and no signal will be issued. (If an inactive timer is **reset**, then it remains inactive.) A **reset** will also remove a timer signal instance already in the input port. This happens when the timer has expired, but the time-out signal has not been consumed.

If an active Timer is **set**, the **time** value associated with the timer receives a new value. The timer is still active. If a timer is **set** to a **time** which is already passed, the timer will immediately issue the time-out signal.

Timer signals may contain data as other signals may contain data. Different parameter values in **set** means generation of several timer instances. **reset** must match these parameter values to eliminate the correct timer instance.

For more details, see timers.

output



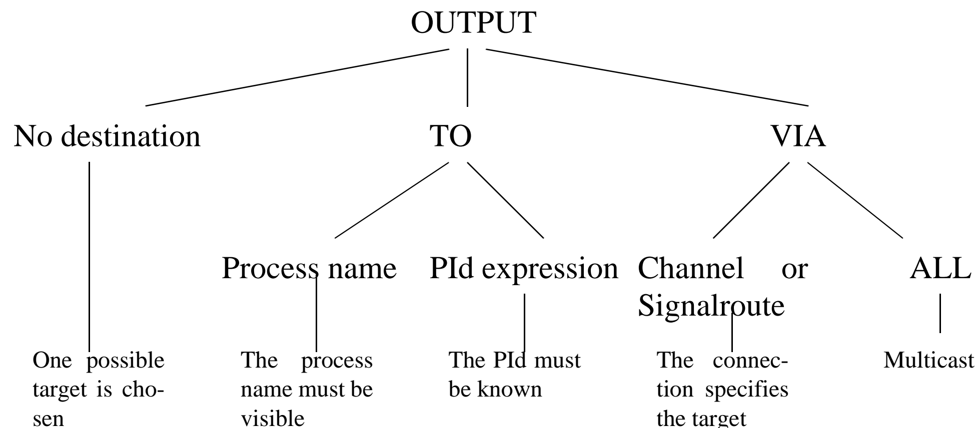
An output generates a signal of the specified signal type (here Code), containing the specified actual parameters (here cid and PIN), and send this signal instance to the specified destination.

The destination of a signal can be specified in various ways. The output symbol may in addition to the signal name (and actual parameters) contain a **to**-clause and/or a **via**-clause.

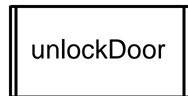
When the **to**- and **via**-clauses are omitted, there should be a unique destination for the signal based on the signal identifier. If there is a set of possible destinations, one of the destinations will be chosen non-deterministically. In our case the path and destination follow implicitly from the signalroutes and channels in the block diagrams.

When the **to**-clause is explicit, it specifies a process uniquely either by its (visible) name or by a “pointer” value. This “pointer” value in SDL is known as “PId” (Process Identifier). When a process is identified by its name in the **to**-clause, this means that it has to be within the same block since process names outside the block cannot be visible.

In order to specify the path the signal should follow, it is possible to append to the output statement a **via**-clause which lists the path of signalroutes and channels which the signal will be sent through. The VIA-clause may also specify a gate. Furthermore, the **via**-clause may be extended to “**via all**” and then if there is more than one channel instance in the path a signal instance will be generated for each channel instance. This happens for example when we have block sets. This is how we can describe a multicast message.



procedure call



A procedure call transfers the interpretation to the procedure definition referenced in the call, and that procedure graph is interpreted.

The interpretation of the transition containing the procedure call continues when the interpretation of the called procedure is finished.

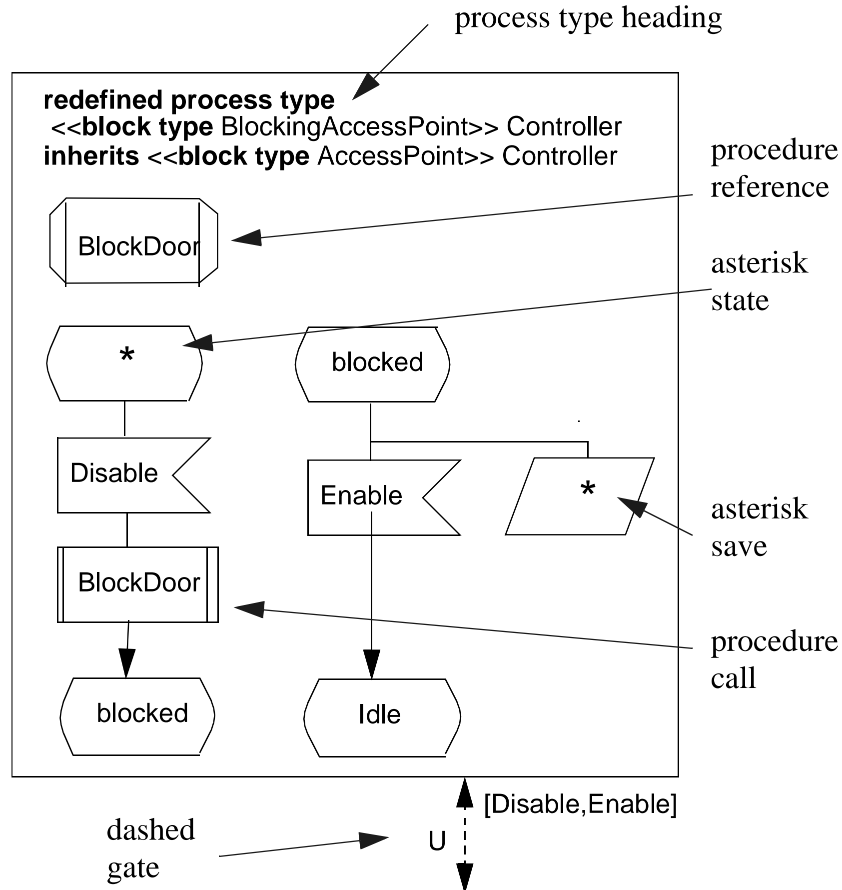
Note that a procedure call symbol has one and only one entrance and one and only one exit. As specified here, the procedure has no parameters.

TIMe Electronic Textbook v 4.0 © SINTEF Modified: 1999-07-16

Process type diagram, redefined Controller in BlockingAccessPoint

Figure 13-23: Redefined process type with added states and transitions

Open figure



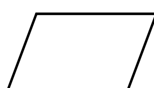
This process type defines the process type Controller (in block type BlockingAccess-Point) as a redefinition of the corresponding virtual process type in block type AccessPoint.

It is also specified that it inherits the same process type. This is, however, not necessary, as by default a redefinition of a virtual type without an explicit constraint will inherit the properties of the virtual type.

Inheritance of a process type implies inheritance of all states and transitions of the super-type. The asterisk state implies all states, also the inherited. The state Idle indicated as nextstate is the state Idle defined in the supertype.

For more details on this mechanisms, see virtual types and specialisation.

save

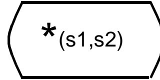


A save specifies that the signals in the save symbol are retained in the input port in the order of their arrival.

As specified in Figure 13-23 (p.13-57) (an asterisk save) all signals except Enable are saved. For a given state there may be only one asterisk save,

The effect of the save is valid only for the state to which the save is attached. In the following state, signal instances that have been “saved” are treated as normal signal instances.

asterisk state



An asterisk state is a shorthand for all states except those listed in an accompanying asterisk state list.

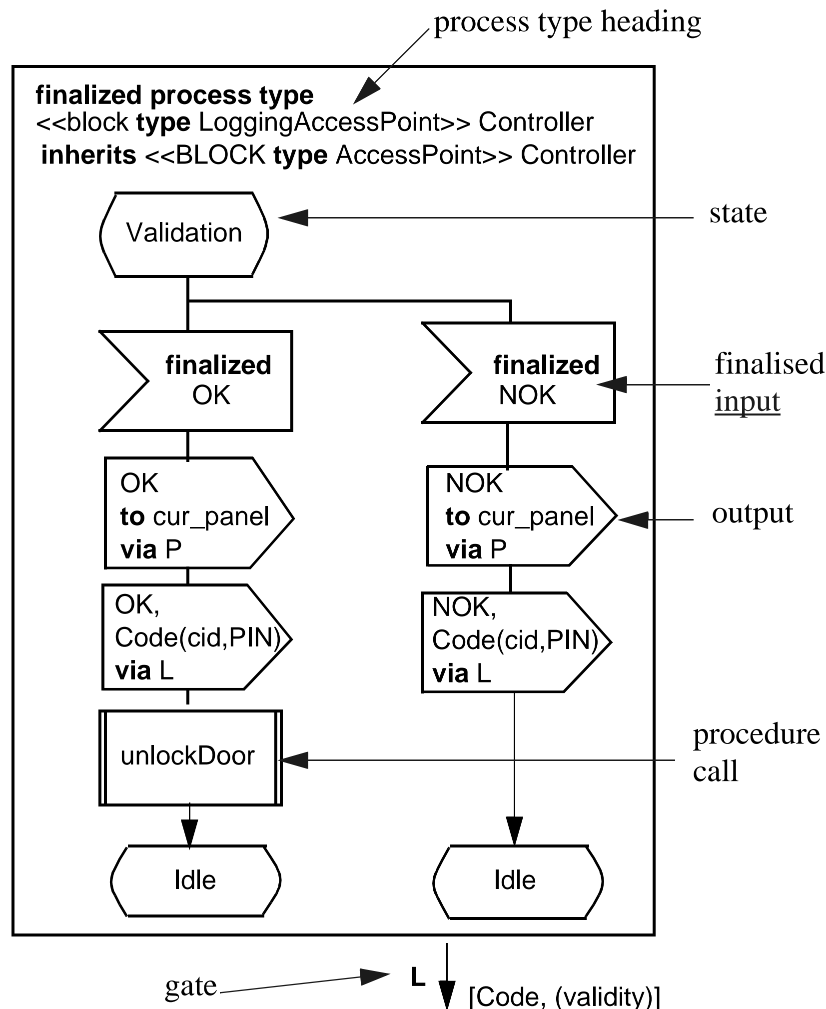
The state names in an asterisk state list must be distinct and must be contained in other state list in the enclosing body or in the body of a supertype. As specified here, the asterisk state implies a state (with the corresponding transition) for each of the states except s1 and s2.

TIMe Electronic Textbook v 4.0 © SINTEF Modified: 1999-07-16

Process type diagram, finalised Controller in LoggingAccessPoint

Figure 13-24: Finalised process type

Open figure



This process type defines the process type Controller (in block type LoggingAccess-Point) as a finalised redefinition of the corresponding virtual process type in block type AccessPoint. This means that it is not virtual, so it can not be redefined in subtypes of the enclosing block type.

It is also specified that it inherits the same process type. This is, however, not necessary, as by default a redefinition of a virtual type without an explicit constraint will inherit the properties of the virtual type.

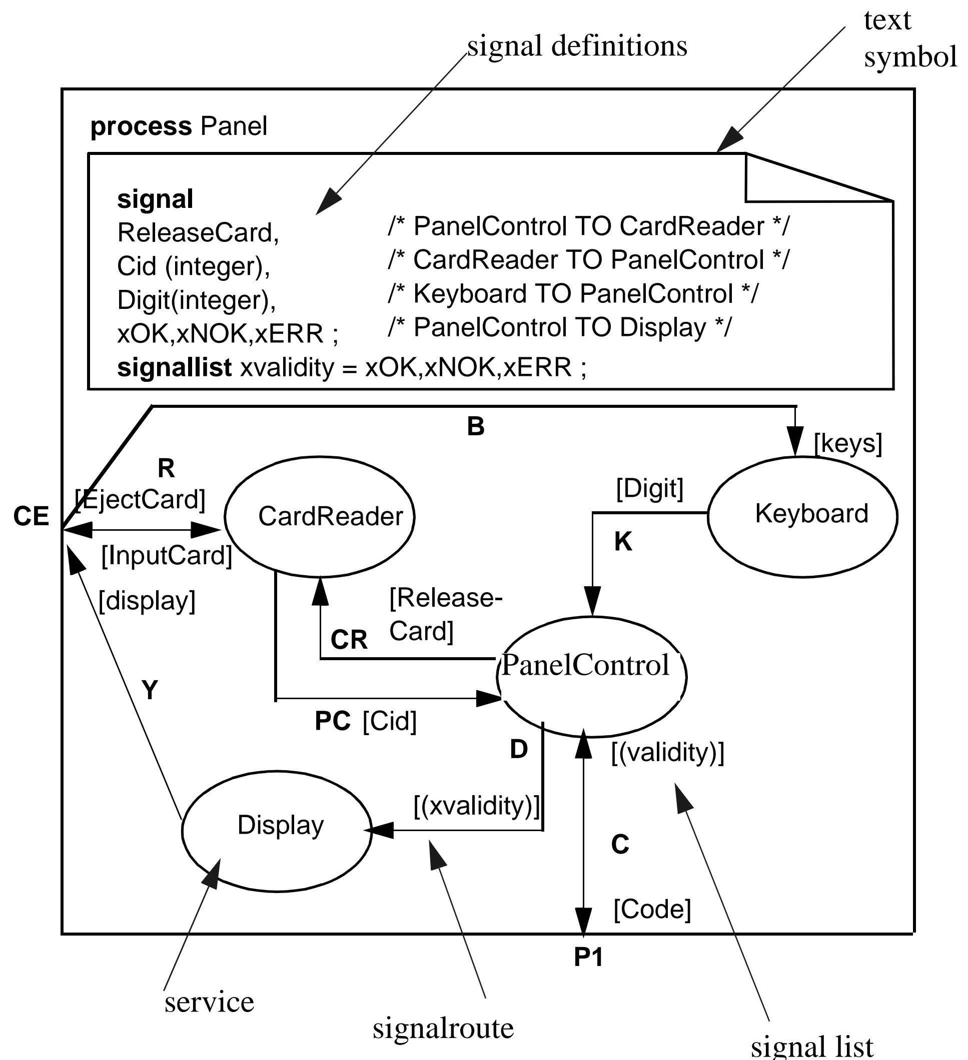
All transitions are inherited from the supertype, except the transitions starting with the state Validation and the signals OK and NOK. The are redefined in this process type.

For more details on this mechanisms, see virtual types and specialisation.

Process diagram, Panel in terms of services

Figure 13-25: Process in terms of services

Open figure



process diagram A process diagram defines the properties of a process set, where each of the process instances in the set have the specified properties.

The behaviour of processes may be defined either by means of a procedure graph (states and transitions) or by means of a substructure of services connected by signal routes. The behaviour of each of the services is defined by means of states and transitions. The process defined in Figure 13-25 (p.13-60) is defined by means of services.

process heading The heading of process diagrams (defining a process set directly without any process type) defines the name of the process set and the initial/maximum number of instances in the set.

TIMe Electronic Textbook v 4.0 © SINTEF Modified: 1999-07-16

formal parameters

If the process shall have formal parameters they are also specified as part of the process heading. Formal parameters are (local) variables of the process instances. They get values as part of the creation of the process instance.

When a system is created, the initial processes are created in arbitrary order. The formal parameters of these initial processes have no associated values; i.e. they are undefined.

If the initial number is omitted (as in Figure 13-25 (p.13-60)), then the (default) value is 1. If the maximum number is omitted, then there is no limit on the number of instances.

service composition

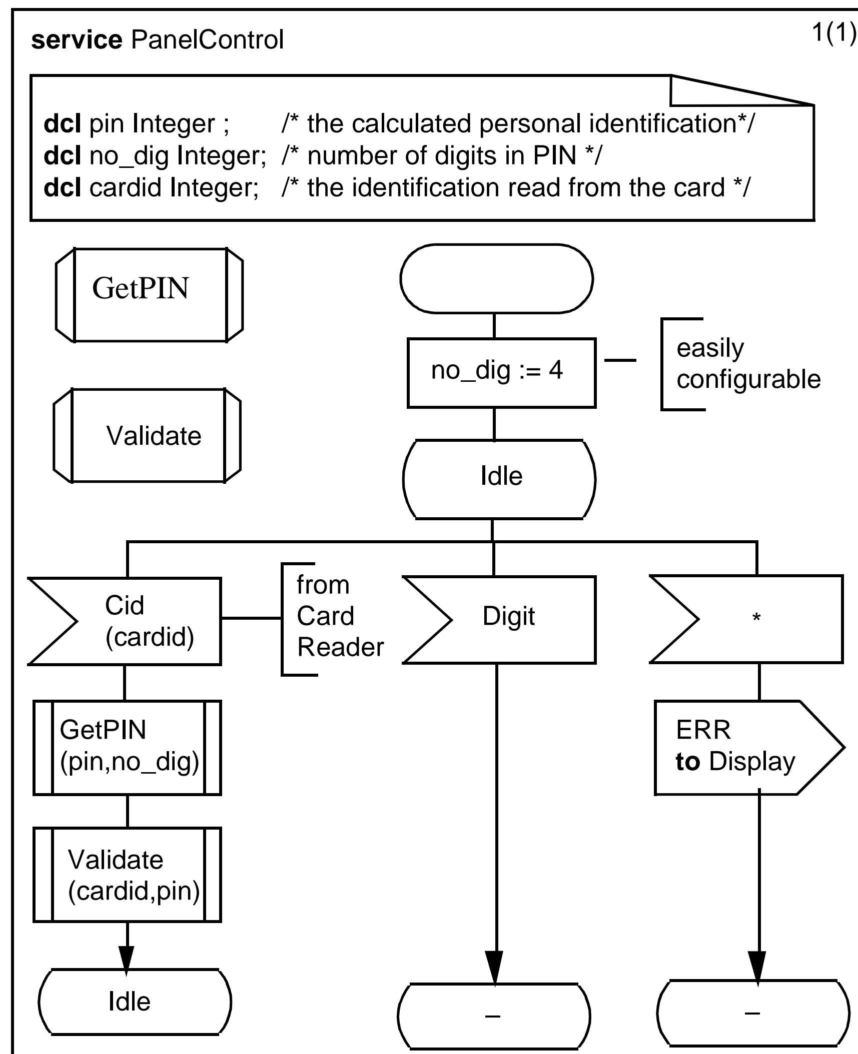
As defined in Figure 13-25 (p.13-60) the processes of this process set are defined by means of a composition of services. Service instances are components of the process instance, and cannot be addressed as separate objects. They share the input port and the expressions **self**, **parent**, **offspring** and **sender** of the process instance.

A service instance is a state machine, and it is described as in Figure 13-26 (p.13-62).

Service diagram, PanelControl

Figure 13-26: Service diagram, PanelControl

Open figure



When the process instance is created, the service starts are executed in arbitrary order. No state of any service is interpreted, before all service starts have been completed. A service start is considered completed when the service instance for the first time enters a state (possibly inside a called procedure) or interprets a stop.

Only one service at a time is executing a transition. When the executing service reaches a state, the next signal in the input port (which is not saved by the service, otherwise capable of consuming it) is given to the service that is capable of consuming it.

When a service ceases to exist, the input signals for that service are discarded. When all services have ceased to exist, the process instance ceases to exist.

TIMe Electronic Textbook v 4.0 © SINTEF Modified: 1999-07-16

variables in services

Variables can be defined in processes, services and procedures. They are defined in text symbols.

Variables of services are created when the service is created as part of the creation of the containing process instance.

Variables will get default initial values if nothing else is specified.

procedure call with parameters

A procedure may have formal parameters, and in the call the actual parameters are provided.

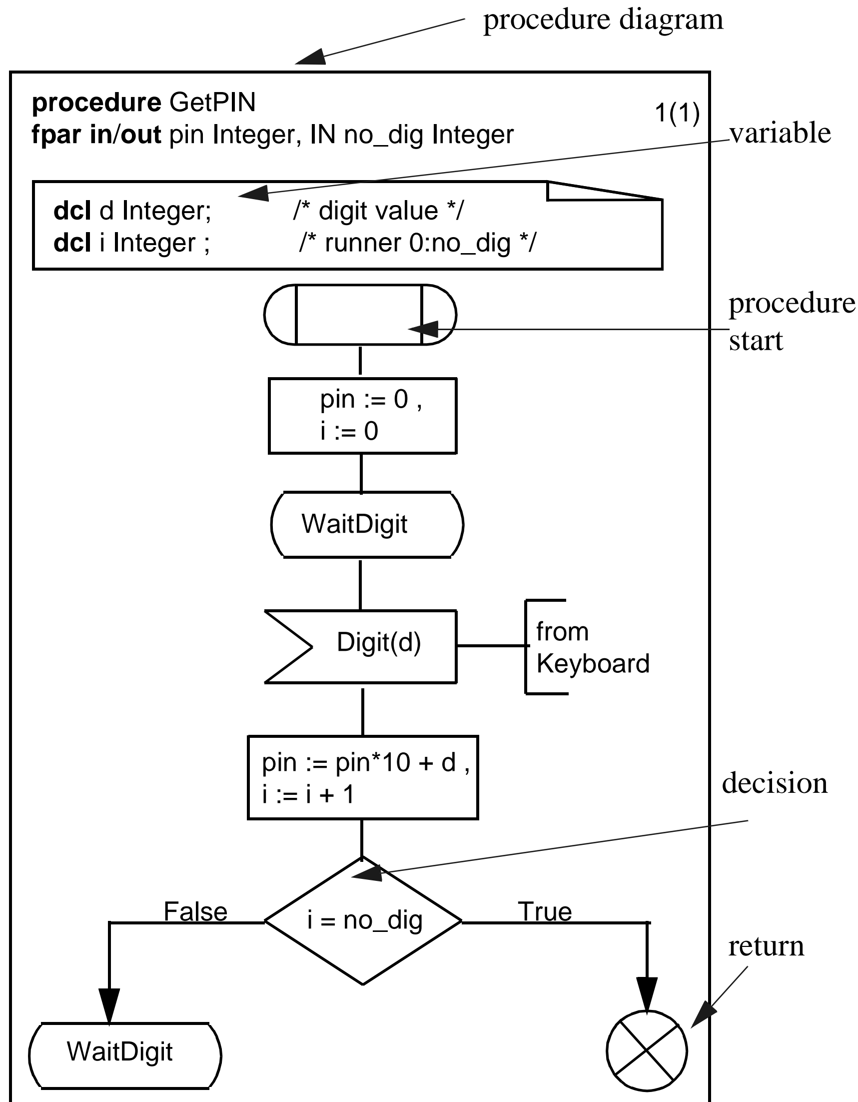
The pin parameter is **in**/**out** which means that the actual parameter corresponding to formal pin will be updated whenever the formal pin is updated within GetPIN. This is just like var parameters in Pascal or reference parameters in C++. The no_dig parameter is an **in** parameter which means that the procedure will have a local variable with the name of the parameter. This variable will assume the value of its corresponding actual argument at entry. Changes in the value of **in** parameters will not be transmitted to the actual argument. This is just like traditional value parameters.

Procedure diagram, GetPIN

The PanelControl service referenced in Figure 13-26 "Service diagram, PanelControl" (p.13-62) is defined by the service diagram in Figure 13-27 "Procedure diagram, GetPIN" (p.13-63).

Figure 13-27: Procedure diagram, GetPIN

Open figure



procedure

Procedures define patterns of behaviour that processes/services may execute at several places or several times during their life-time. The behaviour of a procedure is defined in the same way as for processes (that is by means of states and transitions), a procedure may have (local) variables, and in addition it may have **in**, **out**, **in**/**out** parameters.

State names are not visible outside the procedure. The process states are not visible within the procedure.

The procedure in Figure 13-27 (p.13-63) accepts a number of Digits as input signals in the state WaitDigit. The local variable i is increased by one for each digit, and when i equals the required number of digits, the procedure returns.

local variable

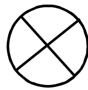
A procedure variable is a local variable within the procedure instance. It is created when the procedure start is interpreted, and it ceases to exist when the return of the procedure graph is interpreted. Variables will get default initial values if nothing else is specified.

TIMe Electronic Textbook v 4.0 © SINTEF Modified: 1999-07-16

procedure start

The start transition of a procedure is slightly different from the the start of process/service.

return



Procedure calls may be actions or part of expressions (value returning procedures only). A value returning procedure is a procedure where an expression is associated with the return, and the value of this expression is returned.

The interpretation of a procedure call causes the creation of a procedure instance and the interpretation to commence in the following way:

formal parameter

1. A local variable is created for each **in** parameter, having the name and sort of the **in** parameter. The variable gets the value of the expression given by the corresponding actual parameter if present. Otherwise the variable gets no value, i.e. it becomes “undefined”.
2. A formal parameter with no explicit attribute has an implicit in attribute.
3. A local variable is created for each variable definition in the procedure-definition.
4. Each **in**/**out** parameter denotes a variable which is given in the actual parameter expression. The contained Variable-name is used throughout the interpretation of the procedure graph when referring to the value of the variable or when assigning a new value to the variable.
5. The transition contained in the <procedure start area> is interpreted.

The nodes of the procedure graph are interpreted in the same manner as the equivalent nodes of a process or service graph, i.e. the procedure has the same complete valid input signal set as the enclosing process, and the same input port as the instance of the enclosing process that has called it, either directly or indirectly.

remote procedures

A procedure may be exported by a (server) process, so that other (clients) processes (clients) can request these procedures executed by the server.

The remote procedure mechanism consists of four interdependent language constructs:

1. *The exporting of a procedure*. A procedure which is made visible by other processes is marked with the keyword **exported** preceding the procedure heading, e.g. “**exported procedure** Validate ...” from a process within the CentralUnit. The exporting process can control in which states it will accept the remote request. It may also specify to save the request to other states. The controlling of the acceptance is done by using input and save symbols with the remote procedure name preceded by the keyword **procedure**.
2. *The importing of a procedure*. When a process, service or procedure wants to import a remote procedure, it must specify the signature of this procedure in an “imported procedure specification” in a text area. The specification in our case would read: “**imported procedure** Validate; **returns** integer;” where the integer returned would give the result of the validation.
3. *The specification of remote procedure*. In SDL all names must be defined in a specific scope. Thus, the names of remote procedures must be defined in the context in which the actual definition of the procedure and the calls will be contained. In our case the definition of the procedure Validate is within the CentralUnit and the call is in Con-

troller of the AccessPoint. The scope unit enclosing all these is the system itself. There we will find a text area with the following text: "**remote procedure** Validate; **returns** integer;".

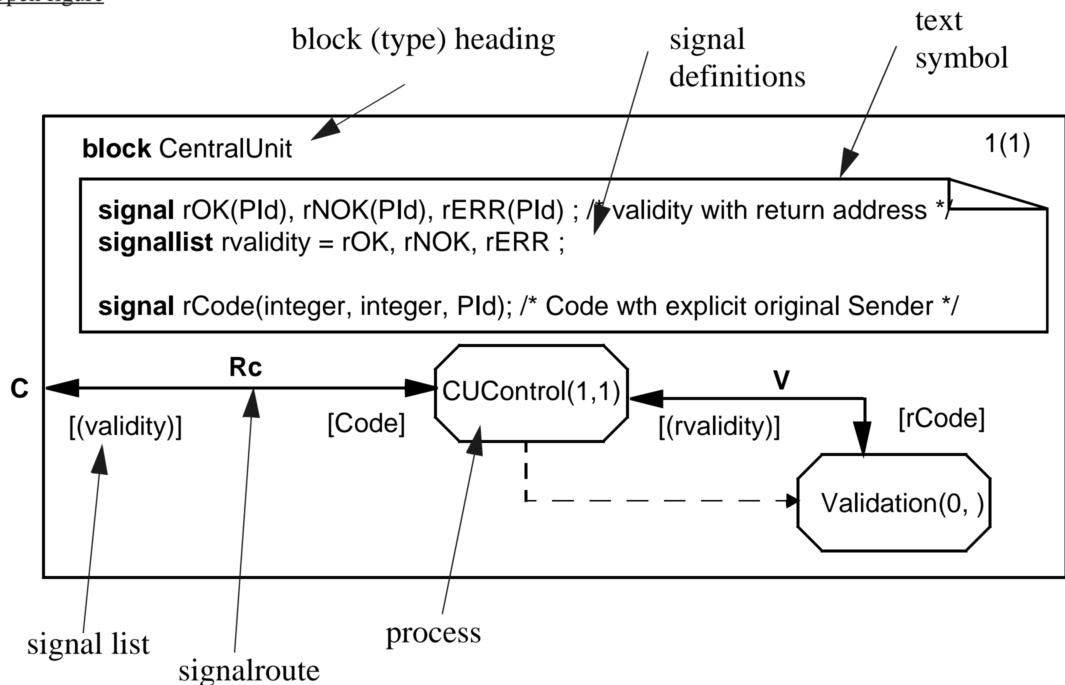
4. *The calling of a remote procedure*. The calling of the remote procedure is indistinguishable from local procedure calls unless the caller explicitly states which process it will request the procedure executed by. This can be done by a **to**-clause with a PId following the procedure name of the call.

Remote procedures may be value returning (as in our example above), and they may be virtual.

Block diagram, CentralUnit

Figure 13-28: Block diagram defining the CentralUnit

Open figure



create

A process may create processes in other process sets in the same block, possibly providing actual parameters to the new instance.

The create line (dashed line with arrowhead) indicates possible creations.

Create lines are optional.

create action

As specified in Figure 13-28 (p.13-66) the process CUControl creates Validation processes. In the process graph of CUControl, the creation will be specified by a create action.

TIMe Electronic Textbook v 4.0 © SINTEF Modified: 1999-07-16

List of figures

Behaviour Specification 4
Block type AccessPoint with processes 7
Block diagram of AccessPoint with block substructure 9
System design in SDL 10
Package diagram SignalLib 10
System using a package of type definition 12
Block type AccessPoint with virtual Controller process type 13
Virtual process type Controller 14
Block type BlockingAccessPoint as a subtype of AccessPoint 15
LoggingAccessPoint as a subtype of AccessPoint. 16
Redefined process type with added states and transitions 16
Finalised process type 17
Process in terms of services 19
Service diagram, PanelControl 19
Panel and card of an access control system 36
System diagram for access control system with three types of access points 38
Package diagram SignalLib 40
Package diagram AccessPointLib 42
Block type AccessPoint with virtual Controller process type 44
Block type BlockingAccessPoint as a subtype of AccessPoint 47
LoggingAccessPoint as a subtype of AccessPoint. 49
Virtual process type Controller 51
Redefined process type with added states and transitions 57
Finalised process type 59
Process in terms of services 60
Service diagram, PanelControl 62
Procedure diagram, GetPIN 63
Block diagram defining the CentralUnit 66

List of definitions

asterisk state 69
block. 69
block set 70
block type. 70
block type diagram. 71
block (type) heading 71
block type reference. 71
channel 71
create 72
dashed entity 72
decision 72
diagram heading. 73
entity kinds. 73
environment 74
finalised input. 74
finalised process type. 74
gate. 75
identifier. 75
input 76
local variables 76
output 76
package 77
package reference clause 77
page numbering 77
procedure 78
procedure call. 78
procedure heading 78
procedure reference 79
process 79
process diagram 79
process type 80
process type diagram 80
process (type) heading 80
process (reference). 80
process set 80
redefined process type 81
remote procedures 81
return 82
save 82
scope units 83

TIMe Electronic Textbook v 4.0 © SINTEF Modified: 1999-07-16

service 83
service (reference) 83
service (type) heading 84
signal definition 84
signal list 84
signal route 84
specialisation 84
start 85
state 85
system 86
system (type) heading 86
task 86
transition 86
text symbol 87
timer 87
variable definition 88
virtual process type 89
virtuality 89
virtuality constraint 89
virtual (input) transition 89

asterisk state

An asterisk state is a shorthand for all states except those listed in an accompanying asterisk state list.

The state names in an asterisk state list must be distinct and must be contained in other state list in the enclosing body or in the body of a supertype.

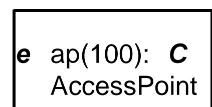
Z.100

block

A block is a container of processes (or of blocks, that in turn may contain processes or blocks etc.). Processes of a block are contained in process sets that are connected by signal routes.

A block is created as part of the creation of the enclosing block or system. All blocks are created as part of the system creation, that is there is no dynamic creation of blocks.

A block is specified either directly (singular block), like *CentralUnit,* or as a block set according to a block type. The block set *ap* is not a reference (as *CentralUnit*). Instead it designates a set of block instances. The example here specifies a set of 100 blocks of type AccessPoint.



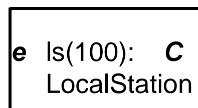
In the latter case, the AccessPoint must have been defined as a block type, as shown here:

The block *CentralUnit* is defined in a separate block diagram, while the properties of the blocks in the *ls* block set is defined by the block type *LocalStation*. A block type is defined by a block type diagram. To see a block type defined in terms of a substructure of blocks, look at block type diagram of AccessPoint with block substructure.

Z.100

block set

Type-defined blocks are contained in block sets. A block set is a fixed number of blocks with properties according to a block type.



The set of LocalStations is called ls and the number (100) designates the cardinality of the set. All the block instances within a block set typically have the same relationship with its surroundings (given by the channels).

A channel connected to a block set (via the gates e or C) will actually represent a set of channel instances.

A block set is not an array, so the thirteenth block cannot be identified by e.g. ls(13). The number of elements in a block set is determined when the system is created, all blocks in the set are created as part of the creation of the system, blocks will be permanent part (instances) of the system instance, and sets of blocks cannot be created dynamically.

Z.100

block type

A block type defines the common properties for a category of blocks.

Block types are defined in block type diagrams, and these may be referenced by means of block type references.

Block types may contain a connectivity graph of block instances connected by channels. This makes up a structure of nested blocks. At the leaves of this structure there are blocks which contain processes. In SDL, block types may not contain both blocks and processes at the same time.

In addition to containing structures of blocks or structures of processes, block types may contain other type definitions. This makes up the scoping hierarchy of SDL. Names in enclosing type definitions are the only names visible.

Block types may contain data type definitions, but no variable declarations. This follows from the fact that processes in SDL do not share data other than signal queues. They share a signal queue in the way that one process appends (output) signals to the queue

TIMe Electronic Textbook v 4.0 © SINTEF Modified: 1999-07-16

(the input port), while the other process consumes (input) signals from the same queue. Appending and consuming signals are atomic, non-interruptible operations. The input port is the basic synchronisation mechanism of SDL.

Block types may contain process types, service types and procedures as well as block types and data types.

Z.100

block type diagram

A block type diagram defines the properties of a block type.

Z.100

block (type) heading

The heading of block diagrams defines the name of the block.

The heading of block type diagrams defines the name of the block type, possible formal context parameters, whether the block type is virtual or not and if it inherits from another block type.

Z.100

block type reference

Block types are defined in block type diagrams, and they are referenced by means of block type references. The block type reference indicates in which block or system scope unit the block type is defined.

Z.100

channel

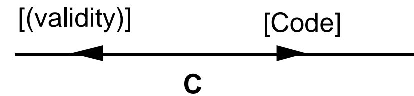
A channel is a one-way or two-way directed connection. It is characterised by the signals that it may carry; these constitute the signal list(s) of the channel. A channel has a signal list for each direction.

One or two arrows indicate the direction(s) of the channel.

Channels connect blocks or block sets with other blocks or block sets, or with the environment of the system. It provides a (one or two way) communication path for signals. If there is no channel between two blocks, then processes in these two blocks cannot communicate by signal exchange. Processes may, however, communicate by means of remote procedure calls without channels connecting the enclosing blocks. A channel cannot connect a block or block set with itself.

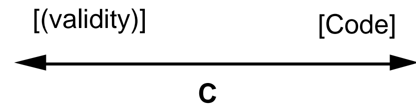
Channels may be delaying or non-delaying.

A *delaying* channel is specified by a channel symbol with the arrows at the middle of the channel:



The delay of signals is non-deterministic, but the order of signals is maintained.

A *non-delaying* channel is specified as follows, that is with the arrows at the endpoints:



Associated with each direction of a channel are the types of signals that may be conveyed by the channel. The list enclosed by the signal list symbol can be signals (as e.g. Code) or signal lists (as e.g. validity) enclosed in ().

Channels connected to the frame symbol represent the connections to the environment.

Z.100

create

A process may create processes in other process sets in the same block, possibly providing actual parameters to the new instance.

The create line (dashed line with arrowhead) indicates possible creations. Create lines are optional.

Z.100

dashed entity

A dashed entity is the graphical way of representing an entity that is inherited from a supertype and which needs to be used in the definition of the subtype. There are dashed block sets and process sets, services and gates.

The Z.100 terminology is *existing entity*.

An existing block set/block may be connected by channel, and these will then be there in addition to those specified in the super type.

An existing process set/service may be connected by signal routes, and these will then be there in addition to those specified in the super type.

An existing gate can have constraints in terms of signals on the endpoints of the gate specified, and these are then added to the inherited gate and will then apply in addition to those of the inherited gate.

In the PR version of a specification, inherited entities are simply identified by name.

Z.100

decision

A decision transfers the interpretation to the outgoing path whose range condition contains the value given by the interpretation of the question.

TIMe Electronic Textbook v 4.0 © SINTEF Modified: 1999-07-16

Z.100

diagram heading

In the upper left-hand corner of the first page of diagrams, we find the heading.The heading defines the name of the entity, it may contain definition of formal parameters, context parameters, it may specify if a type inherits from another type and the virtuality of atype (virtual, redefined or finalised).

The heading of the first page of a diagram must be a full heading of the form:

<heading> ::= <kernel-heading> [<additional-heading>]

while

the following pages only need a kernel heading:

<kernel-heading>::= [<virtuality>] [exported]

<diagram-kind> [<qualifier>] <diagram-name>

The kernel heading depends upon the diagram kind, see

- system (type) heading
- block (type) heading
- process (type) heading
- service (type) heading
- procedure heading

entity kinds

SDL defines the folowing different kinds of entities:

- packages
- system
- system types
- blocks
- block types
- channels
- signal routes
- signals
- gates
- timers
- block substructure
- channel substructures

- processes
- process types
- services
- service types
- procedures
- remote procedures
- variables (and formal parameters)
- synonyms
- literals
- operators
- remote variables
- data types
- generators
- signal lists and
- views.

environment

The environment consists of a set of SDL processes that may send signals to the system and which may receive signals from the system.

Z.100

finalised input

A finalised input is a redefinition of a virtual input transition that cannot be redefined in further subtypes. A virtual input is a special case of a virtual transition.

Z.100

finalised process type

is a finalised redefinition of the corresponding virtual process type in the super block type, and it is **not** virtual, so that it can **not** be redefined in further subtypes of this block type.

A final redefinition of the process type must be a subtype of the type identified in the virtuality constraint.

Z.100 (virtual types)

TIMe Electronic Textbook v 4.0 © SINTEF Modified: 1999-07-16

gate

A gate is a potential connection point for channels/signal routes when connecting sets of blocks/processes/services. The same symbol is used in all cases.

Gate are defined in block/process/service types and used when connecting sets or instances of these with channels/signal routes.

The signal list associated with the endpoints represents constraints (on incoming/outgoing signals) the gate.

Z.100

identifier

An identifier contains an optional qualifier in order to denote the scope unit in which the entity is defined:

<identifier>::= [<qualifier>] <name>

where qualifier defines the path:

<qualifier>::= <path-item>{‘/‘<path-item>;* |

‘<<‘ <path-item>{‘/‘<path-item>* ‘>>‘

The qualifier gives the path from either the system level, or from the innermost level from where the name is unique, to the defining scope unit.

Each path-item have this form:

<path-item>::= <scope-unit-kind>{<name> | <quoted-operator>}

where scope-unit-kind is one of

- package,
- system type,
- system,
- block,
- block type,
- substructure,
- process,
- process type,
- service,
- service type,

(more)

- procedure,
- signal,
- type, or

- operator.

A definition in an inner scope unit overrides definitions with the same name in outer scope units. Qualifiers may be used in order to identify overridden entities.

Qualifiers may be omitted if not needed in order to identify the right entity in the right scope unit.

States, connectors and macros cannot be qualified. States and connectors are not visible outside their defining scope unit, except in a subtype definition.

input

An input allows the consumption of the specified input signal instance. The consumption of the input signal makes the information conveyed by the signal available to the process. The variables associated with the input are assigned the values conveyed by the consumed signal.

The values will be assigned to the variables from left to right. If there is no variable associated with the input for a sort specified in the signal, the value of this sort is discarded. If there is no value associated with a sort specified in the signal, the corresponding variable becomes “undefined”.

The sender expression of the consuming process is given the PId value of the originating process, carried by the signal instance.

Z.100

local variables

Local variables of a procedure become parts of the procedure instance when the procedure is called, and they cease to exist when the procedure returns.

The local variables will get default initial values if nothing else is specified.

Z.100 (variable definition)

output

An output generates a signal of the specified signal type, containing the specified actual parameters, and send this signal instance to the specified destination.

Stating a <process identifier> in <destination> indicates the destination as any existing instance of the set of process instances indicated by <process identifier>. If there exist no instances, the signal is discarded.

If no signal route identifier is specified and no destination is specified, any process, for which there exists a communication path, may receive the signal.

If an expression in the list of actual parameters is omitted, no value is conveyed with the corresponding place of the signal instance, i.e. the corresponding place is “undefined”.

The PId value of the originating process is also conveyed by the signal instance.

TIMe Electronic Textbook v 4.0 © SINTEF Modified: 1999-07-16

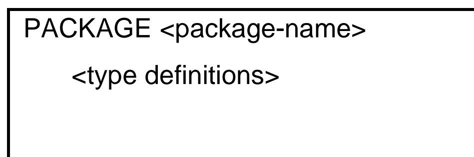
Z.100

package

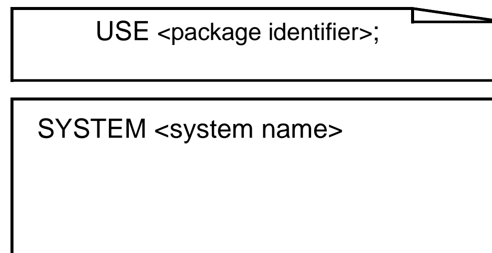
A package is a collection of types. A package is defined by a package diagram. Packages can be provided (that is defined) together with a system diagram (or together with another package diagram) or they can be used by means package identifiers.

A package may contain definitions of types, data generators, signal lists, remote specifications and synonyms. Definitions within a package are made visible to a system definition or other package definitions by a package-reference-clause (use clause). All (or selected) definitions of packages provided in this way will be visible in the system definition (or in the new package).

A package diagram has this form:



A package can be used either either in the definition of a new package, or as here, a system. This is done by the use clause.



Z.100

package reference clause

A package reference clause specifies that a system diagram or package diagram use the definitions of other packages. The names following the “/” after the package name denotes the subset of the definitions that are used.

Z.100

page numbering

A diagram may be split into a number of pages. In that case each page is numbered in the rightmost upper corner of the frame symbol. The page numbering consists of the page number followed by (an optional) total number of pages enclosed by (), e.g. 1 (4), 2 (4), 3 (4), 4 (4).

procedure

Procedures define patterns of behaviour that processes/services may execute at several places or several times during their life-time. The behaviour of a procedure is defined in the same way as for processes (that is by means of states and transitions), a procedure may have (local) variables, and in addition it may have IN, OUT, IN/OUT parameters.

Procedures are defined by procedure diagrams.

Z.100

procedure call

A procedure call transfers the interpretation to the procedure definition referenced in the call, and that procedure graph is interpreted.

The interpretation of the transition containing the procedure call continues when the interpretation of the called procedure is finished.

The actual parameter expressions are interpreted in the order given.

If an <expression> in <actual parameters> is omitted, the corresponding formal parameter has no value associated, i.e. it is "undefined".

Z.100

procedure heading

The procedure-heading of a procedure diagram has this format:

<procedure heading> ::=

[<virtuality>] [<export-as>] **procedure** <procedure-name>
[<virtuality-constraint>] [<specialisation>]
[<procedure-formal-parameters>]
[<result>]

<procedure-formal-parameters> defines the formal parameters of the procedure and have the format:

<procedure-formal-parameters> ::=
fpar [in`/`out | **in**] <typed-parameters>
{, [in`/`out | **in**] <typed-parameters> }*

where <typed-parameters> have the format

<typed-parameters> ::
<variable-name> {`,` <variable-name>}* <data-type-identifier>

<typed-parameters> is a list of parameter names followed by a data type name.

<result> has the format:

<result> ::= **returns** [<variable-name>] <data-type-identifier>

TIMe Electronic Textbook v 4.0 © SINTEF Modified: 1999-07-16

where <data-type-identifier> gives the data type of the value returned by the procedure. The optional <variable-name> can be used to name the result. The result can either be stated as an expression next to the return symbol, or as an assignment in a task to the variable introduced in result.

procedure reference

A procedure reference specifies that there is a procedure in the enclosing entity and that the properties of this procedure are defined in a separate (referenced) procedure diagram outside this diagram.

Z.100

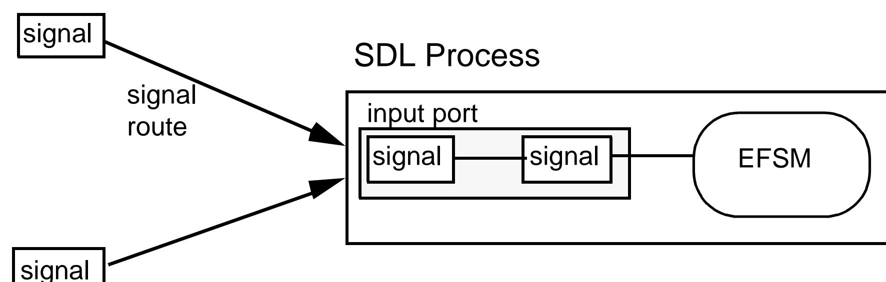
process

A process instance is part of a process set, which in turn is part of a block.

The properties of a process is either defined by a process diagram or it is defined by a process type diagram.

Each process consists of the input port and an extended finite state machine (EFSM) with a sequential behaviour defined by a process graph, which is a sort of state transition diagram. The finite state machine fetches signals from the input port in strict FIFO order except when the order is modified by the save operator (see below). For each signal it performs one transition which will take a short but undefined time.

Signals are messages that the finite state machine consumes. Each signal has a signal type identification which the FSM uses to select the next transition action. In addition, the signal carries the sender identity and possibly some additional data.



An SDL process with signal instances in the input port

process diagram

A process diagram defines the properties of a process set, where each of the process instances in the set have the specified properties.

The behaviour of processes may be defined either by means of a procedure graph (states and transitions) or by means of a substructure of services connected by signal routes. The behaviour of each of the services is defined by means of states and transitions.

Z.100

process type

A process type defines the common properties of a category of process instances. A process type is defined by a process type diagram.

process type diagram

A process type diagram defines the properties of a process type.

Z.100

process (type) heading

The heading of process diagrams (defining a process set directly without any process type) is a <process heading>, defining the name of the process set and the initial/maximum number of instances in the set.

The heading of process type diagrams is a <process type heading>, defining the name of the process type, its virtuality (and constraint), its formal context parameters and if it inherits from another process type.

Formal parameters are variables of the process instances. They get values as part of the creation of the process instance.

When a system is created, the initial processes are created in arbitrary order. The formal parameters of these initial processes have no associated values; i.e. they are undefined.

If the initial number is omitted, then the (default) value is 1. If the maximum number is omitted, then there is no limit on the number of instances.

Z.100

process (reference)

A process reference specifies that there is a process in the enclosing block and that the properties of this process are defined in a separate (referenced) process diagram outside this diagram.

Z.100

process set

A process set defines a set of processes according to a process type.

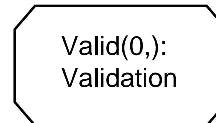
Just like we have the distinction between block reference, block type and block set according to type, we have the distinction between process reference, process type and process set according to a type. Our recommendation is that process sets should be described with reference to a process type.

TIMe Electronic Textbook v 4.0 © SINTEF Modified: 1999-07-16

Process reference:
Process set without
any associated type.

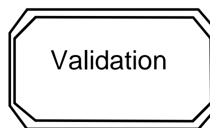
This is both a specification of a process set as part of the enclosing block and a reference to the corresponding process diagram, which defines the properties of the processes in the set.

Process set according
to a process type
(Validation)



The numbers in parentheses after the process set name specify the number of instances in the process set. As defined in above, there are initially no processes, and there is no limit on the number of instances that may be created.

A process set according to a type requires that the corresponding process type is defined:



Z.100

redefined process type

is a redefinition of the corresponding virtual process type in the super block type, and it is virtual, so that it can be redefined in further subtypes of this block type.

A redefinition of the process type must be a subtype of the type identified in the virtuality constraint.

Z.100 (virtual types)

remote procedures

The remote procedure mechanism consists of four interdependent language constructs:

1. *The exporting of a procedure*. A procedure which is made visible by other processes is marked with the keyword **exported** preceding the procedure heading, e.g. “**exported procedure** Validate ...” from a process within the CentralUnit. The exporting process can control in which states it will accept the remote request. It may also specify to save the request to other states. The controlling of the acceptance is done by using input and save symbols with the remote procedure name preceded by the keyword **procedure**.

2. *The importing of a procedure*. When a process, service or procedure wants to import a remote procedure, it must specify the signature of this procedure in an “imported procedure specification” in a text area. The specification in our case would read: “**imported procedure** Validate; **returns** integer;” where the integer returned would give the result of the validation.
3. *The specification of remote procedure*. In SDL all names must be defined in a specific scope. Thus, the names of remote procedures must be defined in the context in which the actual definition of the procedure and the calls will be contained. In our case the definition of the procedure Validate is within the CentralUnit and the call is in Controller of the AccessPoint. The scope unit enclosing all these is the system itself. There we will find a text area with the following text: “**remote procedure** Validate; **returns** integer;”.
4. *The calling of a remote procedure*. The calling of the remote procedure is indistinguishable from local procedure calls unless the caller explicitly states which process it will request the procedure executed by. This can be done by a **to**-clause with a PId following the procedure name of the call.

Remote procedures may be value returning (as in our example above) and they may be virtual. Z.100

return

A return represents the the completion of a call of a procedure.

A return is interpreted in the following way:

a)All variables created by the interpretation of the procedure start will cease to exist.

b)The interpretation of the procedure-graph is completed and the procedure instance ceases to exist.

c)Hereafter the calling process, service (or procedure) interpretation continues at the node following the call.

Z.100

save

A save specifies that the signals in the save symbol are retained in the input port in the order of their arrival.

The effect of the save is valid only for the state to which the save is attached. In the following state, signal instances that have been “saved” are treated as normal signal instances.

Asterisk save implies that all signals are retained in the input port.

Z.100

scope units

The following kinds of definitions form scope units:

- package
- system type
- system
- block
- block type
- block substructure
- channel substructure
- process
- process type
- service
- service type
- procedure
- signal
- operator, and
- type.

service

A service is a state machine being part of a process instance, and cannot be addressed as a separate objects. It shares the input port and the expressions self, parent, offspring and sender of the process instance.

Only one service at a time is executing a transition. Services alternate based on signals in the input port of the process.

Z.100

service (reference)

A service symbol specificies that a service is part of the containing process (type), and that the definition of the service can be found in a separate service diagram.

Process behaviour by means of services is an alternative to process behaviour by means of a process graph through a set of services. Each service may cover a partial behaviour of the process.

Z.100

service (type) heading

The heading of service diagrams is:

<service-heading> ::= service [<qualifier>] <service-name>

while service type diagrams have the following heading:

<service-type-heading>::=
[<virtuality>]
service type [<qualifier>] <service-type-name>
[<formal-context-parameters>]
[<virtuality-constraint>][<specialisation>]

signal definition

A signal definition defines a set of types of signals. Signal definitions are part of text symbols.

Signals may be defined in system and block diagrams, and these may then be used for communication between the blocks of the system or the processes of the block. Signals may also be defined in process (type) diagrams, but then they can only be used for communication between processes of the same set. Often signal definitions are collected in packages.

Z.100

signal list

Associated with each arrowhead of channels and signal routes or signal lists, that specifies the allowed signals in that direction.

Signallists are defined in text symbols.

Z.100

signal route

A signal route represents a communication path between process sets and between process sets and the environment of the enclosing block/block type.

Z.100

specialisation

A type may be defined as a specialisation of another type. This is done by the following construct:

<specialisation>::= **inherits** <type-expression> [**adding**]

Specialisation applies to system, block, process, service, data types, and to signals and procedures, and the same semantics apply in all cases:

TIMe Electronic Textbook v 4.0 © SINTEF Modified: 1999-07-16

- All definitions of the supertype are inherited:
- The formal context parameters of a subtype are the unbound, formal context parameters of the supertype definition followed by the formal context parameters added in the <specialisation>.
- The formal parameters of a specialised process type or procedure are the \formal parameters of the process supertype or procedure followed by the \formal parameters added in the <specialisation>.
- The complete valid input signal set of a specialised type is the union of the complete valid input signal set of the<specialisation> and the complete valid input signal set of the supertype.
- A specialised signal definition may add (append) data type identifiers to the \data type list of the supertype.
- A specialised partial type definition may add properties in terms of operators, literals, axioms, operators and default assignment.
- Definitions and transitions (where appropriate) may be added in subtypes.
- Virtual \transitions and types in the supertype may be redefined in the subtype, but for virtual types only to subtypes of their constraint.

A virtual type or procedure is defined by prefixing the keyword of the diagram (e.g. **process** or **procedure**) by one of the keywords **virtual**, **redefined** and **finalized**.

(more)

virtual is used when a type is introduced as a virtual type. A virtual type must be a type defined locally to another type; the implication is that it can be redefined in types that inherit from the enclosing type.**redefined** is used when the redefinition of a virtual type is still virtual. **finalized** is used when the redefinition is not virtual.

Z.100

start

There is only one start symbol for a process. The transition from the start 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.

Z.100

state

A state represents a particular condition in which a process may consume a signal resulting in a transition. If the state has neither spontaneous transitions nor continuous signals, and there are no signal instances in the input port, otherwise than those mentioned in a save, then the process waits in the state until a signal instance is received.

Z.100

system

A system is a set of blocks, block sets and channels. Blocks and block sets are connected with each other or with the environment of the system by means of channels.

Z.100

system (type) heading

The heading of system diagrams, that is a system-heading is as follows:

<system-heading> ::= **system** <system-name>

while system type diagrams have system-type-headings:

<system-type-heading>::=
system type [<qualifier>] <system-type-name>
[<formal-context-parameters>]
[<specialisation>]

As indicated in the syntax rule above, a system type can have formal context parameters and it can be a specialisation (of a more general system type).

task

A task may contain a sequence of <assignment statement>s or <informal text>. The <assignment statement>s or <informal text>s are executed in the specified order.

A task is part of a transition.

Z.100

transition

A transition performs a sequence of actions. During a transition, the data of a process may be manipulated and signals may be output.

Actions may be:

- task,
- output,
- set,
- reset,
- export,
- create request,
- procedure call, or
- remote procedure call

The transition will end with the process entering a:

- next state,

- with a stop,
- with a return or
- with the transfer of control to another transition.

Z.100

text symbol

Text symbols are used in order to have textual specifications as part of diagrams, especially for specification of signal types, data types and variables.

There is no limit to the number of text symbols that may occur in a diagram. Text symbols are not connected to other symbols by flow lines.

The text symbol is also used for the graphical representation of a use clause, see package.

Z.100

timer

The notion of timers provides a mechanism for specifying time-related matters. Timers are just like alarm clocks. The process waiting for a timer is passively waiting since the process needs not sample them. Timers will issue time-out signals when their time is reached. There may well be several different timers active at the same time. Active timers do not affect the behaviour of the process until the timer signal is consumed by the process.

A timer is declared similarly to a variable.

```
TIMER door_timeout ;
```

Timers are **set** and **reset** in tasks. When a timer has not been **set**, it is inactive. When it is **set**, it becomes active.

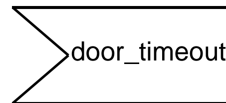
```
set
(now +10,
door_timeout)
```

A timer is set with a **time** value. **time** is a special data type and is mainly used in connection with timers. The expression “**now**+10” is a **time** value and it adds the **time** expression **now** and the duration 10 (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. You may add or subtract duration to **time** and get **time**. You may divide or multiply duration by a real and get duration. You may subtract a **time** value from another **time** value and get duration.

(more...)

The semantics of timers is this: a time value is **set** in a timer and it becomes active. When the time is reached, a signal with the same name as the timer itself will be sent to the process itself. Then the timer becomes inactive.

The timer signal can be input in the same way as ordinary signals:



A timer may be **reset**. It then becomes inactive and no signal will be issued. (If an inactive timer is **reset**, then it remains inactive.) A **reset** will also remove a timer signal instance already in the input port. This happens when the timer has expired, but the time-out signal has not been consumed.

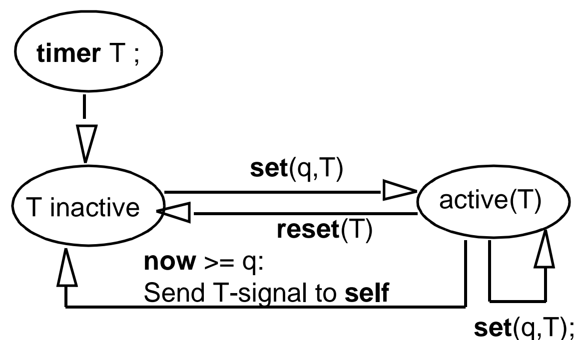
If an active Timer is **set**, the **time** value associated with the timer receives a new value. The timer is still active. If a timer is **set** to a **time** which is already passed, the timer will immediately issue the time-out signal.

There is an operator **active** which has a timer as a parameter and which returns a Boolean that can be used to check whether a certain timer is active or not.

Timer signals may contain data as other signals may contain data. Different parameter values in **set** means generation of several timer instances. **reset** must match these parameter values to eliminate the correct timer instance.

(more...)

The following is a sketch of a finite state machine of the behaviour of a timer.



Z.100

variable definition

Variables can be defined in processes, services and procedures.

Variables of process are created as part of the creation of the process instance.

Variables of services are created when the service is created as part of the creation of the containing process instance.

Local variables of a procedure become parts of the procedure instance when the procedure is called, and they cease to exist when the procedure returns.

TIMe Electronic Textbook v 4.0 © SINTEF Modified: 1999-07-16

Variables will get default initial values if nothing else is specified.

Z.100

virtual process type

A virtual process type is a process type that can be redefined in a subtype of the enclosing block type.

The virtuality is specified in the process type heading or by <virtuality> in the corresponding process type reference symbol.

A redefinition of the process type must be a subtype of the type identified in the virtuality constraint.

Z.100 (virtual types)

virtuality

The virtuality of a type defines whether the type is virtual (so that it can be redefined in a subtype of the enclosing type), redefined (a redefined type, but still virtual), or finalised, that a redefinition that cannot be further redefined.

<virtuality>::= **virtual** | **redefined** | **finalized**

- **virtual** is used when a type is introduced as a virtual type. A virtual type must be a type defined locally to another type; the implication is that it can be redefined in types that inherit from the enclosing type.
- **redefined** is used when the redefinition of a virtual type is still virtual.
- **finalized** is used when the redefinition is not virtual.

virtuality constraint

A constraint on a virtual type has the form of a virtuality\-constraint:

<virtuality-constraint> ::= **atleast** <identifier>

where <identifier> identifies a type (which is called the constraint type) of the appropriate kind (block, process, service or procedure).

The implication of a constraint is that a redefined or finalized definition of the virtual type must be a type definition that inherits from the constraint type. In case of no constraint specified, the definition of the virtual type itself is the constraint.

virtual (input) transition

A virtual input transition is a special case of a general notion of virtual transition (virtual priority input, virtual start, virtual spontaneous transition). In addition SDL has virtual save.

Redefinition of virtual transitions/saves corresponds closely to redefinition of virtual types.

- A virtual start transition can be redefined to a new start transition.
- A virtual priority input or input transition can be redefined to a new priority input or input transition or to a save.
- A virtual save can be redefined to a priority input, an input transition or a save.
- A virtual spontaneous transition can be redefined to a new spontaneous transition.

Z.100

TIMe Electronic Textbook v 4.0 © SINTEF Modified: 1999-07-16