Chapter

1

Object Oriented Design Using SDL

This methodology chapter will take you into the world of object oriented SDL, as introduced in the 1992 version of the language. It will follow one case (a simple Access Control system) from the specification to the final SDL design. A simple OO analysis (according to the SOMT method) is performed, followed by an object oriented design using SDL.

The object oriented SDL concepts are introduced step by step by developing different versions of the Access Control system. The first version will make use of the OO concepts block types and process types only. The final version will use more advanced OO concepts, such as inheritance (specialization), virtual types and type libraries (package diagrams).

Note that this chapter does not deal with all parts of the SOMT method described in the SOMT Methodology Guidelines starting in chapter 69 in the User's Manual; it mainly focuses on the usage of object oriented SDL in the design activities of SOMT.

Requirements on the Access Control System

This section should only be viewed as a background for the design of the system and not as a description of a complete requirements analysis phase.

Description of the System to be Built

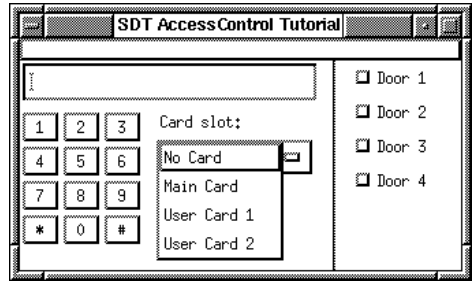
This application is chosen because it is a good example of an embedded system, with features that make it very suitable to be specified using SDL and the object oriented extensions (introduced in the 1992 version of the language).

The Access Control system is a system to control the access to a building. To enter the building, a user must have a registered card and a personal code (four digits). The device used for entering the card and personal code consists of a card reader, a keypad and a display.

The main characteristics of the system are:

- Moderate real-time demands
- Mostly signal oriented
- Simple data representation
- Simple interface to the environment (hardware)
- A non-distributed system
- Adding new features to the system can be achieved in an easy way by adding new program logic, while the interface to the environment remains the same
- The system can be simulated in the host environment by using a graphical user interface (see Figure 1).

On UNIX:



In Windows:

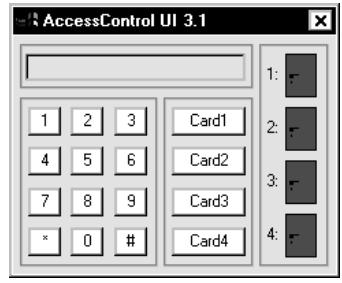


Figure 1: Graphical interface to the Access Control system

Textual Requirements

This description serves as an initial set of requirements. These requirements are normally collected and refined to a standardized form to make the requirements analysis easier to deal with and each requirement easier to refer to. Only the initial set of requirements will be shown for this simple example.

We will also focus only on the *functional requirements* and leave out the *non-functional requirements* (like performance, reliability, availability, etc.).

Basic Requirements

The hardware devices consists of the following components:

- An 8751 microcontroller
- 64 kilobytes of program memory (RAM or ROM)

- 64 kilobytes of data memory (RAM)
- A card reader for credit cards
  The card reader reads track 2. Data is stored as 40 five-bit words according to the most common standard.
- A keypad
  The keys are organized according to normal telephone standard. Valid keys are the digits 0-9. In the basic version, the function keys "*" and "#" are not recognized.
- A display unit
  The display unit can display 2 lines each consisting of 16 characters.
- 4 LEDs
  Four light emitting diodes will indicate the status of the controlled doors. Off = closed, on = open.

The system should be able to fulfill the following tasks for a user:

- Reading the code on the back of a standard credit card.
- Reading a personal code, consisting of 4 digits, typed from the keypad.
- Validate that the card and the personal code are registered.
- If the system is configured to control more than one door, give the user the possibility to choose which door to open after the card and code have been validated.

The system should be able to fulfill the following tasks for a system administrator/supervisor:

- Registration of a user card and a personal code. Only one code is allowed for each user card.
- Registration of the supervisor card at system startup time. Only one supervisor card is allowed for each system.

General requirements:

- The system must be designed in such a way that it is easy, at system generation time, to configure the system to handle from one to four doors.

Additional Requirements

The system should be able to fulfill the following tasks for a user:

- Displaying time.
- Displaying which category (see below) of card is valid in the current situation.

The system should be able to fulfill the following tasks for a system administrator/supervisor:

- Stopping the opening of one door (only the supervisor can open the door after this).
- Stopping the opening of all doors (only the supervisor can open a door after this).
- Removing the blocking of one or all the doors.
- Allowing free access through one or several doors.
- Specifying different categories of cards permitting different access possibilities during a 24-hour period
- Displaying the time.
- Setting of the current time.
- Blocking a user card.
- Remove the blocking of a user card.

Use Cases

The most interesting functional requirements are described by a number of use cases. These use cases describe the interaction between the system and its environment and formalizes (to some extent) the functional requirements.

The outside entities that communicate with the system are usually called the *actors* of the use cases. Actors are often

- human users
- other systems
- hardware

There are two different actors of the Access Control system that are relevant (the hardware is not taken into account in this simple example): *user* and *supervisor*.

- The user functions are the services available for all users, such as reading the card code, reading the four digit personal code, etc.
- The supervisor functions are only available for suitable privileged personnel (e.g. a supervisor) and perform services such as registration of a new card and code.

The use cases could be described either textually or by MSCs or by a combination of the two notations. An example of a use case with the user actor is the Open Door use case (described by the MSC OpenDoor in Figure 2). The use case ends with the fulfillment of the *goal* of the use case: the opening of the door.

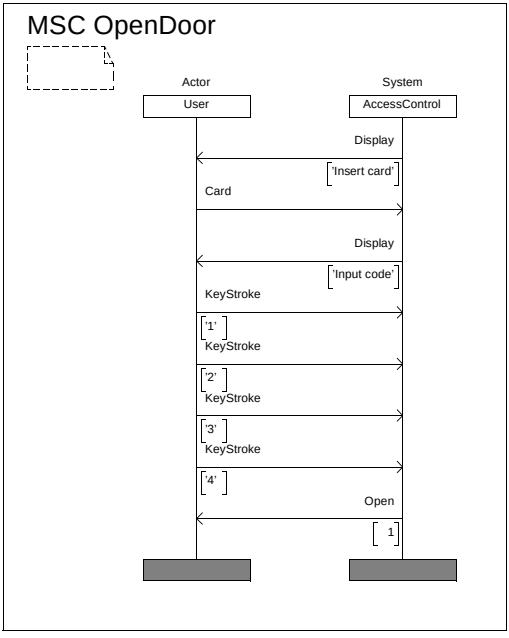


Figure 2: Requirements use case OpenDoor

Use cases that describe requirements usually show only the interaction between the actors and the system. When the use cases are refined in later activities, they can also express the inner behavior of the system.

Object Model

The requirements object model is a simple object model that relates the known domain entities of an access control system and its environment. The environment of the system could be anything that is related to the system as long as it *is relevant for understanding the problem*, typically the actors of the use cases that describe the wanted behavior of the system. The objective of the model is to give a simple picture of the problem without going into details.

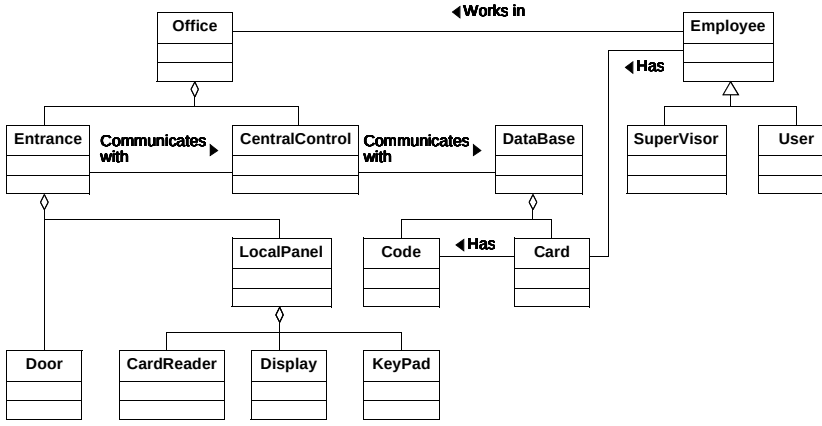


Figure 3: Requirements object model

When elaborating the requirements object model into an analysis object model, concern about the system properties rather than the real world properties will affect the model. In the requirements activity, it is not known what a certain class will result in or if it should be modeled at all. When analyzing the requirements and the system to be built, classes can be mapped to software entities, hardware entities or not mapped at all.

System Analysis of the Access Control System

The system analysis is based on the results after analyzing the requirements and the problem domain on a high level. The models in the system analysis focuses more on the internal structure of the system to be built, without taking design decisions (or at least as few as possible).

Analysis Object Model: Basic Version

The *inheritance* concept is not used in the basic version because the information that needs to be modeled has a very simple structure. What can be seen in Figure 4 is the *aggregation* and the *association* relations between the classes and the attributes and operations for the individual classes.

Compared to the requirements object model, the following changes have been made:

- The actors are removed from the object model to simplify the reading.
- Classes have been structured in a way that makes the mapping to an SDL design easier. Especially the aggregation hierarchy is designed with this in mind; the structure will basically be kept when making an SDL design.
- Classes from the requirements object model that are redundant (only introduced to increase the understanding of the problem) are removed.
- A difference between *active* and *passive* objects has been taken into account. The active objects have behavior while the passive objects only have data structure and data manipulation. The classes in the aggregation hierarchy are all active, while the classes in the information structure are passive.
- Attributes and operations have been added to the classes.
- The analysis object model is structured into three parts, each part showing a different view of the relationship between the classes: containment, communication and information.

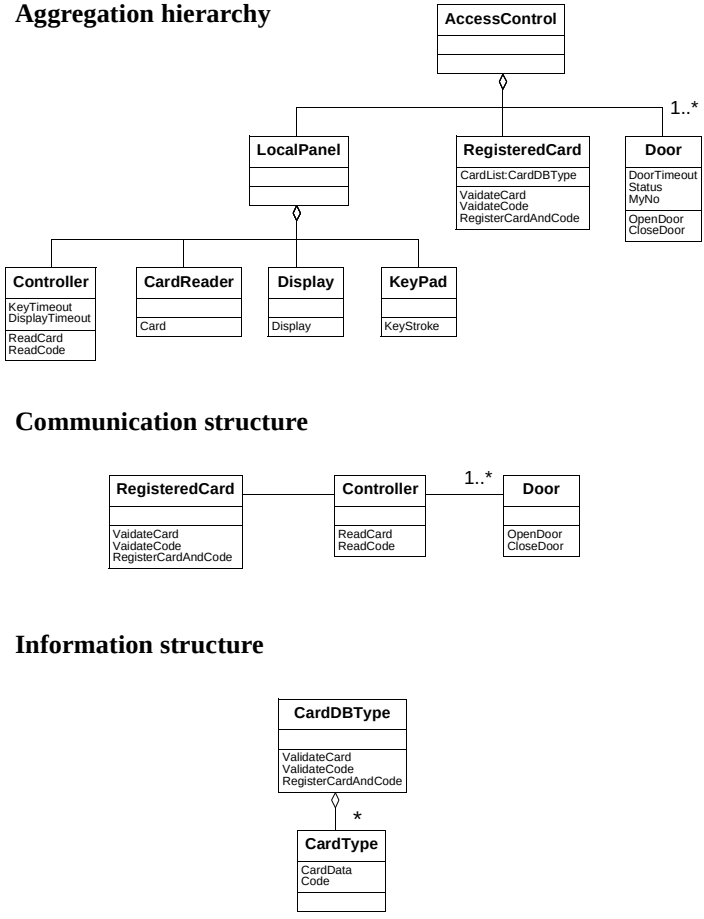


Figure 4: The analysis object model of the Access Control system (basic version)

The Analysis Use Case Model

The following MSC describes the use case for opening a door and is a part of the complete analysis use case model. The level of granularity can either be very detailed (each involved leaf object is represented by an MSC instance) or general (each subsystem of the aggregation hierarchy is described by an instance). This choice between readability and expressiveness is dependent of the application area and design customs. In this case, the subsystem representation was chosen (see Figure 2).

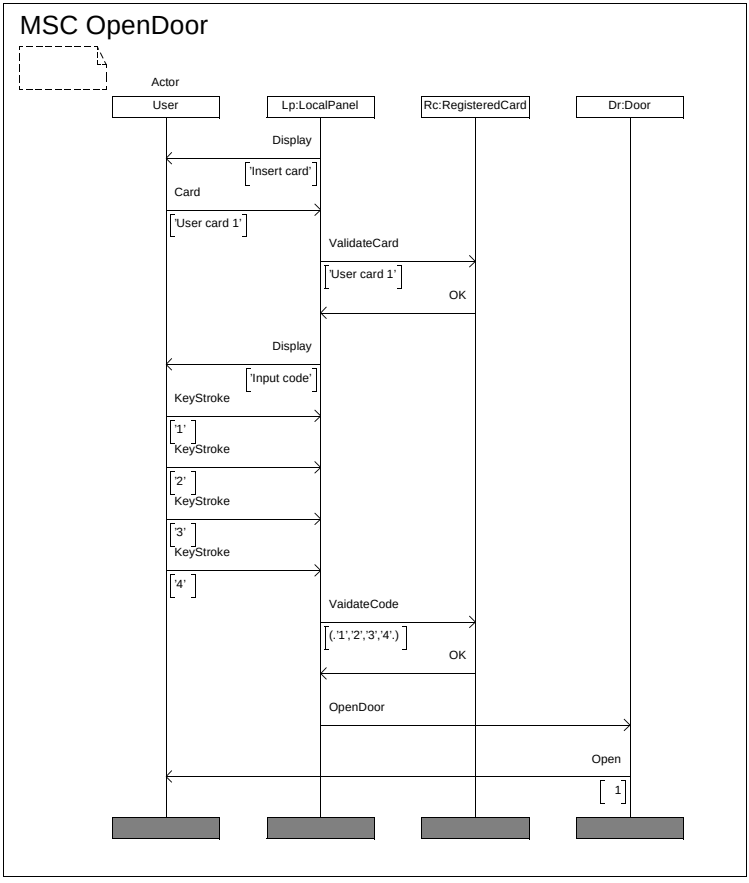


Figure 5: Analysis use case OpenDoor

Note that the MSC instances are, in fact, instances, i.e. they represent *objects*. To indicate this, the naming of the instances include both an object name and the correspondent class name.

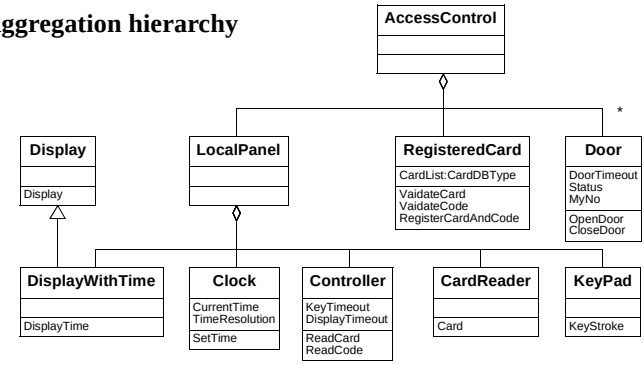
It should also be noticed that all MSC messages do not map strictly to class operations. In some cases, the operation is synchronous, that is demands a return message. This return message is also described in the MSC use case in Figure 2 (e.g. the operation *ValidateCard* is described by the messages *ValidateCard* and *OK*).

Analysis Object Model: Enhanced Version

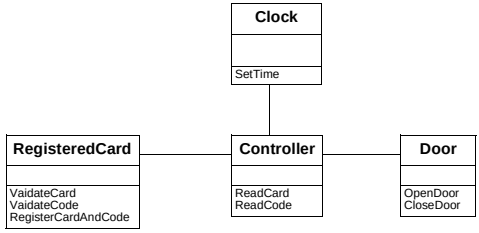
The following example is how an analysis of the additional requirement of time handling can be performed. The addition of a clock function will mainly add a new operation to the class Display (display of current time). A new class Clock must also be introduced. The properties of this class handle the clock and update the current time. Figure 6 describes the enhanced analysis object model for the Access Control system.

When adding behavior, the other models must of course also be extended, including the internal textual requirements and the use case models of the requirements and system analysis.

Aggregation hierarchy



Communication structure



Information structure

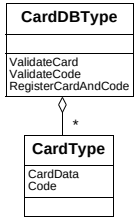


Figure 6: The analysis object model of the Access Control system (extended version with time handling)

Object Oriented Design of the Access Control System

System Design

The system design activity aims at producing a design architecture and to refine the use cases into use cases that could give a better help during the detailed design. Another purpose for refining the use cases is to make them suitable for verifying the design by means of the SDL Validator functionality *Verify MSC*.

Since the goal of this methodology handbook is to describe the object oriented features of SDL during the design, the description of a complete system design has been left out.

Object Design

We will now introduce the new SDL concepts step by step.

- In the first version of the Access Control system we will only use the new type concept for blocks and processes.
- In Version 2 we will make use of the new procedure concepts, such as remote procedures, value returning procedures and global procedures. We will also introduce the package concept, the specialization concept and the virtual concept.

Version 1: Block Types and Process Types

According to the analysis object model, the top class of the aggregation hierarchy has been mapped to an SDL system. The leaf nodes of the aggregation hierarchy have been mapped to processes and the classes between the top and the leave nodes have been SDL blocks. Note that even if there are no classes between the top class and a leave class in an aggregation chain, an SDL process still has to be contained in an SDL block.

Six processes (CardReader, Controller, Display, Door, KeyPad and RegisteredCard) have been identified from the analysis object model and three blocks (LocalPanel, Doors and RegisteredCard) have been created in order to preserve the structure described by the aggregation hierarchy of the analysis object model (see Figure 7).

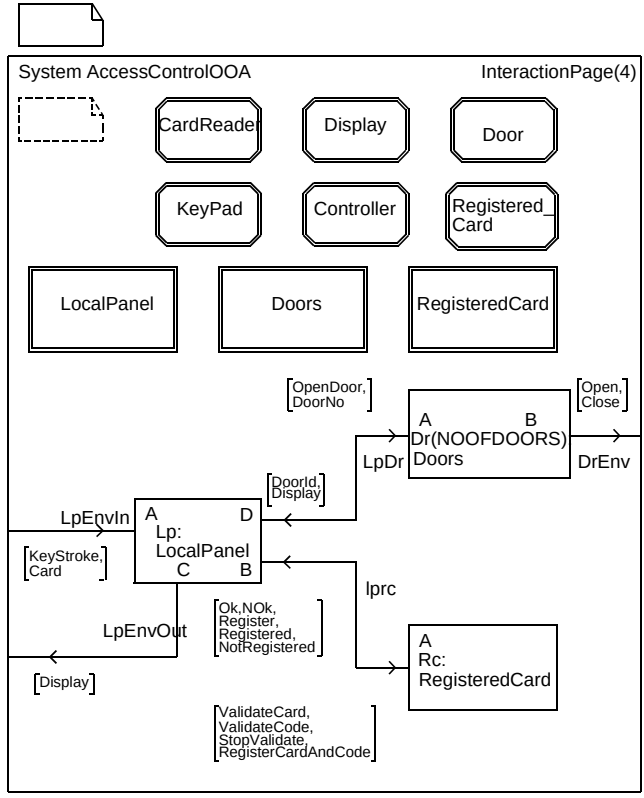


Figure 7: System diagram AccessControlOOA

Block Types and Process Types

A type definition can be placed anywhere in a system. For this example, the choice was to place them on the system level so that they will have maximum visibility. Normally, they would have been placed in separate packages to support parallel editing and analysis of the separate sub-systems (block types).

To place a type at a high level (in a system or package) means that they can be instantiated anywhere where they are visible, and also be used for specialization anywhere in the system.

Block Type LocalPanel

Even if the types are placed on the same level, the structure is kept by instantiating the types according to the analysis object model. This means that the instantiation of the process types CardReader, Display, KeyPad and Controller is made inside the block type LocalPanel (see Figure 8).

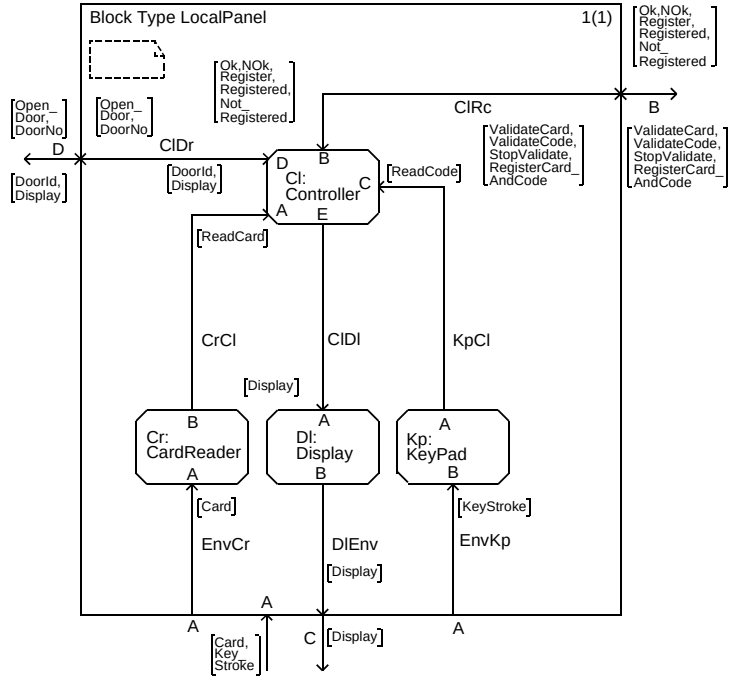


Figure 8: The block type LocalPanel

Block Type RegisteredCard

The block type RegisteredCard will only contain an instance of the process type RegisteredCard. It is perfectly legal in SDL to use the same name for a block type and a process type because they are of different entity classes (see Figure 9).

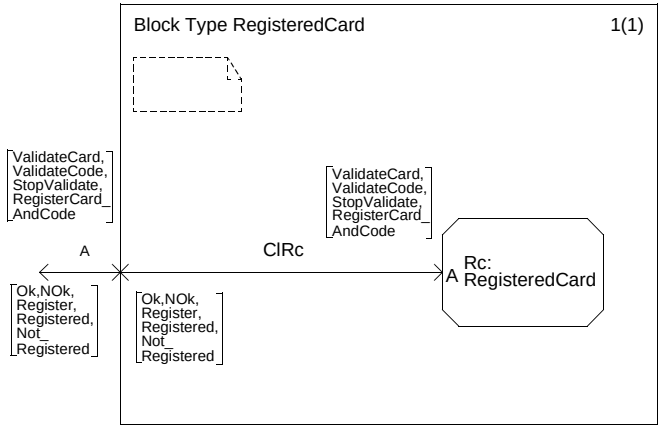


Figure 9: The block type RegisteredCard

The Classes CardDBType and CardType

As previously mentioned there are classes that will mainly contain data and data manipulation operations. The classes CardDBType and CardType are of this type and they are implemented in the design as abstract data types. The data type CardDbType has a number of operators defined to validate a card and a code, and to register a new card and a new code. These operators are implemented in-line, as “C” functions (see Figure 10).

```
NEWTYPE CardType
STRUCT
  CardData Charstring;
  Code     CodeArray;
ENDNEWTYPE CardType;
NEWTYPE CardDbType
 array(Index,CardType)
ADDING
LITERALS
 NewDb;
OPERATORS
 ValidateCard:Charstring,CardDbType->ValCardResType;
 ValidateCode:CardType,CardDbType->ValCodeResType;
 ListFull:CardDbType->Boolean;
 RegisterCardAndCode:CardType,CardDbType->CardDbType;
/*#ADT(B)
#BODY
#ifndef XNOPROTO
extern #(CardDbType) #(NewDb) (void)
#else
extern #(CardDbType) #(NewDb) ()
#endif
{
 return(yMake_#(CardDbType)(yMake_#(CardType)("V\0",
 yMake_#(CodeArray)('0'))));
}
......
.......
*/
ENDNEWTYPE;
```

Figure 10: The data types CardType and CardDbType

An instance of the type CardDbType is declared in the process type RegisteredCard. The operations ValidateCard, ValidateCode and RegisterCardAndCode for the class RegisteredCard are now implemented as operators for the data type CardDbType (see Figure 11).

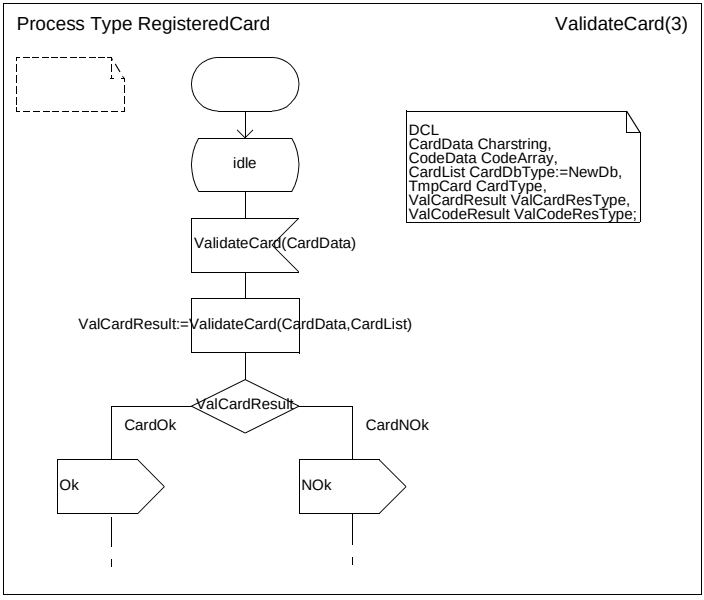


Figure 11: Call of operators inside the Process RegisteredCard

Block Type Doors and Process Type Door

A requirement for the Access Control system is that it should be able to control up to four doors. In our object oriented SDL design, this can be accomplished by creating a block set of the block type Doors. The Synonym NOOFDOORS is by default 1 but can be assigned any value between 1 and 4. Block type Doors consists of an instance of the process type Door. To follow the OO analysis, the process type Door will control how long a door should be opened (attribute DoorTimeOut) and also the opening and closing of a door (operations OpenDoor and CloseDoor).

Version 2: Procedures, Specialization and Packages

A general rule when designing an SDL process is to keep the transitions as short as possible. Using procedures is often the solution.

The Use of Procedures in Version 1

In the first version procedures are frequently used and we shall now take a look at two of them, namely RegisterCard and ReadCode. Both are declared and called by the Controller process.

Procedure RegisterCard

This procedure is called when a new card should be registered (user cards or the supervisor's credit card). The function of this procedure is as follows:

- First it calls the procedure ReadCode to read the four user digits in the user's code.
- In the case of a successful return from the ReadCode (ReadCodeResult=Successful), send a request for the registration of a new card (signal RegisterCardAndCode) to the process RegisteredCard.
- Wait for the result of the Registration (return signals Registered or NotRegistered) and return (see Figure 12).

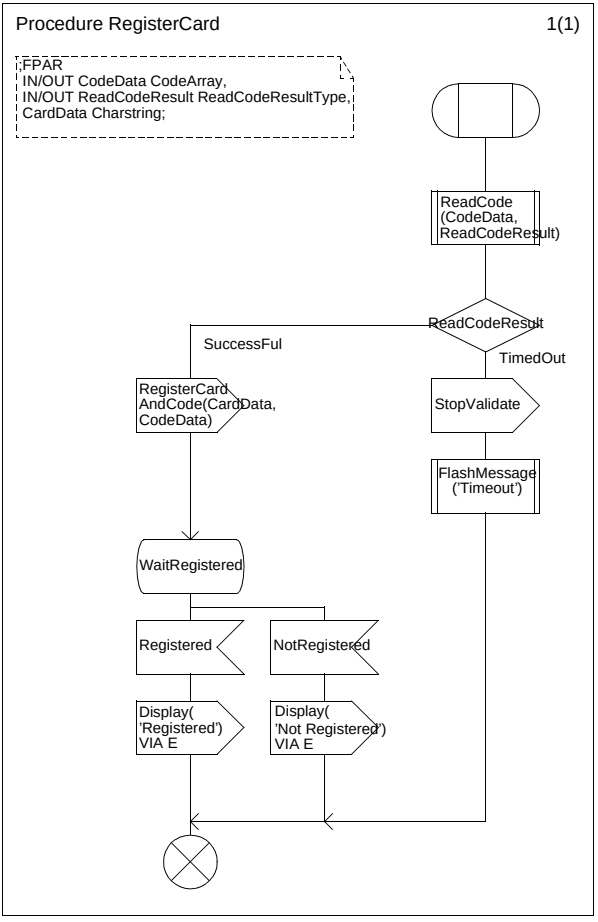


Figure 12: Procedure RegisterCard

Procedure ReadCode

This is a procedure to read four digits from the keypad. The digits read will be stored in an array named CodeData. If four digits are successfully received, the ReadCodeResult is assigned “Successful”, and a return to the calling process or procedure will take place.

This procedure is called both by the procedure RegisterCard and by the process Controller in the sequence of validating card and code (see Figure 13).

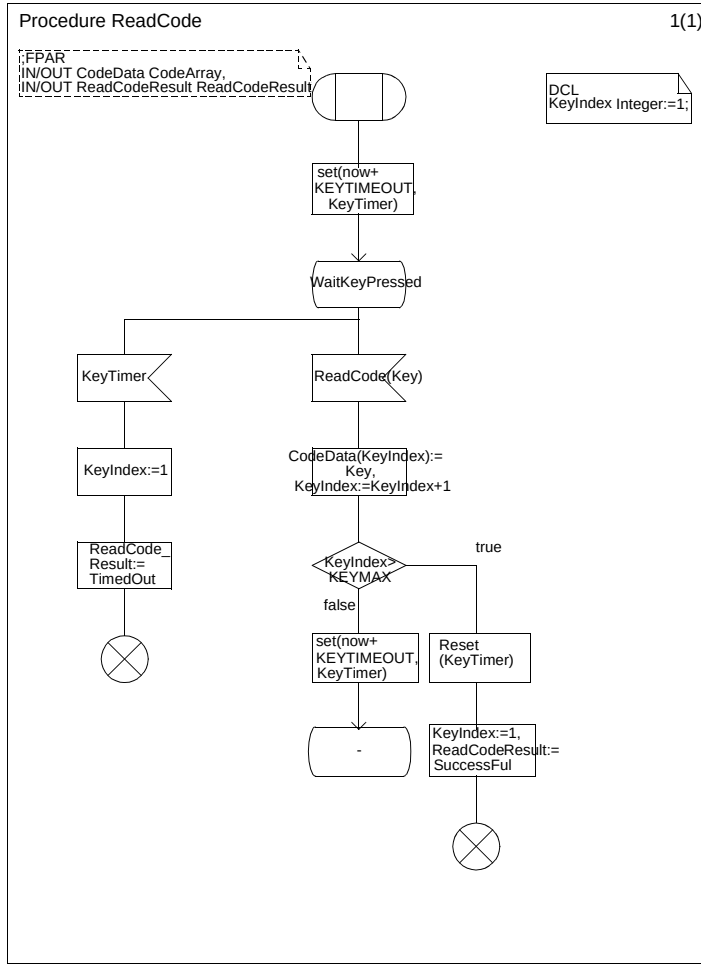


Figure 13: Procedure ReadCode

Remote Procedures and Value Returning Procedures

The idea in version 2 is to move the procedure RegisterCard from the process Controller to the process RegisterCard. There are two reasons for doing this:

1. This is the most natural place for it, because this procedure is called whenever a card is to be registered.
2. No signals have to be exchanged between process Controller and process RegisterCard to announce when to start and stop the registration procedure.

The procedure ReadCode must also be moved, because there will be a deadlock situation when the RegisterCard procedure calls the ReadCode procedure.

> **Note:**
>
> When a process calls a remote procedure, it enters a new implicit state where it will wait for an (implicit) return signal indicating that the procedure call has been executed. Any new signals, including calls to remote procedures, will be saved. This can easily lead to deadlock situations!

The ReadCode procedure can be placed in the KeyPad process and FlashMessage (another procedure also called by the RegisterCard) can be placed in the Display process.

Remote Procedures in SDL

Normally a procedure can only be called from the declaring process (or procedure) but by declaring it as EXPORTED it can be called from any process or procedure in the system. The remote procedure concept is modeled with an exchange of signals.

The Save Concept

The service process (the process with the EXPORTED procedure) can only handle a remote procedure call when it is in a state. If it is essential that a process is not interrupted with a remote procedure call in certain states, it is possible to use the SAVE symbol to save the call and handle it in a later state. This will mean that you cannot be sure that a remote procedure call will be handled directly. The model for the calling process is also that a new implicit state is introduced for each remote pro-

cedure call. The process will remain in this state until the remote procedure call is handled and executed.

How to Declare an Exported Procedure

1. Declare it as EXPORTED in the procedure heading.
2. Make an import procedure specification in each process/procedure that wants to call the remote procedure.
3. Introduce the name and signature (FPAR) of exported and imported procedures by making a remote procedure definition. This could be done in the system diagram, in a block diagram or inside a package. This declaration determines the visibility of the remote procedure by placing it in a certain scope.

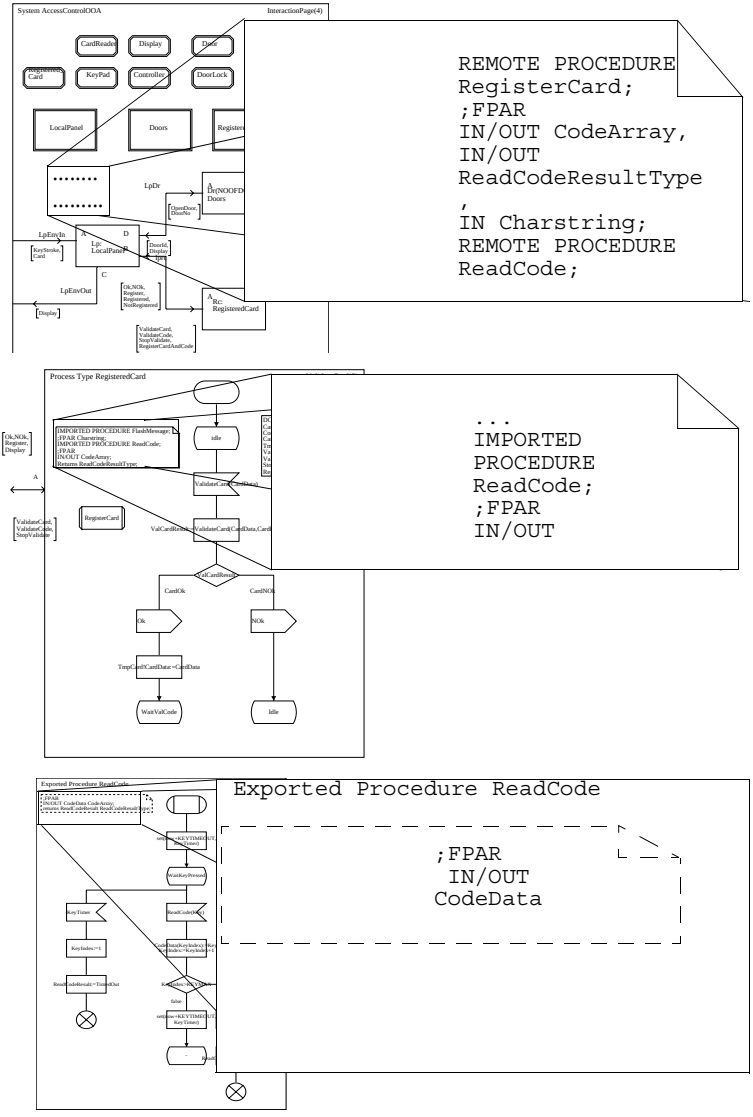


Figure 14: Declaration of a remote procedure

Value Returning Procedures in SDL

Any procedure can be called as a value returning procedure provided the last parameter is of IN/OUT type. The recommended way is to declare it as value returning if it is intended to be used as such. A call to a value returning procedure can be used directly in an expression, e.g. in an assignment (see Figure 15).

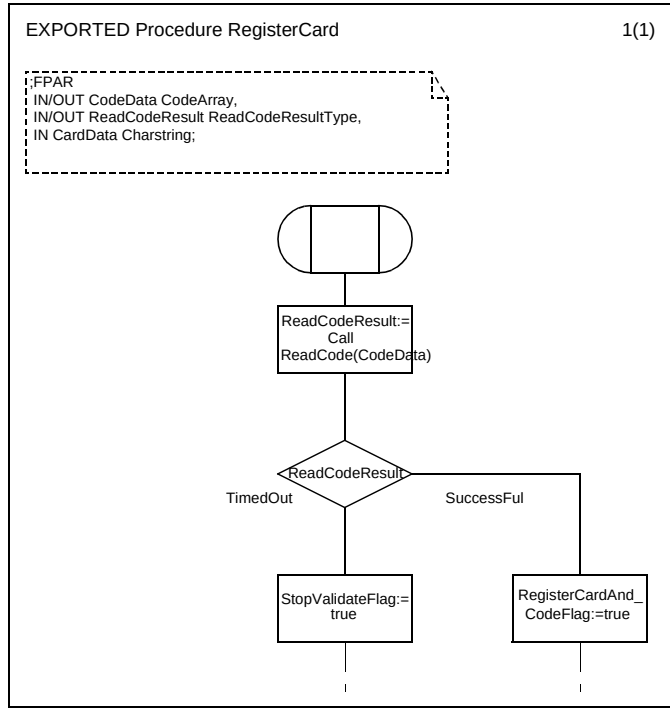


Figure 15: Use of value returning procedures

In version 2 we have declared the procedure ReadCode as a value returning procedure, and it will return a ReadCodeResultType value. We want to return this result from the procedure RegisterCard also, so we save it in a variable. (See Figure 15.)

Global Procedures in SDL

A procedure can also be defined globally in SDL. The conceptual model is that a local copy of the procedure is created in each process where it is called.

A Global Procedure for Sending a Signal

It is mostly the process Controller that displays messages. But the procedure RegisterCard (in process RegisteredCard) and the process Door also send messages.

The process Controller acts as an intermediate conveyer of the signal Display to the process Display in version 1. It is tempting to declare a global procedure that can send any message on the signal Display, and to call this procedure from process Display, process Door and the procedure RegisterCard. Unfortunately, this will not work. The reason is the above mentioned model with the creation of an implicit local model of the procedure. Calling the procedure from, for example, the process Door will in fact result in sending the signal from the calling process. Besides obscuring the signal sending, nothing will be gained by this; the signal must still be declared on the outgoing channel, etc. An alternative is of course to declare the procedure as an EXPORTED procedure and call it as a Remote procedure, but the remote procedure concept should be used with moderation and definitely not in this case, with the sole purpose of hiding signal sending.

When to Use the Different Kinds of Procedures

Local Procedures

- To keep the transitions short in order to highlight the signal interactions.
- To describe local routines.

Remote and Value Returning Procedures

- To make a local routine globally accessible.
- Use value returning procedures to simplify expressions.
- Use remote procedure calls instead of signals to access and manipulate data.

Global Procedures

- An alternative to macros.
- An alternative to a remote procedure if there is no natural “owner” of the procedure.

Specialization: Adding/Redefining Properties

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

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

- A subtype may add properties not defined in the supertype. One may, for example, add new transitions to a process type[1], add new processes to a block type, etc.
- A subtype may redefine virtual types and virtual transitions defined in the supertype. One may, for example, redefine the contents of a transition in a process type, redefine the contents/structure of a block type, etc.

Behavior (i.e. transitions) can be added to a process type using the adding mechanism. For example, the process type TimeDisplay (Figure 16) is a subtype of Display with the addition of a new transition. The keyword INHERITS defines the new type DisplayTime as a subtype of Display, stating that all definitions inside the process type Display is inherited by DisplayTime.

The gates A and B are dashed in order to indicate that they refer to the gate definitions in the process type Display, with the addition of the signal DisplayTime.

1. SDL differs from most other object oriented languages in the sense that SDL offers possibilities to specialize behavior specifications. In most other languages this is accomplished by redefining virtual methods in subclasses; in SDL this is easily accomplished by adding new transitions to a process type.

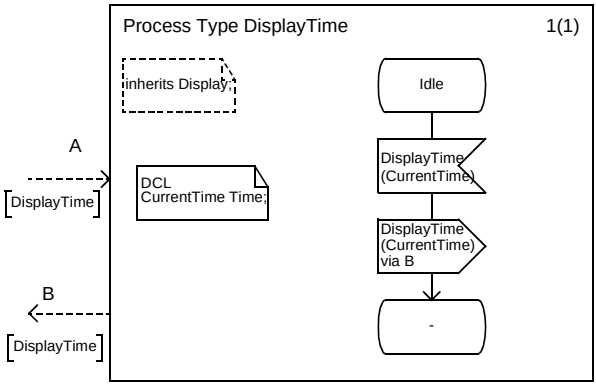


Figure 16: The process type TimeDisplay

In some cases it may be necessary not only to add properties, but also to redefine properties of a supertype. In Figure 16, the process type Door has to be redefined in order to send the signals OpenDoor and CloseDoor respectively to the new process DoorOpener. Therefore, the corresponding transitions of Door have to be defined as virtual transitions, as depicted in Figure 17.

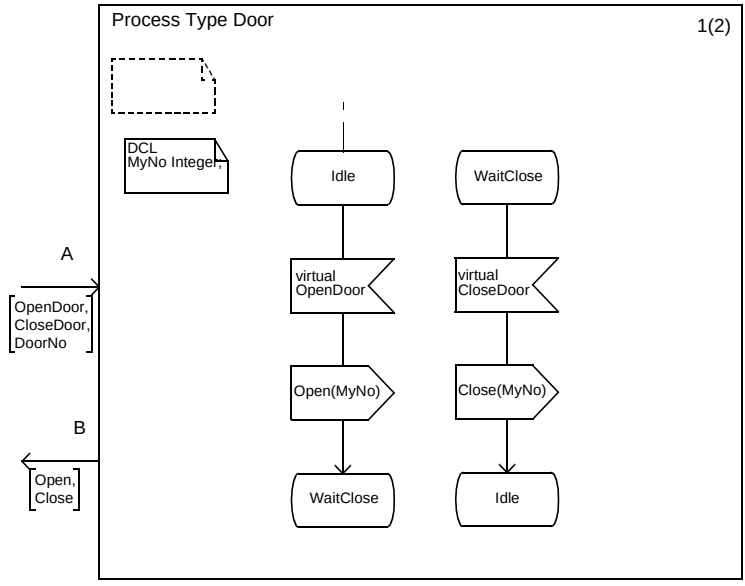


Figure 17: The process type Door with virtual transitions

Then, in the definition of the new block type SpecialDoor, the corresponding transitions of the process type Door are redefined as shown in Figure 18.

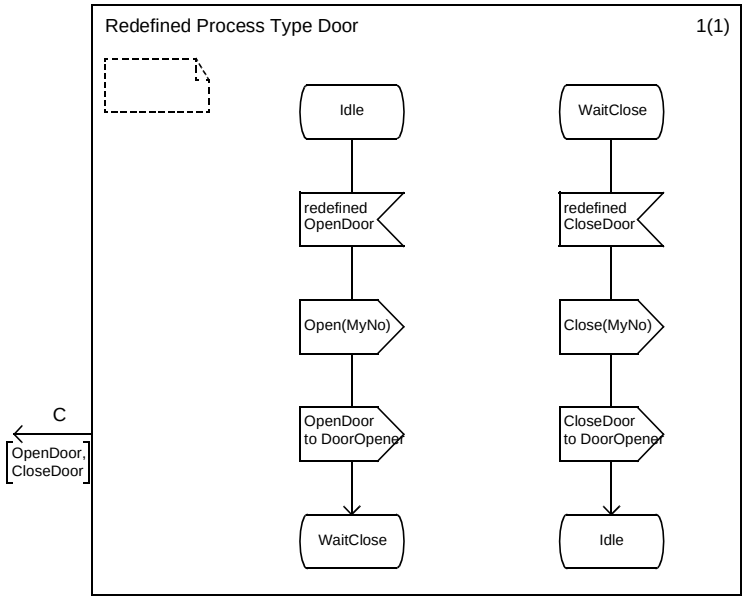


Figure 18: The redefined process type Door

In addition to virtual transitions, it is also possible to specify start transitions, saves, continuous signals, spontaneous transitions, priority inputs, remote procedure inputs and remote procedure saves as virtual. All of the above concepts have in common that they define *how* a transition should be initiated or *if* it should be initiated (*Save*). Furthermore, a virtual save can be redefined into an input transition or vice versa.

Example: Adding a Clock to the Access Control System

The Access Control system described in the previous sections can be extended to contain a clock which holds the current time. The time is displayed on the display in the format “HH:MM”, and the time can be set from the panel by first entering a “#” followed by the time in the format “HHMM”.

After an analysis of the problem, e.g. using OOA as described earlier, it is decided that the clock functionality is easiest realized by adding a clock to the local panel. Each minute the clock sends the current time to the controller, which displays the time on the display. Furthermore, the controller is extended to cope with the setting of the time from the key pad.

In order to apply the SDL concepts of specialization to this problem, the original access control specification has to be slightly modified. Since an access control system containing a clock can be regarded as a specialization of the original access control specification, it must be possible to inherit the properties of the original access control system when defining the new system. Therefore, it is necessary that the original Access Control system is defined as a system type (named BaseAccessControl), as depicted in Figure 19.

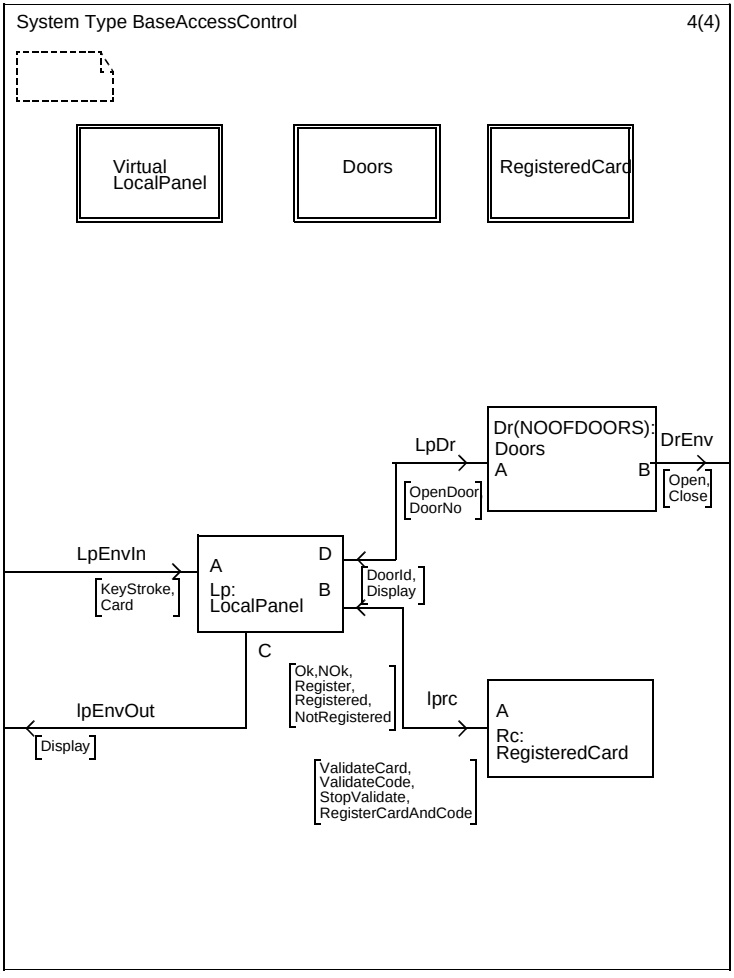


Figure 19: The system type BaseAccessControl

Since the specialization of BaseAccessControl requires changes to the block type LocalPanel, it is defined as virtual. For the same reason, the process types used in the block type LocalPanel (i.e. CardReader, Display, KeyPad and Controller) are all defined as virtual. Finally, for reasons of clarity, the definitions of process types that previously where

made on the system level are now made in the block types where they are used.

Now, a definition of the access control system containing a clock (named TimeAccessControl) can be based on the system type BaseAccessControl, as depicted in Figure 20.

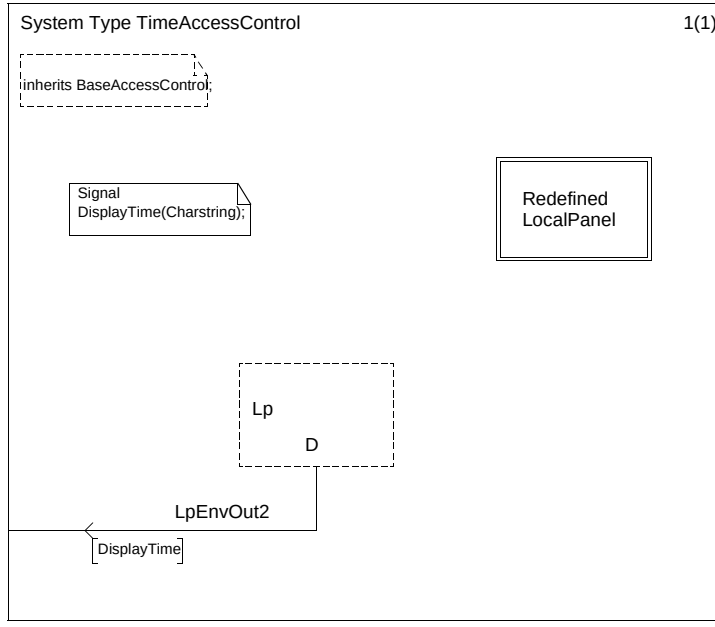


Figure 20: The system type TimeAccessControl

The system type TimeAccessControl inherits BaseAccessControl with the addition of a new signal DisplayTime, which is sent from the block Lp (of type LocalPanel) to the environment. Furthermore, the block type LocalPanel is redefined in TimeAccessControl; as depicted in Figure 21.

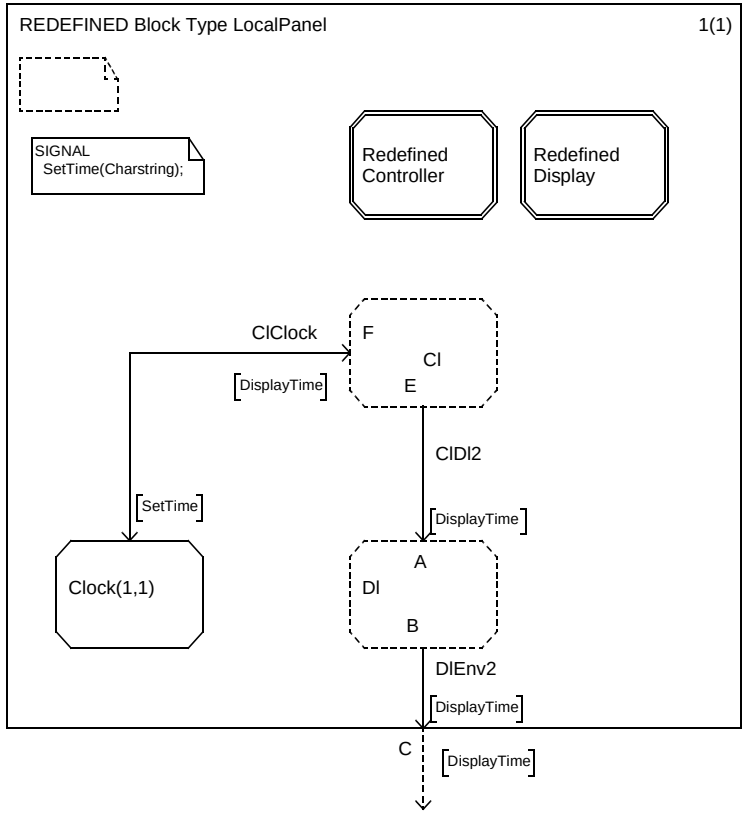


Figure 21: The redefined block type LocalPanel

LocalPanel is redefined to contain a process Clock which sends the signal DisplayTime to Cl (of type Controller) and receives the signal SetTime from Cl. Furthermore, Cl is extended to send the signal DisplayTime to Dl (of type Display), which in turn sends it on to the environment via gate C.

The redefinition of Display is straightforward. As depicted in Figure 22, a new transition for the signal DisplayTime is added.

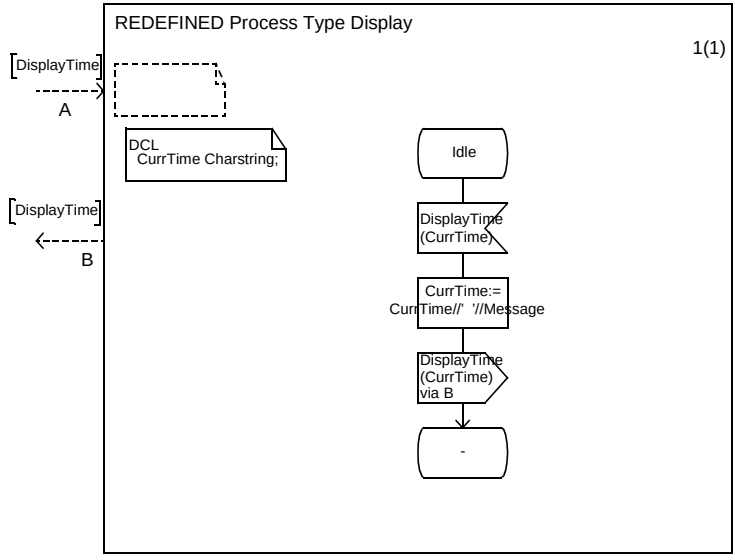


Figure 22: The redefined process type Display

The redefinition of Controller (Figure 23 on page 36) involves two issues: the addition of functionality to treat the signal DisplayTime, and the addition of functionality to read a new time from the KeyPad and correspondingly set the clock.

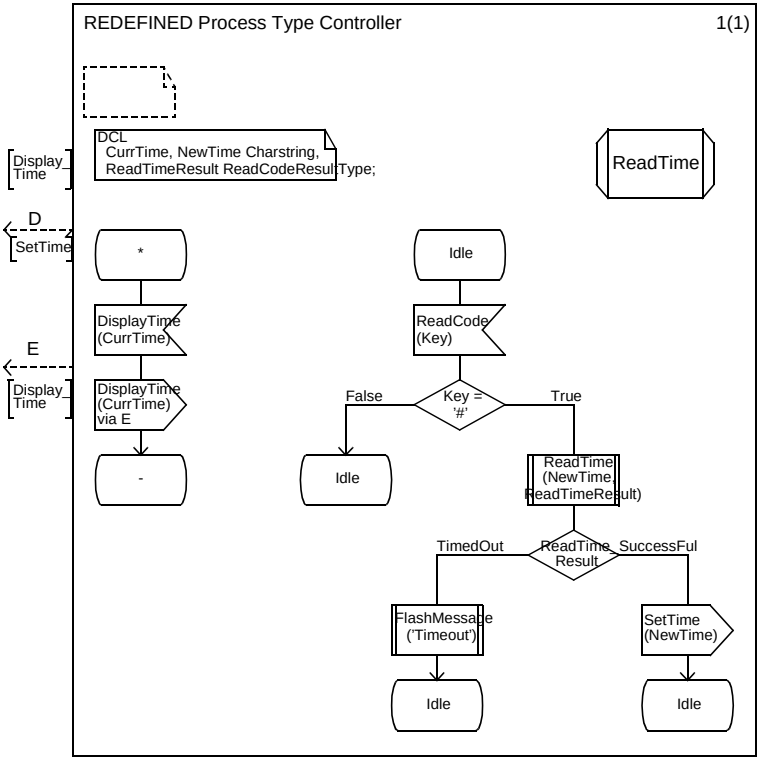


Figure 23: The redefined process type Controller

To cope with the signal DisplayTime, a transition is added to every state that, upon receipt of DisplayTime, sends it on to the process Dl (via gate E). Furthermore, a transition for the signal ReadCode is also added to state Idle in order to realize the setting of the clock. If the key pressed on the key pad is “#” then the new time is read (in the procedure ReadTime). If the new time was read successfully, then the signal SetTime is sent to the process Clock.

Finally, the process Clock is defined as depicted in Figure 24.

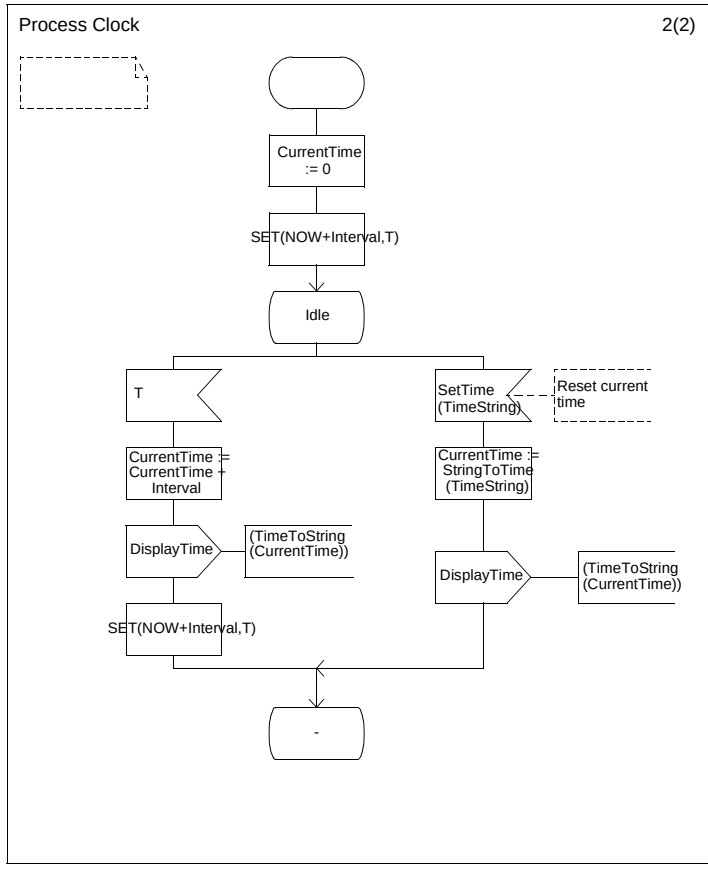


Figure 24: The process Clock

The variable CurrentTime of type Time, holds the current time in minutes. Every minute (duration 60), a timer expires which causes the variable CurrentTime to be incremented by 1, and the signal DisplayTime to be sent to process Cl. Receipt of the signal SetTime causes the variable CurrentTime to be updated with the new value. Since the time outside process Clock (i.e. the parameter of the signals DisplayTime and SetTime) is represented as a charstring, there is a need for functions converting Time to Charstring and vice versa. These functions can be defined in the following way:

```
NEWTYPE TimeOperators
  LITERALS Dummy;
  OPERATORS
    TimeToString : Time -> Charstring;
      /* Converts time to Charstring. The result
         is on the form 'HH:MM' */
      /*#OP (B) */

    StringToTime : Charstring -> Time;
      /* Converts Charstring to Time. Assumes that
         the Charstring is on the form 'HHMM'. */
      /*#OP (B) */
/*#ADT (B)
#BODY

SDL_Charstring #(TimeToString)(T)
SDL_Time T;
{
  SDL_Charstring result:=NULL;
  int Hours, Minutes;
  char tmp1[4], tmp2[4];

  Hours = (T.s/60/60)%24;
  Minutes = (T.s/60)%60;
  tmp1[0]='V';
  tmp2[0]='V';
  sprintf(&(tmp1[1]),"%2ld",Hours);
  sprintf(&(tmp2[1]), "%2ld",Minutes);
  xAss_SDL_Charstring(&result,tmp1,XASS);
xAss_SDL_Charstring(&result,xConcat_SDL_Charstring(r
esult,xMkString_SDL_Charstring(':')));
xAss_SDL_Charstring(&result,xConcat_SDL_Charstring(r
esult,tmp2));
  result[0]='V';
  if(Hours<10)
    result[1]='0';
  if(Minutes<10)
    result[4]='0';
  return result;
}

SDL_Time #(StringToTime)(C)
SDL_Charstring C;
{
  SDL_Time T;
  SDL_Charstring tmpstr;
  tmpstr=xSubString_SDL_Charstring(C,1,2);
  T.s = atoi(++tmpstr)*60*60;
  tmpstr=xSubString_SDL_Charstring(C,3,2);
  T.s = T.s + atoi(++tmpstr)*60;
  return T;
}
*/
ENDNEWTYPE;
```

Packages

The concept of packages enables a mechanism to handle a collection of different types. The different type definitions that are possible to define in a package are:

- Diagram types (system type, block type, process type, service type and procedure)
- Abstract data types and synonyms
- Signals and signal lists

The definitions in a package are included into the system (or into another package) by a USE clause.

The SDL Analyzer supports semantic analysis for packages. This means that large systems can be divided into several packages to enable a more easy handling of large projects.

An important thing to remember is that it must be possible to analyze a type where it is defined. This means that if a process type is placed in a package, the data types and signals that the process type uses must also be visible in the package. In Figure 25, the use of packages are exemplified.

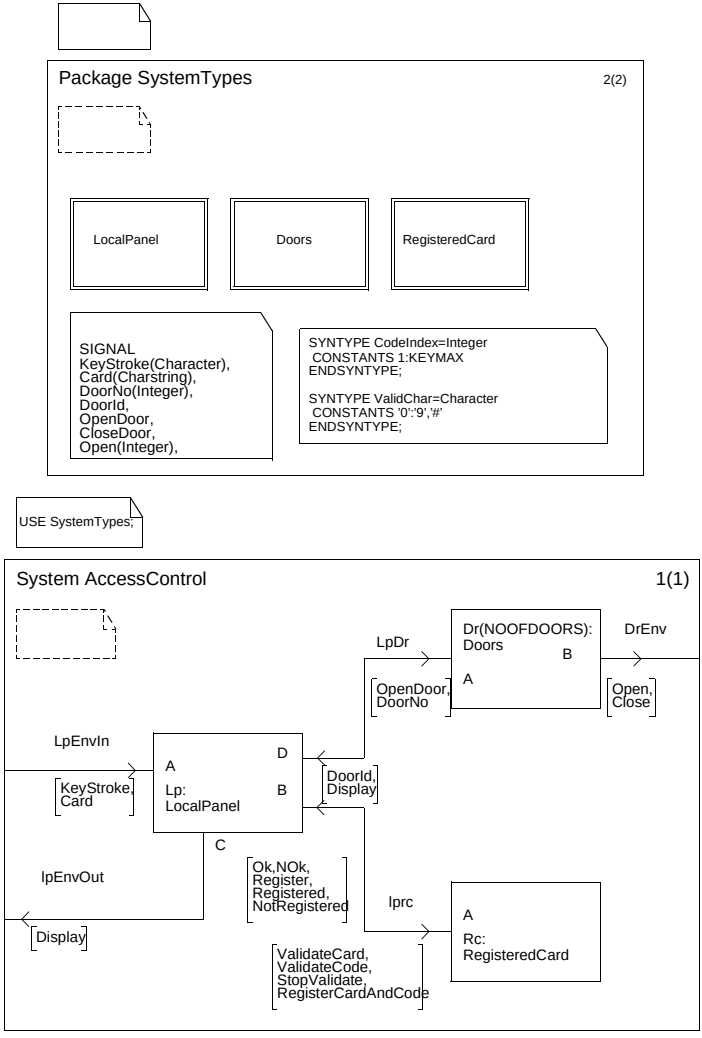


Figure 25: The use of packages in the Access Control system