Rational. SDL and TTCN Suite

IBM.

SDL Suite Methodology Guidelines

Methodology Guidelines

Introduction iv

1. Object Oriented Design Using SDL 1

Requirements on the Access Control System. 2
Description of the System to be Built. 2
Textual Requirements. 3
Use Cases 5
Object Model 7

System Analysis of the Access Control System. 8
Analysis Object Model: Basic Version 8
The Analysis Use Case Model 10
Analysis Object Model: Enhanced Version 11

Object Oriented Design of the Access Control System 13
System Design 13
Object Design 13
Version 1: Block Types and Process Types 13
Version 2: Procedures, Specialization and Packages 19
Specialization: Adding/Redefining Properties 27
Packages 39

2. Data Types 41

Introduction 42

Using SDL Data Types. 42
Predefined Sorts 44
User Defined Sorts 62
Literals 80
Operators 81
Default Value 82
Generators. 83

Using C/C++ in SDL 84
Introduction 84
Workflow 84
Import Specification 101
Accessing C/C++ Constructs not Fully Supported by CPP2SDL 102

C Specific Package ctypes 108
Different Int Types and Float 109
Charstar, Voidstar, Voidstarstar 109
The Carray Generator. 110
The Ref Generator 111

Using ASN.1 in SDL 116
Organizing ASN.1 Modules in the SDL Suite 116
Using ASN.1 Types in SDL 119
Sharing Data between SDL and TTCN 123

3. Using SDL Extensions 127

Own and ORef Generators 128
Introduction 128
Basic Properties of the Own Generator 128
Definition of Own Generator 130
The ORef Generator 132
Run-Time Errors. 133
Implicit Type Conversions 134

Algorithms in SDL 137
Compound Statement 138
Local Variables. 139
Statements. 139
Grammar for the Algorithmic Extensions 146
Algorithms in SDL Simulator/SDL Explorer. 147
Execution Performance in Applications. 148

4. Organizing a Project 149

Introduction. 150
General 150
Improved Support 151

Diagram Binding 152
Automatic Binding 152
Manual Binding 152
Source and Target Directories 153

How to Manage the Diagrams in a Project. 155
Starting to Use RCS Together with an SDL System 156
Using the SDL Suite and RCS in a Multi User Environment 158
Make Local Changes Global Using RCS 159
Make Global Changes Local Using RCS 160
Building and Populating a Work Area from RCS based Original Area 160
Endpoint Handling with RCS 161
Simultaneous Editing of an SDL Diagram 161

Using CM SYNERGY Together with an SDL System 162
Introducing CM SYNERGY with the SDL Suite - Migration 162
Introducing CM SYNERGY with the SDL Suite - Set up your working environment 167

Introducing CM SYNERGY with the SDL Suite - Day-to-Day Working with CM SYNERGY 168
Using ClearCase Together with an SDL System 169
Introducing ClearCase with the SDL Suite – Checking in Files 169
Introducing ClearCase with the SDL Suite – Opening a System 170

IBM Rational SDL Suite 6.3

Methodology Guidelines

This edition applies to IBM Rational SDL Suite 6.3 and IBM Rational TTCN Suite 6.3 and to all subsequent releases and modifications until otherwise indicated in new editions.

Copyright License

Trademarks

Introduction

About this Manual

This volume, *Methodology Guidelines*, contains some practical guidelines on how to use SDL and related notations in software development using the SDL Suite.

The SDL-92 object oriented concepts are introduced, including converting an SDL-88 system into SDL-92. Some IBM Rational-specific extensions to SDL are described, as well as the handling of SDL, C and ASN.1 data types in the SDL Suite. Finally, how to manage the diagram files in a project that includes several project members is explained and exemplified.

Documentation Overview

A general description of the documentation can be found in "Documentation" on page viii in the Release Guide.

Typographic Conventions

The typographic conventions that are used in the documentation are described in "Typographic Conventions" on page x in the Release Guide.

How to Contact Customer Support

Detailed contact information for IBM Rational Customer Support can be found in "How to Contact Customer Support" on page iv in the Release Guide.

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

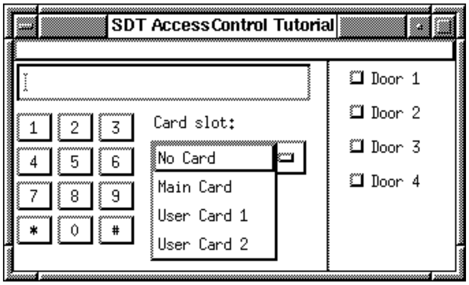
This application is chosen because it is a good example of an embedded system, with features that make it very suitable to be specified using SDL and the object oriented extensions (introduced in the 1992 version of the language).

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:

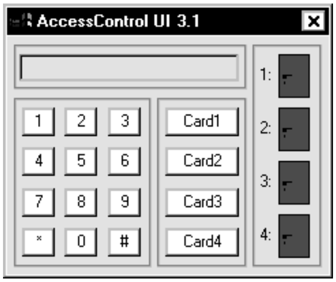


Figure 1: Graphical interface to the Access Control system

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.

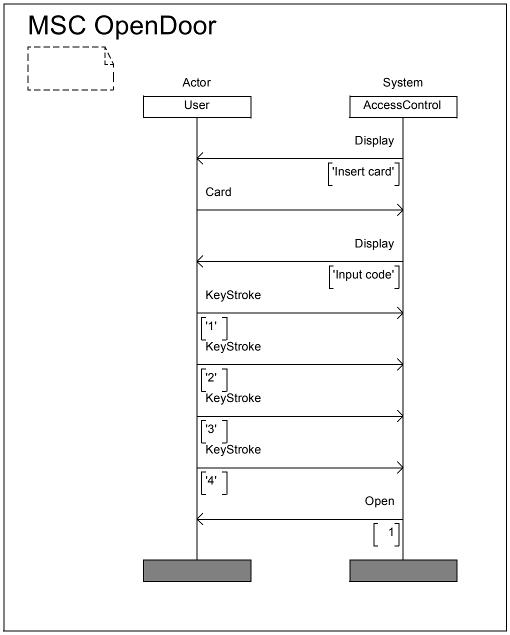


Figure 2: Requirements use case OpenDoor

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.

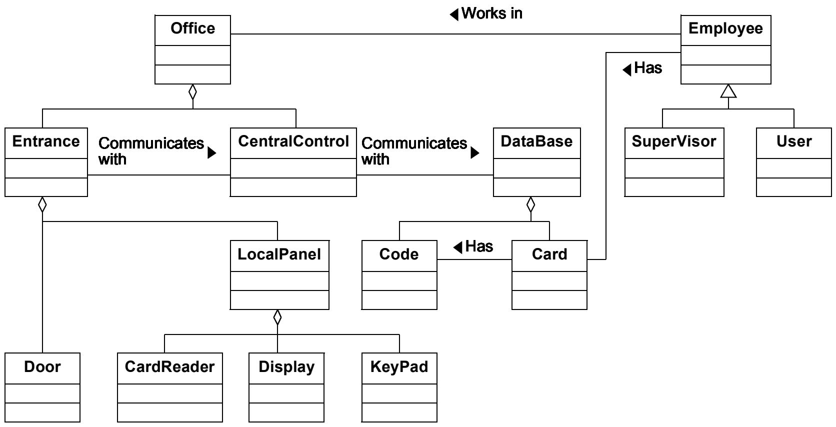


Figure 3: Requirements object model

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.

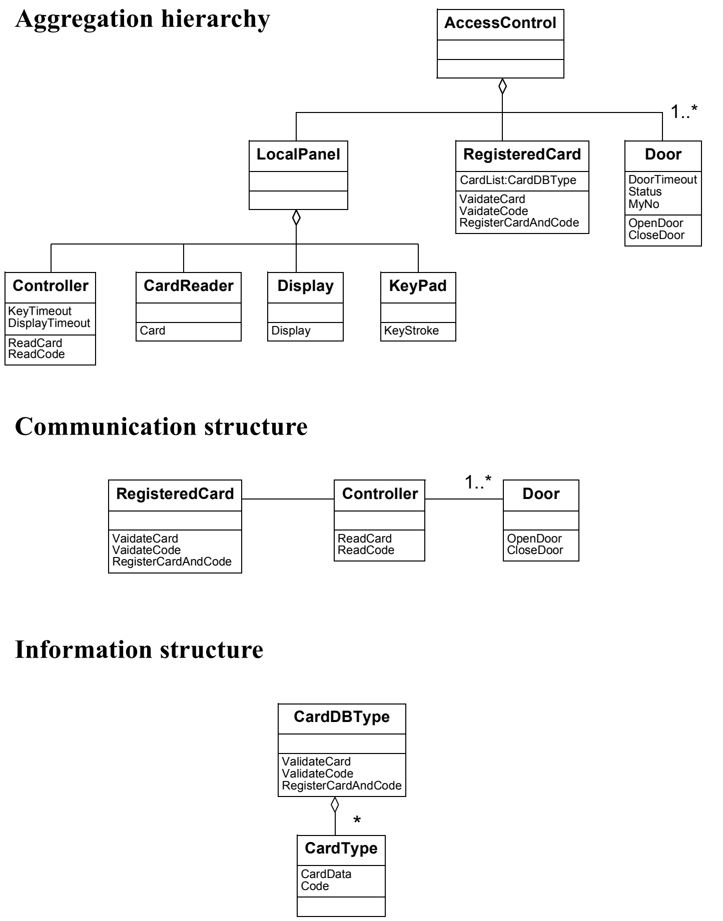


Figure 4: The analysis object model of the Access Control system (basic version)

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).

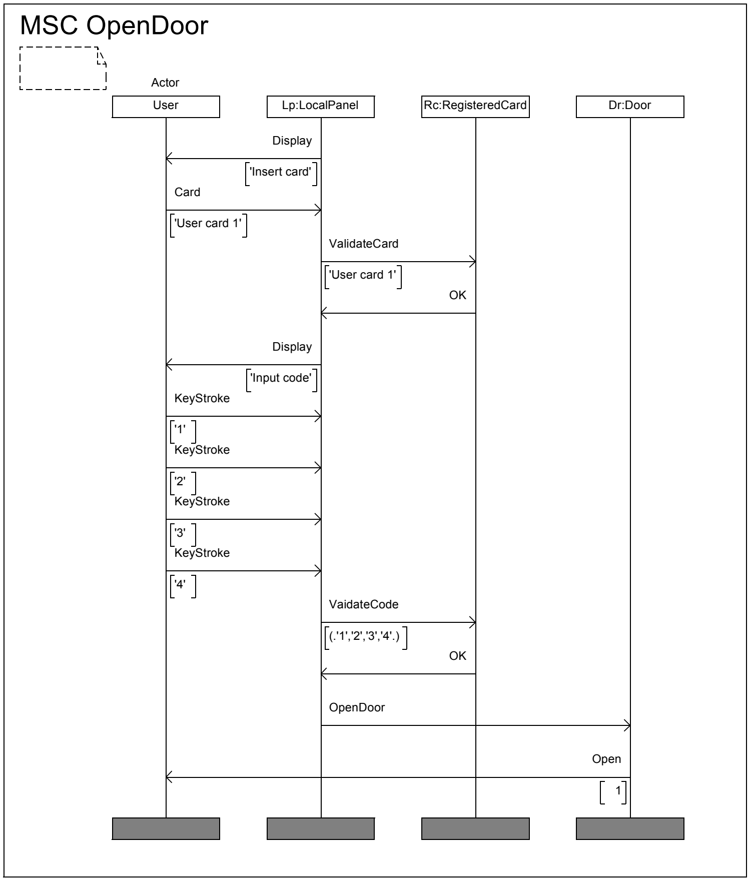


Figure 5: Analysis use case OpenDoor

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.

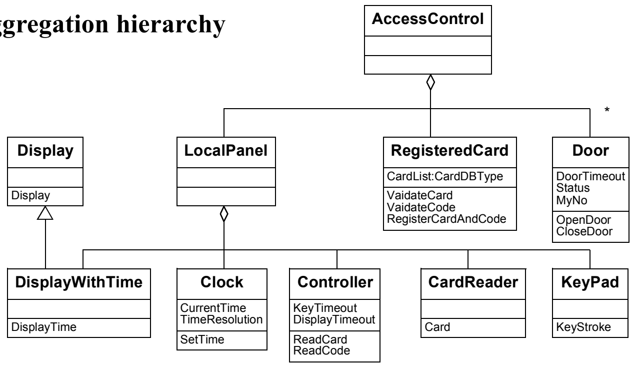
It should also be noticed that all MSC messages do not map strictly to class operations. In some cases, the operation is synchronous, that is demands a return message. This return message is also described in the MSC use case in Figure 2 (e.g. the operation *ValidateCard* is described by the messages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

Analysis Object Model: Enhanced Version

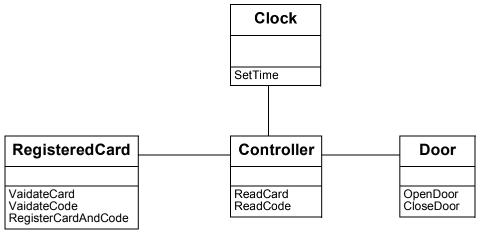
The following example is how an analysis of the additional requirement of time handling can be performed. The addition of a clock function will mainly add a new operation to the class Display (display of current time). A new class Clock must also be introduced. The properties of this class handle the 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

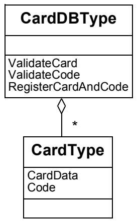


Figure 6: The analysis object model of the Access Control system (extended version with time handling)

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 Explorer 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).

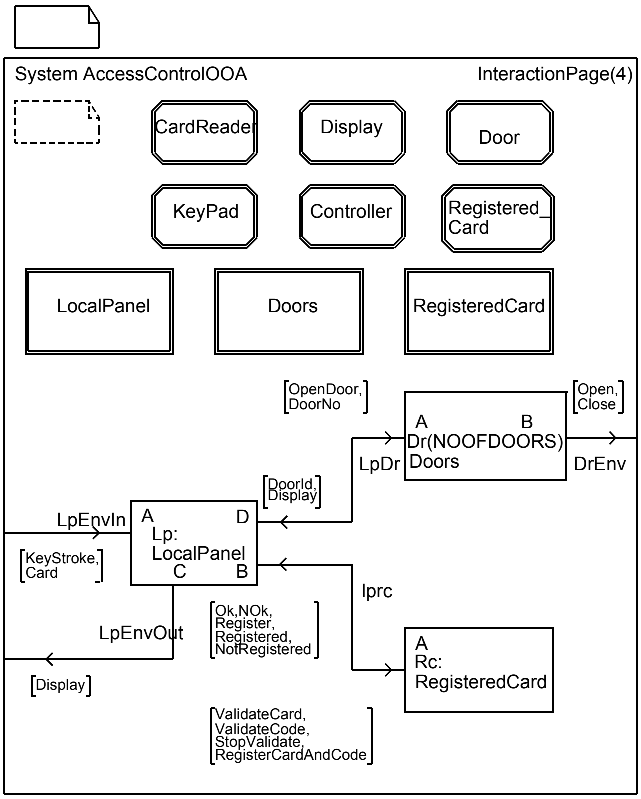


Figure 7: System diagram AccessControlOOA

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).

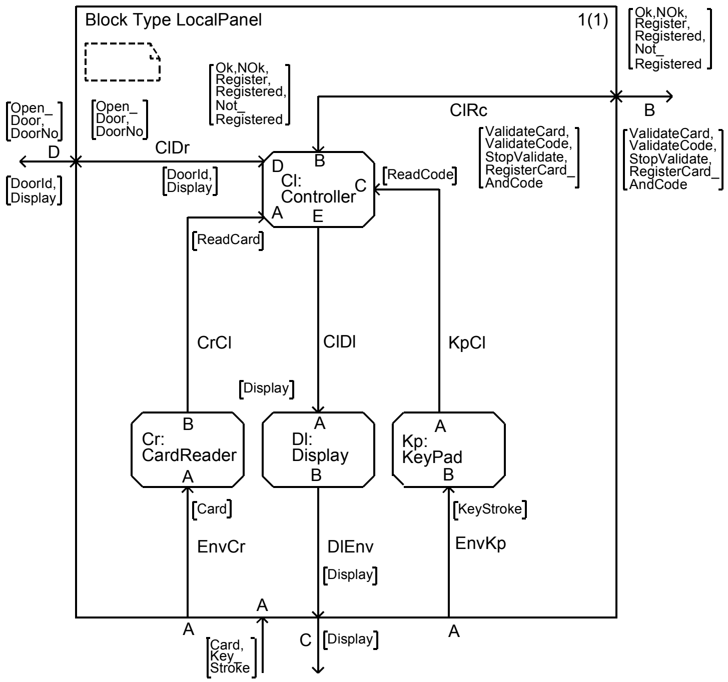
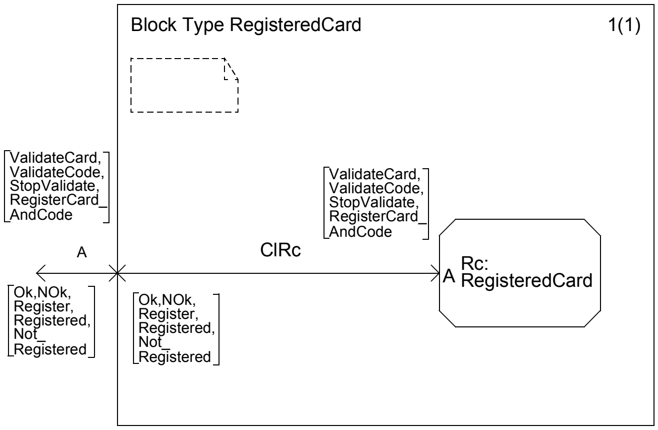


Figure 8: The block type LocalPanel

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).



Block Type RegisteredCard
1(1)
ValidateCard, ValidateCode, StopValidate, RegisterCard_AndCode
A
Ok,NOk, Register, Registered, Not_Registered
ClRc
ValidateCard, ValidateCode, StopValidate, RegisterCard_AndCode
Ok,NOk, Register, Registered, Not_Registered
A
Rc: RegisteredCard

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).

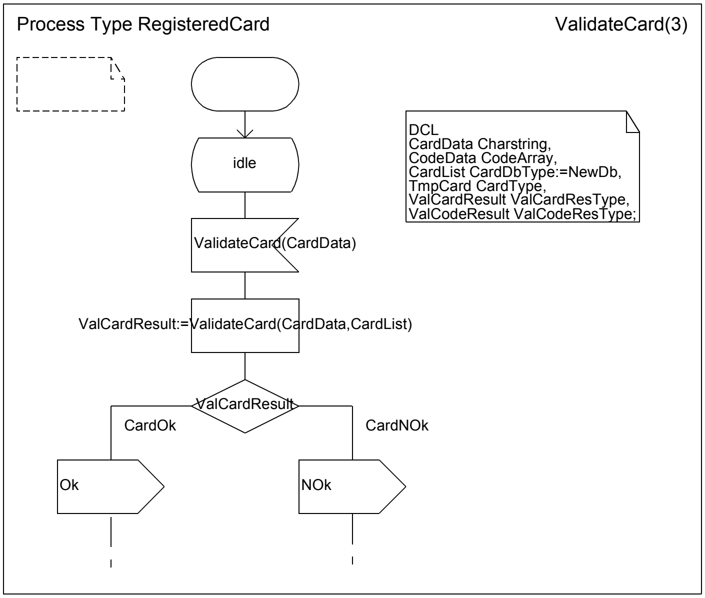

Process Type RegisteredCard
ValidateCard(3)
DCL
CardData Charstring,
CodeData CodeArray,
CardList CardDbType:=NewDb,
TmpCard CardType,
ValCardResult ValCardResType,
ValCodeResult ValCodeResType;
idle
ValidateCard(CardData)
ValCardResult:=ValidateCard(CardData,CardList)
ValCardResult
CardOk
CardNOk
Ok
NOk

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).

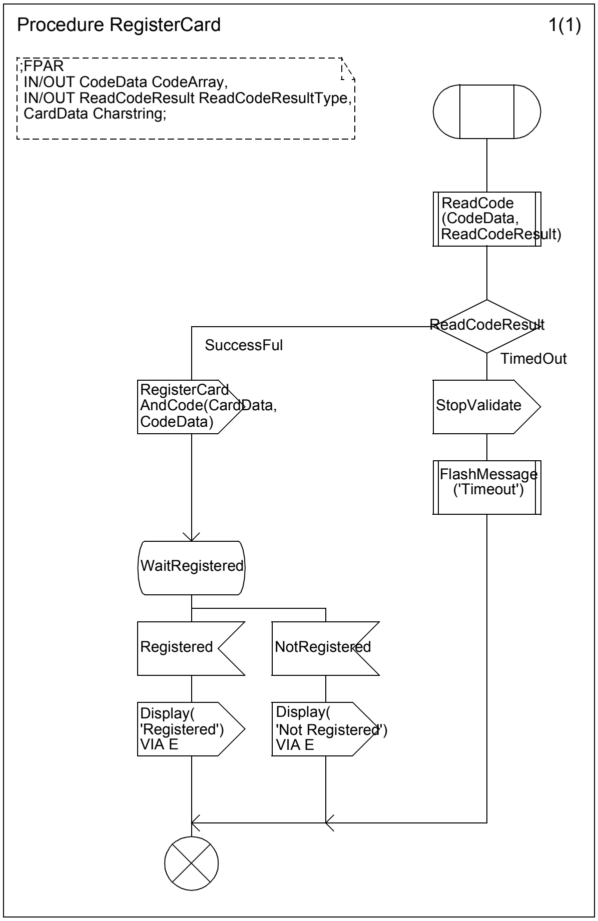

Procedure RegisterCard
1(1)
;FPAR
IN/OUT CodeData CodeArray,
IN/OUT ReadCodeResult ReadCodeResultType,
CardData Charstring;
ReadCode (CodeData, ReadCodeResult)
ReadCodeResult
SuccessFul
TimedOut
RegisterCardAndCode(CardData, CodeData)
StopValidate
FlashMessage ('Timeout')
WaitRegistered
Registered
NotRegistered
Display('Registered') VIA E
Display('Not Registered') VIA E

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).

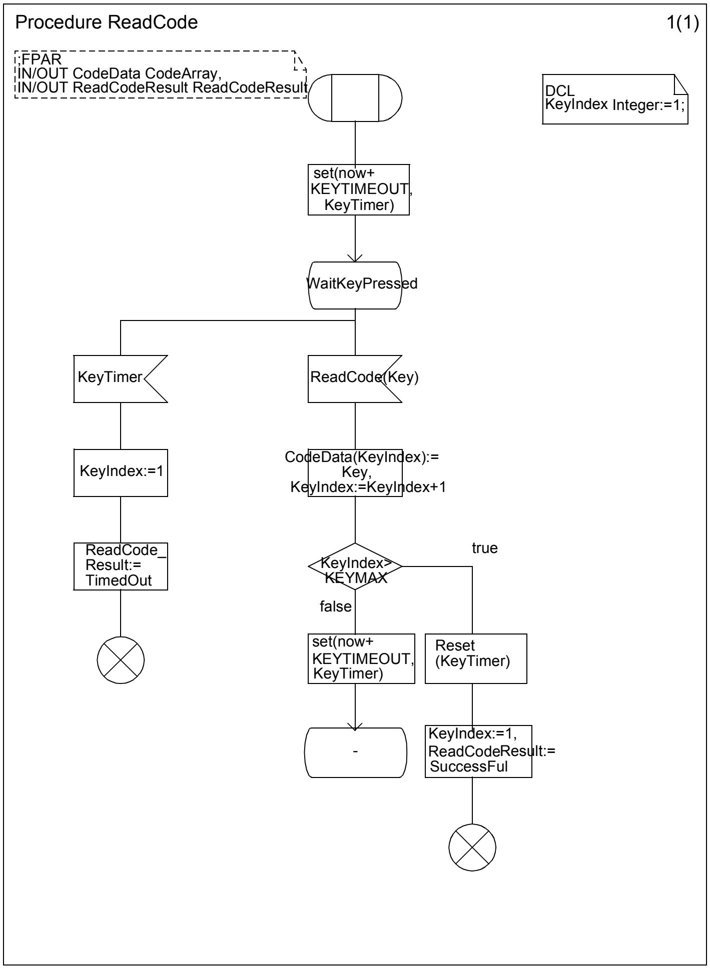

Procedure ReadCode
1(1)
;FPAR
IN/OUT CodeData CodeArray,
IN/OUT ReadCodeResult ReadCodeResult
DCL
KeyIndex Integer:=1;
set(now+ KEYTIMEOUT, KeyTimer)
WaitKeyPressed
KeyTimer
ReadCode(Key)
KeyIndex:=1
CodeData(KeyIndex):= Key, KeyIndex:=KeyIndex+1
ReadCode_ Result:= TimedOut
KeyIndex> KEYMAX
true
false
set(now+ KEYTIMEOUT, KeyTimer)
Reset (KeyTimer)
-
KeyIndex:=1, ReadCodeResult:= SuccessFul

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 procedure 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.

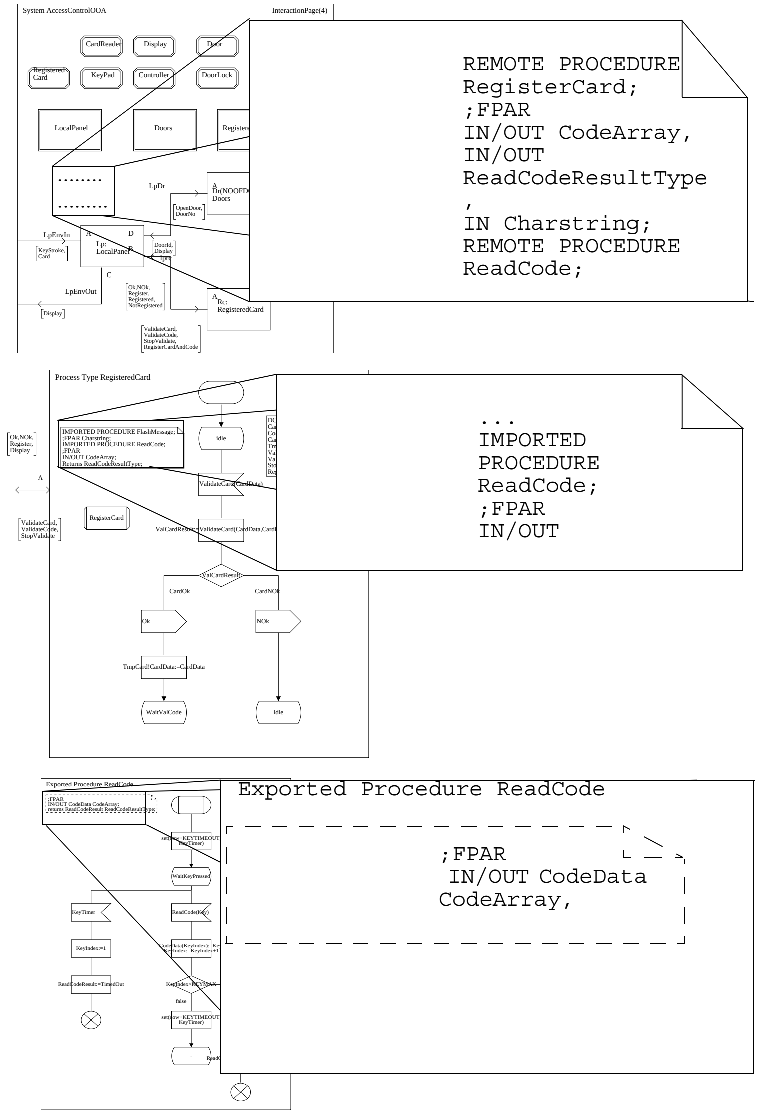


Figure 14: Declaration of a remote procedure

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).

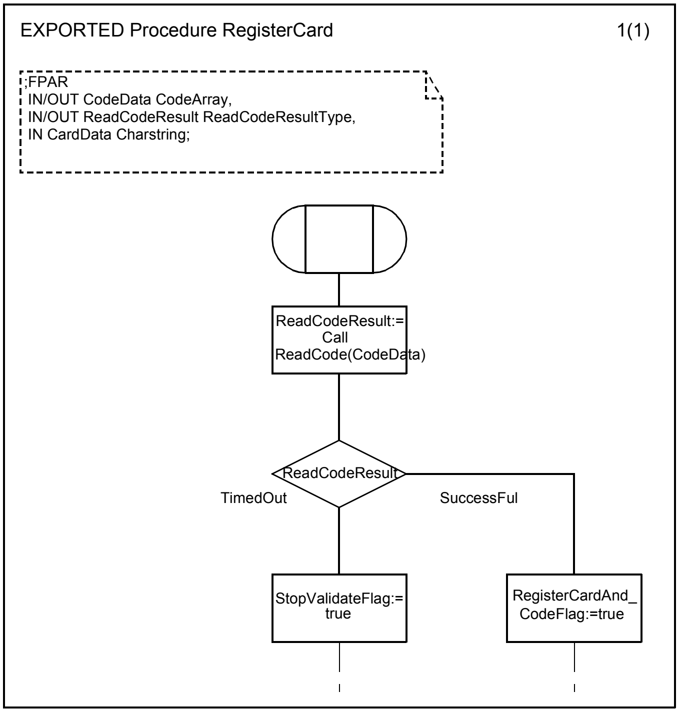

EXPORTED Procedure RegisterCard
1(1)
;FPAR
IN/OUT CodeData CodeArray,
IN/OUT ReadCodeResult ReadCodeResultType,
IN CardData Charstring;
ReadCodeResult:= Call ReadCode(CodeData)
ReadCodeResult
TimedOut
SuccessFul
StopValidateFlag:= true
RegisterCardAnd_ CodeFlag:=true

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.

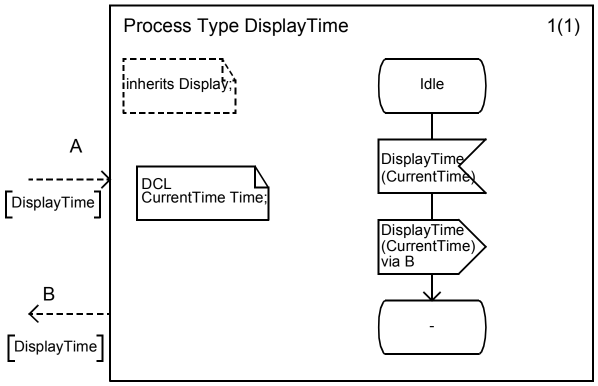


Figure 16: The process type TimeDisplay

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 Close-Door respectively to the new process DoorOpener. Therefore, the corresponding transitions of Door have to be defined as virtual transitions, as depicted in Figure 17.

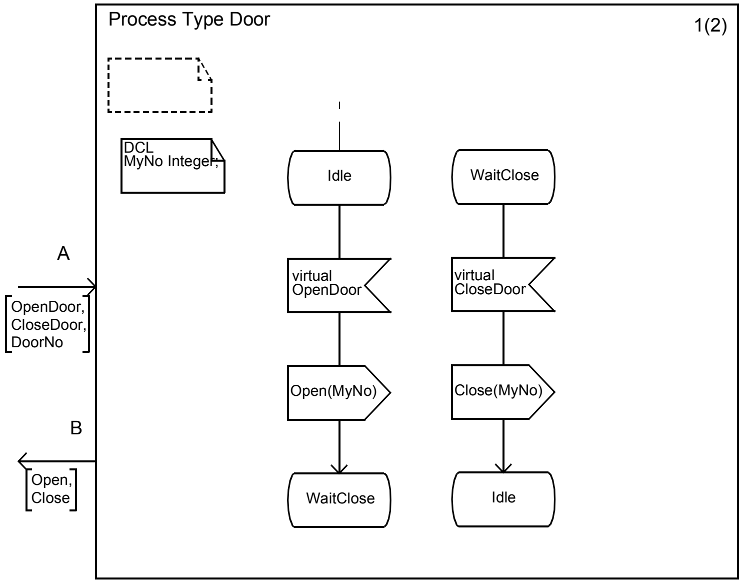


Figure 17: The process type Door with virtual transitions

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.

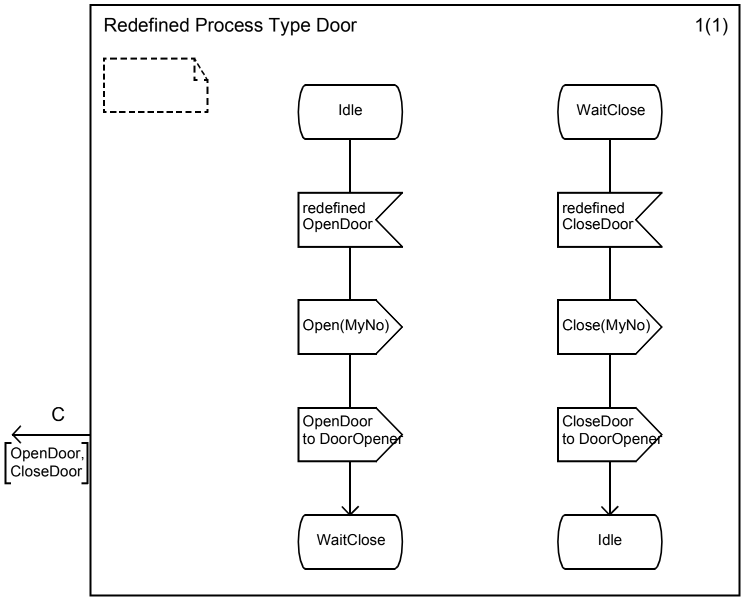


Figure 18: The redefined process type Door

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.

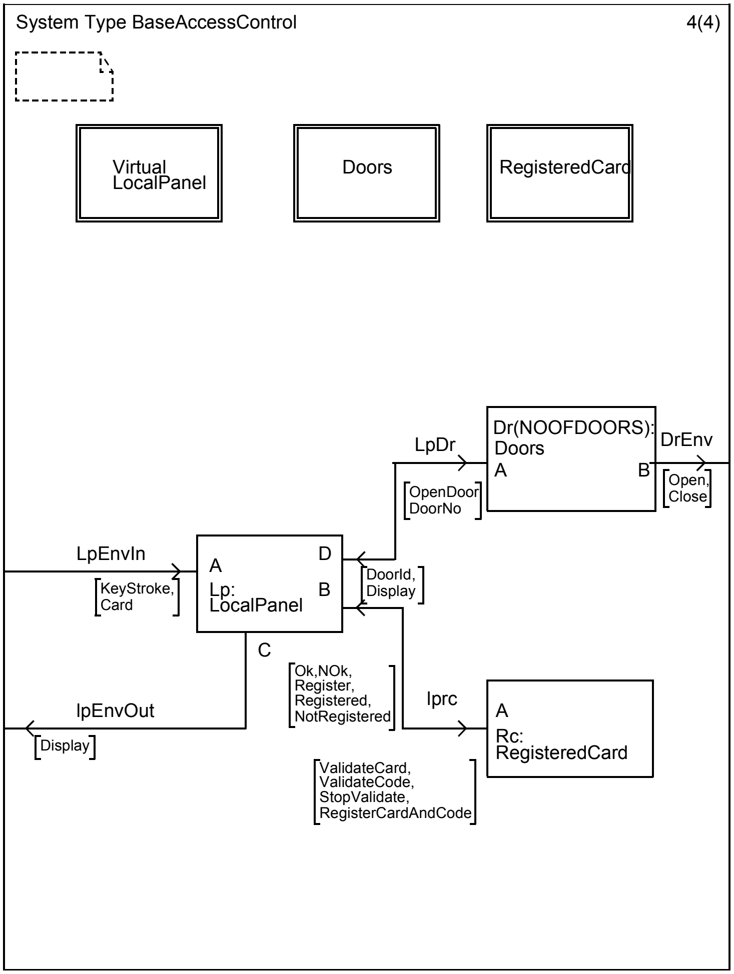


Figure 19: The system type BaseAccessControl

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.

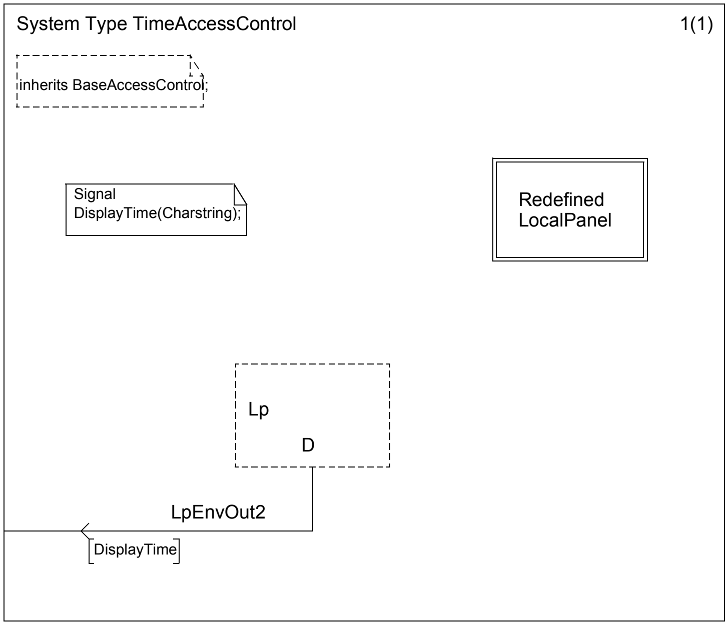


Figure 20: The system type TimeAccessControl

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.

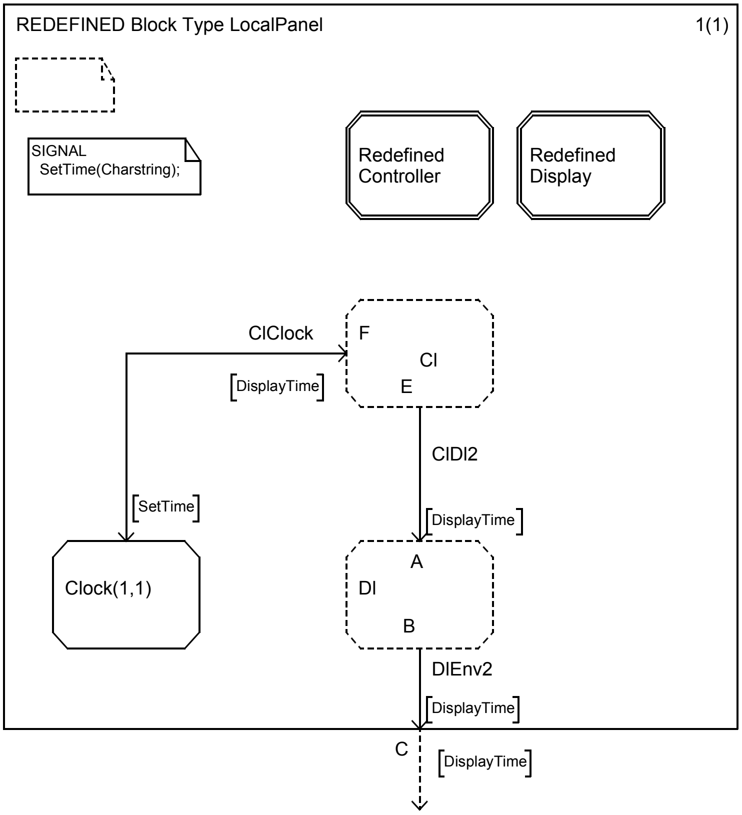


Figure 21: The redefined block type LocalPanel

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.

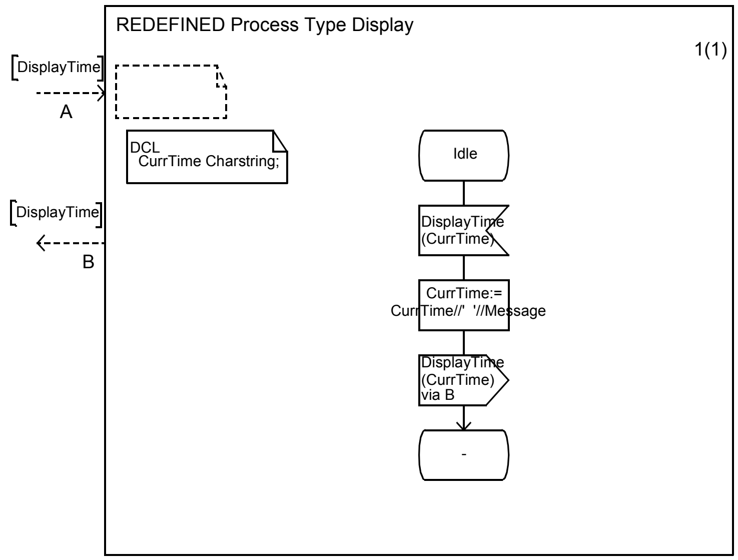


Figure 22: The redefined process type Display

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.

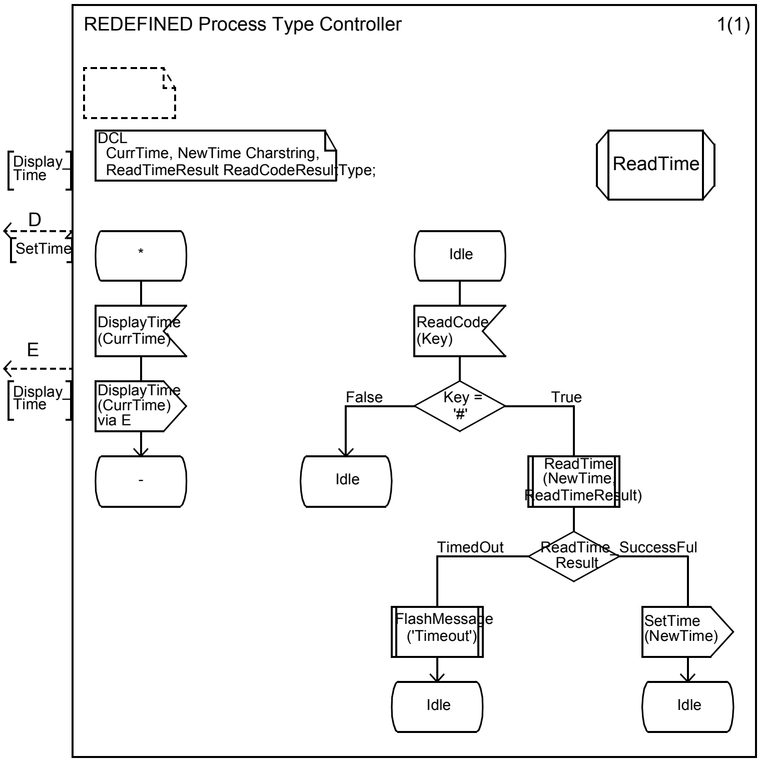


Figure 23: The redefined process type Controller

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.

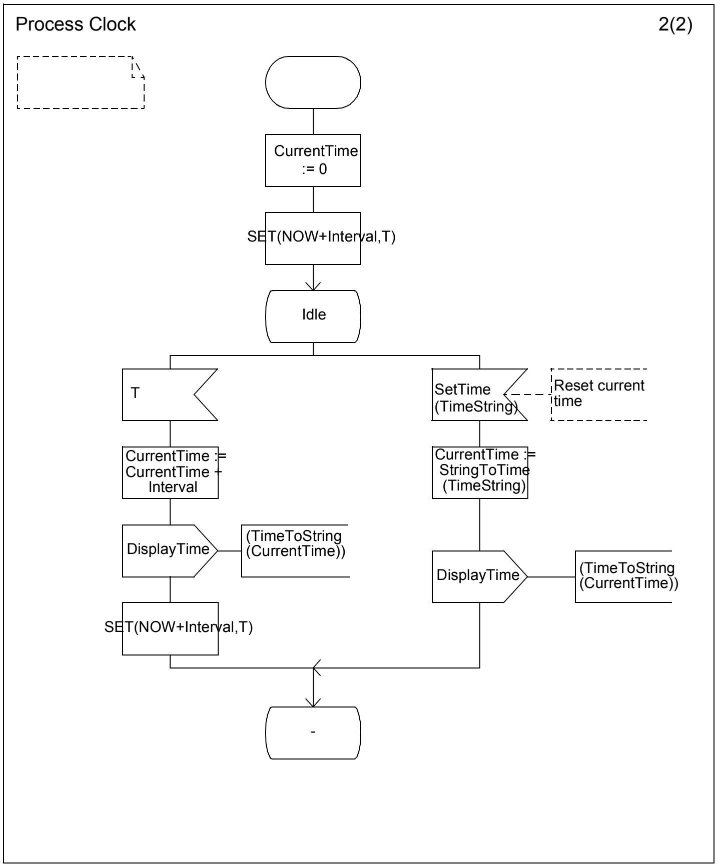


Figure 24: The process Clock

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.

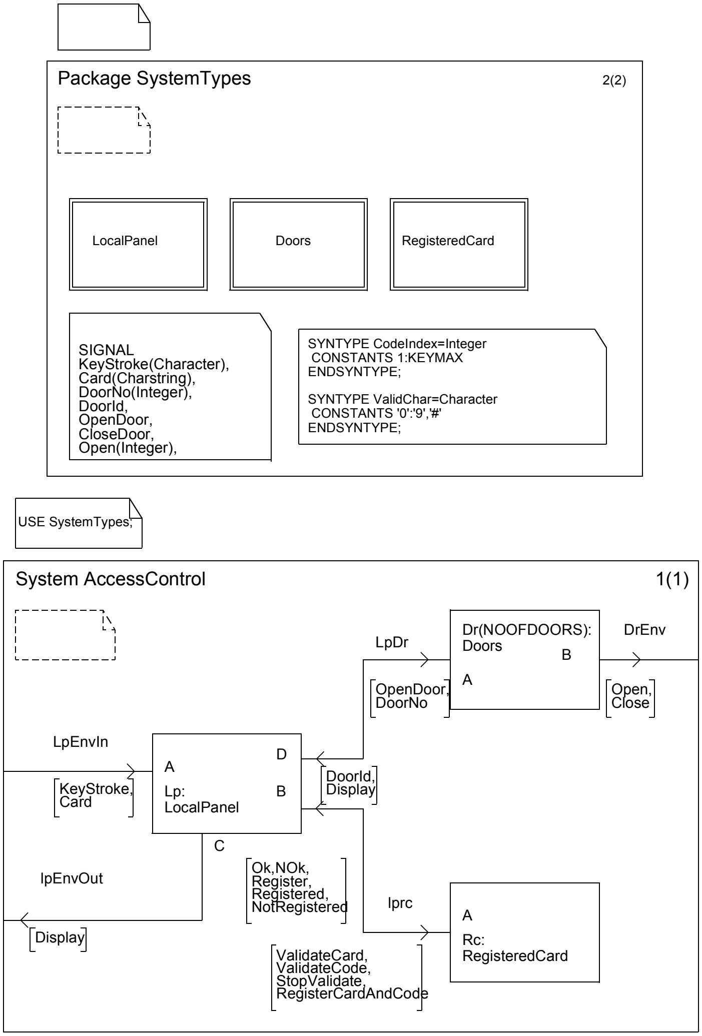


Figure 25: The use of packages in the Access Control system

Chapter 2 Data Types

This chapter describes how data types are handled in the SDL Suite. An overview of all supported SDL data types is given, including examples and guidelines. It is also explained how to use C/C++ and ASN.1, in combination with the SDL Suite.

Chapter 2 Data Types

Introduction

An important and often difficult aspect of system design and implementation is how to handle data in the system.

The SDL Suite offers several ways to use data:

- SDL-specific data types can be used
- Access to C/C++ data types and functions is supported
- ASN.1 data types can be used

This chapter gives an overview of all available data types, together with some guidelines of how to use these different data types, illustrated with a number of examples.

Using SDL Data Types

In this section, an overview is given of the data types that are available in SDL. SDL contains a number of predefined data types. Based on these predefined types it is possible to define user-specific data types. Types, or according to SDL terminology, “sorts”, are defined using the keywords `newtype` and `endnewtype`.

Example 1: Newtype definition

```
newtype example1 struct
  a integer;
  b character;
endnewtype;
```

A newtype definition introduces a new distinct type, which is not compatible with any other type. So if we would have another newtype `otherexample` with exactly the same definition as `example1` above, it would not be possible to assign a value of `example1` to a variable of `otherexample`.

It is also possible to introduce types, *syntypes*, that are compatible with their base type, but contain restrictions on the allowed value set for the type. Syntypes are defined using the keywords `syntype` and `endsyntype`.

Example 2: Syntype definition

```
syntype example2 = integer
  constants 0:10
endsyntype;
```

The syntype `example2` is an integer type, but a variable of this type is only allowed to contain values in the specified range 0 to 10. Such a constant clause is called a *range condition*. The range check is performed when the SDL system is interpreted. Without a range condition a syntype definition just introduces a new name for the same sort.

For every sort or syntype defined in SDL, the following operators are always defined:

- `:=` (assignment)
- `=` (test for equality)
- `/=` (test for non-equality)

These operators are not mentioned among the available operators in the rest of this section. Operators are defined in SDL by a type of algebra according to the following example:

```
"+" : Integer, Integer -> Integer;
num : Character -> Integer;
```

The double quotes around the `+` indicate that this is an infix operator. The above `+` takes two integer parameters and returns an integer value. The second operator, `num`, is a prefix operator taking one Character and returning an Integer value. The operators above can be called within expressions in, for example, task statements:

```
task i := i+1;
task n := num('X');
```

where it is assumed that `i` and `n` are integer variables. It is also allowed to call an infix operator as a prefix operator:

```
task i := "+"(i, 1);
```

This means the same as `i:= i+1`.

Predefined Sorts

The predefined sorts in SDL are defined in an appendix to the SDL Recommendation Z100. Some more predefined sorts are introduced in the Recommendation Z105, where it is specified how ASN.1 is to be used in SDL. These types should not be used if the SDL system must conform to Z.100. The SDL Suite also offers IBM Rational-specific operators for some types. These operators should not either be used if your SDL system must be Z.100 compliant. The rest of this chapter describes all predefined sorts. Unless stated otherwise, the sort is part of recommendation Z.100.

Bit

The predefined `Bit` can only take two values, `0` and `1`. Bit is defined in Z.105 for the definition of bit strings, and is not part of Z.100. The operators that are available for Bit values are:

```
"not" : Bit -> Bit
"and" : Bit, Bit -> Bit
"or"  : Bit, Bit -> Bit
"xor" : Bit, Bit -> Bit
"=>"  : Bit, Bit -> Bit
```

These operators are defined according to the following:

- `not` :
  inverts the bit; 0 becomes 1 and 1 becomes 0,
  `not 0` gives `1`, `not 1` gives `0`
- `and` :
  if both parameters are 1, the result is 1, else it is 0,
  `0 and 0` gives `0`, `0 and 1` gives `0`, `1 and 1` gives `1`
- `or` :
  if both parameters are 0, the result is 0, else it is 1,
  `0 or 0` gives `0`, `0 or 1` gives `1`, `1 or 1` gives `1`
- `xor` :
  if parameters are different, the result is 1, else it is 0,
  `0 xor 0` gives `0`, `0 xor 1` gives `1`, `1 xor 1` gives `0`
- `=>` (implication) :
  if first parameter is 1 and second is 0, the result is 0, else it is 1,
  `0 => 0` gives `1`, `1 => 0` gives `0`, `0 => 1` gives `1`, `1 => 1` gives `1`

The Bit type has most of its properties in common with the Boolean type, which is discussed below. By replacing `0` with `false` and `1` with `true` the sorts are identical.

Bit and Boolean should be used to represent properties in a system that can only take two values, like on - off. In the choice between Bit and Boolean, Boolean is recommended except if the property to be represented is about bits and the literals `0` and `1` are more adequate than `false` and `true`.

Bit_string

The predefined sort `Bit_string` is used to represent a string or sequence of Bits. Bit_string is defined in Z.105 to support the ASN.1 `BIT STRING` type, and is not part of Z.100. There is no limit on the number of elements in the Bit_string.

The following operators are defined in Bit_string:

```
mkstring  : Bit                     -> Bit_string
length    : Bit_string              -> Integer
first     : Bit_string              -> Bit
last      : Bit_string              -> Bit
"//"      : Bit_string, Bit_string -> Bit_string
substring : Bit_string, Integer, Integer
                                    -> Bit_string
bitstr    : Charstring              -> Bit_string
hexstr    : Charstring              -> Bit_string
"not"     : Bit_string              -> Bit_string
"and"     : Bit_string, Bit_string -> Bit_string
"or"      : Bit_string, Bit_string -> Bit_string
"xor"     : Bit_string, Bit_string -> Bit_string
"=>"      : Bit_string, Bit_string -> Bit_string
```

These operators are defined as follows:

- `mkstring` :
  This operator takes a Bit value and converts it to a Bit_string of length 1.
  `mkstring (0)` gives a Bit_string of one element, i.e. `0`

- `length` :
  The number of Bits in the Bit_string passed as parameter.
  `length (bitstr('0110')) = 4`

- `first` :
  The value of the first Bit in the Bit_string passed as parameter. If the length of the Bit_string is 0, then it is an error to call the first operator.
  `first (bitstr ('10')) = 1`

- `last` :
  The value of the last Bit in the Bit_string passed as parameter. If the length of the Bit_string is 0, then it is an error to call the last operator.
  `last (bitstr ('10')) = 0`

- `//` (concatenation) :
  The result is a Bit_string with all the elements in the first parameter, followed by all the elements in the second parameter.
  `bitstr('01')//bitstr('10') = bitstr('0110')`

- `substring` :
  The result is a copy of a part of the Bit_string passed as first parameter. The copy starts at the index given as second parameter. The first Bit has index 0. The length of the copy is specified by the third parameter. It is an error to try to access elements outside of the true length of the first parameter.
  `substring (bitstr('0110'), 1, 2) = Bitstr('11')`

- `bitstr` :
  This IBM Rational-specific operator converts a charstring containing only characters 0 and 1, to a Bit_string with the same length and with the Bit elements set to the corresponding values.

- `hexstr` :
  This IBM Rational-specific operator converts a charstring containing HEX values (0 -9, A-F, a-f) to a Bit_string. Each HEX value is converted to four Bit elements in the Bit_string.
  `hexstr('a') = bitstr('1010'),`
  `hexstr('8f') = bitstr('10001111')`

- `not` :
  The result is a Bit_string with the same length as the parameter, where the `not` operator in the Bit sort has been applied to each element, that is each element has been inverted.
  `not bitstr ('0110') = bitstr ('1001')`

- `and` :
  The result is a Bit_string with the same length as the longest of the two parameters. Each Bit element in the result is computed by applying the and operator in Bit sort to the corresponding Bit elements in the parameters. This calculation is performed up to the length of the shortest parameter. The remaining Bits in the result (if any) are set to 0.
  ```
  bitstr('01101') and bitstr('101') = bitstr('00100')
  ```
- `or` :
  The result is a Bit_string with the same length as the longest of the two parameters. Each Bit element in the result is computed by applying the or operator in Bit sort to the corresponding Bit elements in the parameters. This calculation is performed up to the length of the shortest parameter. The remaining Bits in the result (if any) are set to 1.
  ```
  bitstr('0110') or bitstr('00110') = bitstr('01111')
  ```
- `xor` :
  The result is a Bit_string with the same length as the longest of the two parameters. Each Bit element in the result is computed by applying the xor operator in Bit sort to the corresponding Bit elements in the parameters. This calculation is performed up to the length of the shortest parameter. The remaining Bits in the result (if any) are set to 1.
  ```
  bitstr('10100') xor bitstr('1001') = bitstr('00111')
  ```
- `=>` (implication) :
  The result is a Bit_string with the same length as the longest of the two parameters. Each Bit element in the result is computed by applying the => operator in Bit sort to the corresponding Bit elements in the parameters. This calculation is performed up to the length of the shortest parameter. The remaining Bits in the result (if any) are set to 1.
  ```
  bitstr ('1100') => bitstr ('0101') = bitstr ('0111')
  ```

It is also possible to access Bit elements in a Bit_string by indexing a Bit_string variable. Assume that `B` is a Bit_string variable. Then it is possible to write:

```
task B(2) := B(3);
```

This would mean that Bit number 2 is assigned the value of Bit number 3 in the variable `B`. Is is an error to index a Bit_string outside of its length.

Note:

The first Bit in a Bit_string has index 0, whereas most other string types in SDL start with index 1!

Boolean

The newtype `Boolean` can only take two values, `false` and `true`. The operators that are available for Boolean values are:

```
"not" : Boolean -> Boolean
"and" : Boolean, Boolean -> Boolean
"or"  : Boolean, Boolean -> Boolean
"xor" : Boolean, Boolean -> Boolean
"=>"  : Boolean, Boolean -> Boolean
```

These operators are defined according to the following:

- `not` :
  inverts the value.

  ```
  not false = true
  not true  = false
  ```
- `and` :
  If both parameters are true then the result is true, else it is false.

  ```
  false and false = false
  false and true  = false
  true  and false = false
  true  and true  = true
  ```
- `or` :
  If both parameters are false then the result is false, else it is true.

  ```
  false or false = false
  false or true  = true
  true  or false = true
  true  or true  = true
  ```

- `xor` :
  If parameters are different then the result is true, else it is false.

  ```
  false xor false = false
  false xor true  = true
  true  xor false = true
  true  xor true  = false
  ```

- `=>` (implication) :
  If the first parameter is true and second is false then the result is false, else it is true.

  ```
  false => false = true
  false => true  = true
  true  => false = false
  true  => true  = true
  ```

The Bit sort, discussed above, has most of its properties in common with the Boolean sort. By replacing `0` with `false` and `1` with `true` the sorts are identical. Normally it is recommended to use Boolean instead of Bit; for a more detailed discussion see “Bit” on page 44.

Character

The `character` sort is used to represent the ASCII characters. The printable characters have literals according to the following example:

```
'a'  '-'  '?' '2'  'P'  ''''
```

Note that the character `'` is written twice in the literal. For the non-printable characters, specific literal names have been included in the Character sort. The following:

```
NUL,  SOH,  STX,  ETX,  EOT,  ENQ,  ACK,  BEL,
BS,   HT,   LF,   VT,   FF,   CR,   SO,   SI,
DLE,  DC1,  DC2,  DC3,  DC4,  NAK,  SYN,  ETB,
CAN,  EM,   SUB,  ESC,  IS4,  IS3,  IS2,  IS1
```

correspond to the characters with number 0 to 31, while the literal

```
DEL
```

corresponds to the character number 127.

The operators available in the Character sort are:

```
"<"  : Character, Character  -> Boolean;
"<=" : Character, Character  -> Boolean;
">"  : Character, Character  -> Boolean;
">=" : Character, Character  -> Boolean;
num  : Character             -> Integer;
chr  : Integer               -> Character;
```

The interpretation of these operators are:

- `<, <=, >, >=` :
  These relation operators work with the character numbers according to the ASCII table.
- `num` :
  This operator converts a Character value to its corresponding character number. For example: `num('A') = 65`
- `chr` :
  This operator converts an Integer value to its corresponding character. If the parameter is less than 0 or bigger than 255, it is first taken modulo 256 (using the mod operator in sort Integer). For example:
  `chr(65) = 'A'`

In Z.100 characters in the range 0 to 127 are supported. However IBM Rational has introduced support for characters in the range 0 to 255. This means two things

The operator `num` works modulo 256, not modulo 128 as it is defined in Z.100.

The following literals (128 to 255) are added to the Character sort:

```
E_NUL, E_SOH, E_STX, E_ETX, E_EOT, E_ENQ, E_ACK, E_BEL,
E_BS,  E_HT,  E_LF,  E_VT,  E_FF,  E_CR,  E_SO,  E_SI,
E_DLE, E_DC1, E_DC2, E_DC3, E_DC4, E_NAK, E_SYN, E_ETB,
E_CAN, E_EM,  E_SUB, E_ESC, E_IS4, E_IS3, E_IS2, E_IS1,
' ',   '¡',   '¢',   '£',   '¤',   '¥',   '¦',   '§',
'¨',   '©',   'ª',   '«',   '¬',   '',   '®',   '¯',
'°',   '±',   '²',   '³',   '´',   'µ',   '¶',   '·',
'¸',   '¹',   'º',   '»',   '¼',   '½',   '¾',   '¿',
'À',   'Á',   'Â',   'Ã',   'Ä',   'Å',   'Æ',   'Ç',
'È',   'É',   'Ê',   'Ë',   'Ì',   'Í',   'Î',   'Ï',
'Ð',   'Ñ',   'Ò',   'Ó',   'Ô',   'Õ',   'Ö',   '×',
'Ø',   'Ù',   'Ú',   'Û',   'Ü',   'Ý',   'Þ',   'ß',
'à',   'á',   'â',   'ã',   'ä',   'å',   'æ',   'ç',
'è',   'é',   'ê',   'ë',   'ì',   'í',   'î',   'ï',
'ð',   'ñ',   'ò',   'ó',   'ô',   'õ',   'ö',   '÷',
'ø',   'ù',   'ú',   'û',   'ü',   'ý',   'þ',   'ÿ';
```

Charstring

The `Charstring` sort is used to represent strings or sequences of characters. There is no limit for the length of a Charstring value. Charstring literals are written as a sequence of characters enclosed between two single quotes: `'abc'`. If the Charstring should contain a quote (`'`) it must be written twice.

```
'abcdef 0123'
'$%@^&'
'1''2''3' /* denotes 1'2'3 */
''        /* empty Charstring */
```

The following operators are available for Charstrings:

```
mkstring  : Character              -> Charstring;
length    : Charstring             -> Integer;
first     : Charstring             -> Character;
last      : Charstring             -> Character;
"//"      : Charstring, Charstring -> Charstring;
substring : Charstring, Integer, Integer
                                   -> Charstring;
```

These operators are defined as follows:

- `mkstring` :
  This operator takes one Character value and converts it to a Charstring of length 1. For example: if `c` is a variable of type Character, then `mkstring(c)` is a Charstring containing character `c`.

- `length` :
  This operator takes a Charstring as parameter and returns its number of characters.
  `length ('hello') = 5`

- `first` :
  The value of the first Character in the Charstring passed as parameter. If the length of the Charstring is 0, then it is an error to call the first operator.
  `first ('hello') = 'h'`

- `last` :
  The value of the last Character in the Charstring passed as parameter. If the length of the Charstring is 0, then it is an error to call the last operator.
  `last ('hello') = 'o'`

- `//` (concatenation) :
  The result is a Charstring with all the elements in the first parameter, followed by all the elements in the second parameter.
  `'he' // 'llo' = 'hello'.`

- `substring` :
  The result is a copy of a part of the Charstring passed as first parameter. The copy starts at the index given as second parameter (Note: first Character has index 1). The length of the copy is specified by the third parameter. It is an error to try to access elements outside of the true length of the first parameter.
  `substring ('hello', 3, 2) = 'll'`

It is also possible to access Character elements in a Charstring by indexing a Charstring variable. Assume that `C` is a Charstring variable. Then it is possible to write:

```
task C(2) := C(3);
```

This would mean that Character number 2 is assigned the value of Character number 3 in the variable `C`.

Note:
The first Character in a Charstring has index 1.

IA5String, NumericString, PrintableString, VisibleString

These Z.105 specific character string types are all syntypes of Charstring with restrictions on the allowed Characters that may be contained in a value. These sorts are mainly used as a counterpart of the ASN.1 types with the same names. The restrictions are:

- `IA5String` :
  only `NUL:DEL`, i.e only characters in the range 0 to 127.
- `NumericString` :
  only `'0':'9'` and `' '`
- `PrintableString` :
  only `'A':'Z', 'a':'z', '0':'9', ' ', '''':')', '+':'/', ':', '=', '?'`
- `VisibleString`:
  only `' ':'~'`

It is recommended to use these types only in relation with ASN.1 or TTCN. In other cases use Charstring.

Duration, Time

The `Time` and `Duration` sorts have their major application area in connection with timers. The first parameter in a `Set` statement is the time when the timer should expire. This value should be of sort Time.

Both Time and Duration have literals with the same syntax as real values. Example:

```
245.72  0.0032  43
```

The following operators are available in the Duration sort:

```
"+"  : Duration, Duration -> Duration;
"-"  : Duration           -> Duration;
"-"  : Duration, Duration -> Duration;
"*"  : Duration, Real     -> Duration;
"*"  : Real,     Duration -> Duration;
"/"  : Duration, Real     -> Duration;
">"  : Duration, Duration -> Boolean;
"<"  : Duration, Duration -> Boolean;
">=" : Duration, Duration -> Boolean;
"<=" : Duration, Duration -> Boolean;
```

The following operators are available in the Time sort:

```
"+"  : Time,     Duration -> Time;
"+"  : Duration, Time     -> Time;
"-"  : Time,     Duration -> Time;
"-"  : Time,     Time     -> Duration;
"<"  : Time,     Time     -> Boolean;
"<=" : Time,     Time     -> Boolean;
">"  : Time,     Time     -> Boolean;
">=" : Time,     Time     -> Boolean;
```

The interpretation of these operators are rather straightforward, as they correspond directly to the ordinary mathematical operators for real numbers. There is one “operator” in SDL that returns a Time value; `now` which returns the current global system time.

Time should be used to denote “a point in time”, while Duration should be used to denote a “time interval”. SDL does not specify what the unit of time is. In the SDL Suite, the time unit is usually 1 second.

Example 3: Timers in SDL

```
SET (now + 2.5, MyTimer)
```

After the above statement, SDL timer `MyTimer` will expire after 2.5 time units (usually seconds) from now.

You should note that according to SDL, Time and Duration (and Real) possess the true mathematical properties of real numbers. In an implementation, however, there are of course limits on the range and precision of these values.

Integer, Natural

The `Integer` sort in SDL is used to represent the mathematical integers. `Natural` is a syntype of Integer, allowing only integers greater than or equal to zero.

Integer literals are defined using the ordinary integer syntax. Example:

```
0  5  173  1000000
```

Negative integers are obtained by using the unary - operator given below. The following operators are defined in the Integer sort:

```
"-"   : Integer           -> Integer;
"+"   : Integer, Integer  -> Integer;
"-"   : Integer, Integer  -> Integer;
"*"   : Integer, Integer  -> Integer;
"/"   : Integer, Integer  -> Integer;
"mod" : Integer, Integer  -> Integer;
"rem" : Integer, Integer  -> Integer;
"<"   : Integer, Integer  -> Boolean;
">"   : Integer, Integer  -> Boolean;
"<="  : Integer, Integer  -> Boolean;
">="  : Integer, Integer  -> Boolean;
float : Integer           -> Real;
fix   : Real              -> Integer;
```

The interpretation of these operators are given below:

- `-` (unary, i.e. one parameter) :
  Negate a value, e.g. -5.
- `+, -, * :`
  These operators correspond directly to their mathematical counterparts.

- `/ :`
  Integer division, e.g. `10/5 = 2, 14/5 = 2, -8/5 = -1`
- `mod, rem :`
  modulus and remainder at integer division. `mod` always returns a positive value, while `rem` may return negative values, e.g.
  `14 mod 5 = 4, 14 rem 5 = 4, -14 mod 5 = 1, -14 rem 5 = -4`
- `<, <=, >, >= :`
  These operators correspond directly to their mathematical counterparts.
- `float :`
  This operator converts an integer value to the corresponding Real number, for example:
  `float (3) = 3.0`
- `fix :`
  This operator converts a real value to the corresponding Integer number. It is performed by removing the decimal part of the Real value.
  `fix(3.65) = 3,  fix(-3.65) = -3`

NULL

`NULL` is a sort coming from ASN.1, defined in Z.105. NULL does occur rather frequently in older protocols specified with ASN.1. ASN.1 has later been extended with better alternatives, so NULL should normally not be used. The sort NULL only contains one value, NULL.

Object_identifier

The Z.105-specific sort `Object_identifier` also comes from ASN.1. Object identifiers usually identify some globally well-known definition, for example a protocol, or an encoding algorithm. Object identifiers are often used in open-ended applications, for example in a protocol where one party could say to the other “I support protocol version X”. “Protocol version X” could be identified by means of an object identifier.

An Object_identifier value is a sequence of Natural values. This sort contains one literal, `emptystring`, that is used to represent an Object_identifier with length 0. The operators defined in this sort are:

```
mkstring  : Natural           -> Object_identifier
length    : Object_identifier -> Integer
first     : Object_identifier -> Natural
last      : Object_identifier -> Natural
"//"      : Object_identifier, Object_identifier
                              -> Object_identifier
substring : Object_identifier, Integer, Integer
                              -> Object_identifier
append    : in/out Object_identifier, Natural;
(. .)     : * Natural         -> Object_identifier
```

These operators are defined as follows:

- `mkstring` :
  This operator takes one Natural value and converts it to an Object_identifier of length 1.
  `mkstring (8)` gives an Object_identifier consisting of one element, i.e. 8.

- `length` :
  This operator takes an Object_identifier as parameter and returns its number of object elements, i.e. Natural values.
  `length (mkstring (8)//mkstring(6)) = 2`
  `length (emptystring) = 0`

- `first` :
  The value of the first Natural in the Object_identifier passed as parameter. If the length of the Object_identifier is 0, then it is an error to call the first operator.
  `first (mkstring (8)//mkstring(6)) = 8`

- `last` :
  The value of the last Natural in the Object_identifier passed as parameter. If the length of the Object_identifier is 0, then it is an error to call the last operator.
  `last (mkstring (8)//mkstring(6)) = 6`

- `//` (concatenation) :
  The result is a Object_identifier with all the elements in the first parameter, followed by all the elements in the second parameter.
  `mkstring (8) // mkstring (6)` gives an Object_identifier of two elements, 8 followed by 6.

- `substring`:
  The result is a copy of a part of the Object_identifier passed as first parameter. The copy starts at the index given as second parameter (Note: first Natural has index 1). The length of the copy is specified by the third parameter. It is an error to try to access elements outside of the true length of the first parameter.
  `substring(mkstring(8)//mkstring(6),2,1) =mkstring(6)`
- `append :`
  append is an IBM Rational extension and can be used to add a new component to the end of an existing Object_identifier. append takes a variable as first parameter and a Natural value as second. The variable is then updated to include the second parameter as last component in the Object_identifier. The reason for introducing this operator is that:
  `task append(V, 12);`
  is much more efficient than performing the same calculation as
  `task V := V // mkstring(12);`

Caution!

The `append` operator does not check the size constraints on the string.

The `concat` operator should be used instead if you want range checks to be performed.

- `(. .)`:
  The `(. .)` expression, which is an IBM Rational extension, is an application of the implicit make operator. The make operator takes a sequence of Natural values and returns an Object_identifier that contains these value in the order they are given.
  `Obj_id_var := (. 1, 2, 3 .)` would give an Object_identifier containing 1, 2 and 3.

It is also possible to access the Natural elements in an Object_identifier by indexing an Object_identifier variable. Assume that `C` is a Object_identifier variable. Then it is possible to write:

```
task C(2) := C(3);
```

This would mean that the Natural at index 2 is assigned the value of the Natural at index 3 in the variable `C`. Note that the first Natural in an

Object_identifier has index 1. It is an error to index an Object_identifier outside of its length.

Octet

The Z.105-specific sort `Octet` is used to represent eight-bit values, i.e. values between 0 and 255. In C this would correspond to unsigned char. There are no explicit literals for the Octet sort. Values can, however, easily be constructed using the conversion operators `i2o` and `o2i` discussed below.

The following operators are defined in Octet:

```
"not"     : Octet                    -> Octet;
"and"     : Octet, Octet             -> Octet;
"or"      : Octet, Octet             -> Octet;
"xor"     : Octet, Octet             -> Octet;
"=>"      : Octet, Octet             -> Octet;
"<"       : Octet, Octet             -> Boolean;
"<="      : Octet, Octet             -> Boolean;
">"       : Octet, Octet             -> Boolean;
">="      : Octet, Octet             -> Boolean;
shiftl    : Octet, Integer           -> Octet;
shiftr    : Octet, Integer           -> Octet;
"+"       : Octet, Octet             -> Octet;
"-"       : Octet, Octet             -> Octet;
"*"       : Octet, Octet             -> Octet;
"/"       : Octet, Octet             -> Octet;
"mod"     : Octet, Octet             -> Octet;
"rem"     : Octet, Octet             -> Octet;
i2o       : Integer                  -> Octet;
o2i       : Octet                    -> Integer;
bitstr    : Charstring               -> Octet;
hexstr    : Charstring               -> Octet;
```

The interpretation of these operators is as follows:

- `not`, `and`, `or`, `xor`, `=>` :
  Apply the corresponding Bit operator for each of the eight bits in the Octet. For example:
  `not bitstr ('00110101') = bitstr ('11001010')`

- `<`, `<=`, `>`, `>=` :
  Ordinary relation operators for the Octet values.

- `shiftl`, `shiftr` :
  These IBM Rational-specific operators are defined as left and right shift in C, so `shiftl(a,b)` is defined as `a<<b` in C.

```
shiftl (bitstr('1'), 4) = bitstr('10000')
shiftr (bitstr('1010'), 2) = bitstr ('10')
```

- `+`, `-`, `*`, `/`, `mod`, `rem` :
  These operators are the mathematical corresponding operators. All operations are, however, performed modulus 256.
  `i2o(250) + i2o(10) = i2o(4), o2i(i2o(4)-i2o(6)) = 254`

- `i2o` :
  This IBM Rational-specific operator converts an Integer value to the corresponding Octet value.
  `i2o (128) = hexstr ('80')`

- `o2i` :
  This IBM Rational-specific operator converts an Octet value to the corresponding Integer value.
  `o2i (hexstr ('80')) = 128`

- `bitstr` :
  This IBM Rational-specific operator converts a charstring containing eight Bit values (“0” and “1”) to an Octet value.
  `bitstr('00000011') = i2o(3)`

- `hexstr` :
  This IBM Rational-specific operator converts a charstring containing two HEX values (“0”-“9”, “a”- “f”, “A”- “F”) to an Octet value.
  `hexstr('01') = i2o(1), hexstr('ff') = i2o(255)`

It is also possible to read the individual bits in an Octet value by indexing an Octet variable. The index should be in the range 0 to 7.

Octet_string

The Z.105-specific sort `Octet_string` represents a sequence of `Octet` values. There is no limit on the length of the sequence. The operators defined in the Octet_string sort are:

```
mkstring     : Octet          -> Octet_string;
length       : Octet_string   -> Integer;
first        : Octet_string   -> Octet;
last         : Octet_string   -> Octet;
"//"         : Octet_string, Octet_string
                              -> Octet_string;
substring    : Octet_string, Integer, Integer
                              -> Octet_string;
bitstr       : Charstring     -> Octet_string;
hexstr       : Charstring     -> Octet_string;
```

```
bit_string   : Octet_string   -> Bit_string;
octet_string : Bit_string     -> Octet_string;
```

These operators are defined as follows:

- `mkstring` :
  This operator takes an Octet value and converts it to a Octet_string of length 1.
  `mkstring (i2o(10))` gives an Octet_string containing one element.

- `length` :
  The number of Octets in the Octet_string passed as parameter.
  `length (i2o (8)//i2o (6)) = 2`
  `length ( hexstr ('0f3d88')) = 3`
  `length ( bitstr ('')) = 0`

- `first` :
  The value of the first Octet in the Octet_string passed as parameter. If the length of the Octet_string is 0, then it is an error to call the first operator.
  `first ( hexstr ('0f3d88')) = hexstr('0f') (= i2o(15))`

- `last` :
  The value of the last Octet in the Octet_string passed as parameter. If the length of the Octet_string is 0, then it is an error to call the last operator.
  `last ( hexstr ('0f3d88')) = hexstr('88') (= i2o(136))`

- `//` (concatenation) :
  The result is an Octet_string with all the elements in the first parameter, followed by all the elements in the second parameter.
  `hexstr('0f3d')//hexstr('884F') = hexstr(''0f3d884f')`

- `substring` :
  The result is a copy of a part of the Octet_string passed as first parameter. The copy starts at the index given as the second parameter. The length of the copy is specified by the third parameter. It is an error to try to access elements outside of the true length of the first parameter.
  `substring(hexstr('0f3d889C'), 3, 2) = hexstr('889c')`

- `bitstr` :
  This IBM Rational-specific operator converts a charstring contain-

ing only characters 0 and 1, to an Octet_string with an appropriate length and with the Octet elements set to the value defined in the sequences of eight bits. If the Charstring length is not a multiple of eight, it is padded with zeros.
`bitstr ('101') = bitstr ('10100000')`

- `hexstr` :
  This IBM Rational-specific operator converts a charstring containing HEX values (0 -9, A-F, a-f) to an Octet_string. Each pair of HEX values are converted to one Octet element in the Octet_string. If the Charstring length is not a multiple of two, it is padded with a zero.
  `hexstr ('f') = hexstr ('f0')`
- `bit_string` and `octet_string` :
  These two operators convert values between Bit_string and Octet_string.

It is also possible to access the Octet elements in an Octet_string by indexing an Octet_string variable. Assume that C is an Octet_string variable. Then it is possible to write:

```
task C(2) := C(3);
```

This would mean that the Octet at index 2 is assigned the value of Octet at index 3 in the variable C. It is an error to index an Octet_string outside of its length.

Note:
The first Octet in an Octet_string has index 1.

Pid

The sort `Pid` is used as a reference to process instances. Pid has only one literal, `Null`. All other values are obtained from the SDL predefined variables `Self`, `Sender`, `Parent`, and `Offspring`.

Real

`Real` is used to represent the mathematical real values. In an implementation there are of course always restrictions in size and precision of such values. Examples of Real literals:

```
2.354  0.9834  23  1000023.001
```

The operators defined in the Real sort are:

```
"-"  : Real         -> Real;
"+"  : Real, Real   -> Real;
"-"  : Real, Real   -> Real;
"*"  : Real, Real   -> Real;
"/"  : Real, Real   -> Real;
"<"  : Real, Real   -> Boolean;
">"  : Real, Real   -> Boolean;
"<=" : Real, Real   -> Boolean;
">=" : Real, Real   -> Boolean;
```

All these operators have their ordinary mathematical meaning.

User Defined Sorts

All the predefined sorts and syntypes discussed in the previous section can be directly used in, for example, variable declarations. In many circumstances it is however suitable to introduce new sorts and syntypes into a system to describe certain properties of the system. A user-defined sort or syntype can be used in the unit where it is defined, and also in all its subunits.

Syntypes

A *syntype* definition introduces a new type name which is fully compatible with the base type. This means that a variable of the syntype may be used in any position where a variable of the base type may be used. The only difference is the range check in the syntype. One exception exists. The actual parameter that corresponds to a formal in/out parameter must be of the same syntype as the formal parameter. Otherwise proper range tests cannot be performed.

Syntypes are useful for:

- Introducing a new name for an existing type
- Introducing a new type that has the same properties as an existing type, but with a restricted value range
- Defining index sorts used in arrays

Example 4: Syntype definition

```
syntype smallint = integer
  constants 0:10
endsyntype;
```

In this example `smallint` is the new type name, `integer` is the base type, and `0:10` is the range condition. Range conditions can be more complex than the one above. It may consist of a list of conditions, where each condition can be (assume `X` to be a suitable value):

• `=X`	a single value `X` is allowed
• `X`	same as `=X`
• `/=X`	all values except `X` are allowed
• `>X`	all values `>X` are allowed
• `>=X`	all values `>=X` are allowed
• `<X`	all values `<X` are allowed
• `<=X`	all values `<=X` are allowed
• `X:Y`	all values `>=X` and `<=Y` are allowed

Example 5: Syntype definition

```
syntype strangeint = integer
  constants <-5, 0:3, 5, 8, >=13
endsyntype;
```

In this example all values `<-5, 0, 1, 2, 3, 5, 8, >=13` are allowed.

The range check introduced in a syntype is tested in the following cases (assuming that the variable, signal parameter, formal parameter involved is defined as a syntype):

- Assignment to a variable
- Assigning a value to a signal parameter in an output (also for the implicit signals used in connection with import and remote procedure calls)
- Assigning a value to an `IN` parameter in a procedure call
- Assigning a value to a process parameter in a create request action
- Assigning a value to a variable in an input
- Assigning a value to an operator parameter (also for the operator result)
- Assigning a value to a timer parameter in set, reset, or active

Enumeration Sorts

An *enumeration sort* is a sort containing only the values enumerated in the sort. If some property of the system can take a relatively small number of distinct values and each value has a name, an enumeration sort is probably suitable to describe this property. Assume for example a key with three positions; off, stand-by, and service-mode. A suitable sort to describe this would be:

Example 6: Enumeration sort

```
newtype KeyPosition
  literals Off, Stand_by, Service_mode
endnewtype;
```

A variable of sort `KeyPosition` can take any of the three values in the literal list, but no other.

Struct

The *struct* concept in SDL can be used to make an aggregate of data that belongs together. Similar features can be found in most programming languages. In C, for example, it is also called struct, while in Pascal it is the record concept that has these properties. If, for example, we would like to describe a person and would like to give him a number of properties or attributes, such as name, address, and phone number, we can write:

```
newtype Person struct
  Name        Charstring;
  Address     Charstring;
  PhoneNumber Charstring;
endnewtype;
```

A struct contains a number of components, each with a name and a type. If we now define variables of this struct type,

```
dcl p1, p2 Person;
```

it is possible to work directly with complete struct values, like in assignments, or in tests for equality. Also individual components in the struct variable can be selected or changed.

```
task p1 := (. 'Peter', 'Main Road, Smalltown',
              '+46 40 174700' .);
task BoolVar := p1 = p2;
task p2 ! Name := 'John';
task CharstringVar := p2 ! Name;
```

The first task is an assignment on the struct level. The right hand side, i.e. the (. .) expression, is an application of the implicit make operator, that is present in all structs. The make operator takes a value of the first component sort, followed by a value of the second component sort, and so on, and returns a struct value where the components are given the corresponding values. In the example above, the component `Name` in variable `p1` is given the value `'Peter'`. The second task shows a test for equality between two struct expressions. The third and fourth task shows how to access a component in a struct. A component is selected by writing:

```
VariableName ! ComponentName
```

Such component selection can be performed both in a expression (then usually called *extract*) and in the left hand side of an assignment (then usually called *modify*).

Bit Fields

A *bit field* defines the size in bits for a struct component. This feature is not part of the SDL Recommendation, but rather introduced by IBM Rational to enable the generation of C bit fields from SDL. This means that the syntax and semantics of bit fields follow the C counterpart very much.

Example 7: Bit fields

```
newtype example struct
  a Integer     : 4;
  b UnsignedInt : 2;
  c UnsignedInt : 1;
                : 0;
  d Integer     : 4;
  e Integer;
endnewtype;
```

The following rules apply to bit fields:

- The meaning of the bit field size, i.e. the `: X` (where `X` is an integer number) is the same as in C. When generating C code from SDL, the `: X` is just copied to the C struct that is generated from the SDL struct.
- `: 0` in SDL is translated to `int : 0` in C.
- As C only allows `int` and `unsigned int` for bit field components the same rule is valid in SDL: only `Integer` and `UnsignedInt` (from package `ctypes`) may be used.

Bit fields should only be used when it is necessary to generate C bit fields from SDL. Bit fields should not be used as an alternative to syntypes with a constants clause; the SDL Suite does not check violations to the size of the bit fields.

Optional and Default values

To simplify the translation of ASN.1 types to SDL sorts, two new features have been introduced into structs. Struct components can be *optional* and they can have *default values*. Note that these features have their major application area in connection with ASN.1 data types and applying them in other situations is probably not a good idea, as they are not standard SDL-96.

Example 8: Optional and default values

```
newtype example struct
  a Integer     optional;
  b Charstring;
  c Boolean     := true;
  d Integer     := 4;
  e Integer     optional;
endnewtype;
```

The default values for component `c` and `d`, means that these components are initialized to the given values.

An *optional* component may or may not be present in a struct value. Initially an optional component is not present. It becomes present when it is assigned a value. It is an error to access a component that is not present. It is possible to test if an optional component is present or not by calling an implicit operator called

`ComponentName``present`

In the example above `apresent(v)` and `epresent(v)` can be used to test whether components `a` and `e` are present or not, in the value stored in variable `v`. A component that is present can be set to absent, i.e. not present, again by calling the implicit operator

`ComponentName``absent`

In the example above `aabsent(v)` and `eabsent(v)` can be used to set the components to absent. Note that the absent operators are operators without result.

Components with default values also have `present` and `absent` operators in the same way as optional components. They however do not have the same semantics as for optional components. A component with default value always has a value! Present and absent instead have to do with encoding and decoding of ASN.1 values. A component that contains its default value, i.e. is absent, is in some encoding schemes not encoded.

A component with default value is initialized with the default value and has present equal to false. Present can for components with default values be seen as “is explicitly assigned some value”. This means that when a component with default value is assigned a value, in an assignment for example, present will become true (even if the component is assigned the default value). The absent operator can be used to set the component back to absent. This means that the absent operator performs two things: assigns the component the default value and sets present to false.

According to Z.105, the make operator for a struct does not include components that are optional or contain a default value. Optional components always become absent and components with default values are always initialized with their default values. The `struct example` in the previous example only contains one component that is not optional and does not contain a default value. This means that a variable `v` of this type can be assigned a struct value by:

```
task v := (. ‘hello’ .);
```

If we want to set the other components, this have to be performed in a sequence of assignments after this assignment.

To simplify assigning a complete struct value to a struct in these cases, IBM Rational provide an alternative interpretation of make for a struct. You specify that you want to use this alternative interpretation of make by selecting *Generate > Analyze > Details > Semantic Analysis > Include optional fields in make operator*.

The alternative make always takes all components as parameters. By inserting an empty position you can specify that you want the component not present or given its default value. By giving a value you specify the value to be assigned to that component. Using the example above again it is possible to write:

```
task v := (. 1, ‘hello’, , 10, .);
```

This means that the first, second, and fourth components are given explicit values, while the third and fifth becomes absent.

Choice

The new concept *choice* is introduced into SDL as a means to represent the ASN.1 concept CHOICE. This concept can also be very useful while developing pure SDL data types. The choice in SDL can be seen as a C *union* with an implicit tag field.

Example 9: Choice

```
newtype C1 choice
  a Integer;
  b Charstring;
  c Boolean;
endnewtype;
```

The example above shows a choice with three components. The interpretation is that a variable of a choice type can only contain one of the components at a time, so in the example above a value of `C1` either contains an Integer value, a Charstring value, or a Boolean value.

Example 10: Working with a choice type

```
DCL var C1, charstr Charstring;

TASK var := a : 5; /* assign component a */
TASK var!b := 'hello'; /* assign component b
                           (a becomes absent) */
TASK charstr := var!b; /* get component b */
```

The above example shows how to modify and extract components of a choice type. In this respect, choice types are identical to struct types, except the `a:5` notation to denote choice values, whereas struct values are described using (. ... .).

Extracting a component of a choice type that is not present results in a run-time error. Therefore it is necessary to be able to determine which component is active in a particular value. For that purpose there are a number of implicit operators defined for a choice.

```
var!present
```

where `var` is a variable of a choice type, returns a value which is the name of the active component. This is made possible by introducing an implicit enumeration type with literals with the same names as the choice components. Note that this enumeration type is implicit and should not be inserted by you. Given the example above, it is allowed to test:

```
var!present = b
```

This is illustrated in Figure 26.

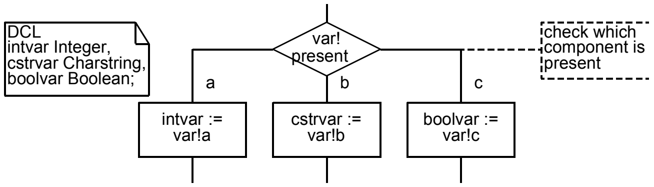


Figure 26: Check which component of a choice is present

It is also possible to test if a certain component is active or not, by using the implicit boolean operators `ComponentNamepresent`. To check if component `b` in the example above is present it is thus possible to write:

```
bpresent(v)
```

The information about which component that is active can be accessed using the `present` operators, but it is not possible to change it. This information is automatically updated when a component in a choice variable is assigned a value.

The purpose of choice is to save memory or bandwidth. As it is known that only one component at a time can contain a value, the compiler can use overlay techniques to reduce the total memory for the type. Also sending a choice value over a physical connection saves time, compared to sending a corresponding struct.

The choice construct is IBM Rational-specific, and not part of recommendation Z.105, so if you want to write portable SDL, you should not use choice. Choice replaces the SDL Suite #UNION code generator directive. It is recommended to replace #UNION directives by choice, as the SDL Suite has better tool support for the latter.

Inherits

It is possible to create a new sort by *inheriting* information from another sort. It is possible to specify which operators and literals that should be inherited and it is then possible to add new operators and literals in the new type.

Note that it is not really possible to change the type in itself by using inheritance. It is, for example, not possible to add a new component to a struct when the struct is inherited.

Our experience with inheritance so far has been that it is not as useful as it might seem in the beginning, and that sometimes the use of inheritance leads to the need of qualifiers in a lot of places, as many expressions are no longer semantically valid.

Example 11: Inherits

```
newtype NewInteger inherits Integer
  operators all;
endnewtype;
```

In the example above a new type `NewInteger` is introduced. This type is distinct from Integer, i.e. an Integer expression or variable is not allowed where a `NewInteger` is expected, and a `NewInteger` expression or variable is not allowed where an Integer is expected. Since in the example all literals and operators are inherited, all the integer literals 0, 1, 2, ..., are also available as `NewInteger` literals. For operators it means that all operators having Integer as parameter or result type are copied, with the Integer parameter replaced with a `NewInteger` parameter. This is true for all operators, not only those defined in the Integer sort, which may give unexpected effects, which will be illustrated below.

Example 12: Inherited operators

The following operators are some of the operators having Integer as parameter or result type:

```
"+" : Integer, Integer -> Integer;
"-" : Integer -> Integer;
"mod" : Integer, Integer -> Integer;
length : Charstring -> Integer;
```

The type `NewInteger` defined above will inherit these and all the others having integer as parameter or result type. Note that length is defined in the Charstring sort.

```
"+" : NewInteger, NewInteger -> NewInteger;
"-" : NewInteger -> NewInteger;
"mod" : NewInteger, NewInteger -> NewInteger;
length : Charstring -> NewInteger;
```

With this `NewInteger` declaration, statements like

```
decision length(Charstring_Var) > 5;
```

are no longer correct in the SDL system. It is no longer possible to determine the types in the expression above. It can either be the length returning integer that is tested against an integer literal, or the length returning a `NewInteger` value that is tested against a `NewInteger` literal.

It is possible to avoid this kind of problem by specifying explicitly the operators that should be inherited.

Example 13: Inherits

```
newtype NewInteger inherits Integer
  operators ("+", "-", "*", "/")
endnewtype;
```

Now only the enumerated operators are inherited and the problem with `length` that was discussed above will not occur.

A newtype which inherits another type does not inherit the default value from the original type.

Predefined Generators

Array

The predefined generator `Array` takes two generator parameters, an index sort and a component sort. There are no restrictions in SDL on the index and component sort.

Example 14: Array instantiation

```
newtype A1 Array(Character, Integer)
endnewtype;
```

The example above shows an instantiation of the Array generator with Character as index sort and Integer as component sort. This means that we now have created a data structure that contains one Integer value for each possible Character value. To obtain the component value connected to a certain index value it is possible to index the array.

Example 15: Using an array type

```
dcl Var_A1 A1;  /* Assume sort in example above */

task Var_A1 := (. 3 .);
task Var_Integer := Var_A1('a');
task Var_A1('x') := 11;

decision Var_A1 = (. 11 .);
  (true) : ...
  ...
enddecision;
```

The example above shows how to work with arrays. First we have the expression `(. 3 .)`. This is an application of the *make!* operator defined in all array instantiations. The purpose is to return an array value with all components set to the value specified in make. The first task above thus assigns the value 3 to all array components. Note that this is an assignment of a complete array value.

In the second task the value of the array component at index `'a'` is extracted and assigned to the integer variable `Var_Integer`. In the third task the value of the array component at index `'x'` is modified and given the new value `11`. The second and third task show applications of the operators *extract!* and *modify!* which are present in all array instantiations. Note that the operators extract!, modify!, and make! can only be used in the way shown in the example above. It is not allowed to directly use the name of these operators.

In the last statement, the decision, an equal test for two array values is performed. Equal and not equal are, as well as assignment, defined for all sorts in SDL.

The typical usage of arrays is to define a fixed number of elements of the same sort. Often a syntype of Integer is used for the index sort, as in the following example, where an array of 11 Pids is defined with indices 0 to 10.

Example 16: Typical array definition

```
syntype indexsort = Integer
  constants 0:10
endsyntype;

newtype PidArray Array (indexsort, Pid)
endnewtype;
```

Unlike most ordinary programming languages, there are no restrictions on the index sort in SDL. In most programming languages the index type must define a finite range of values possible to enumerate. In C, for example, the size of an array is specified as an integer constant, and the indices in the array range from 0 to the (size-1). In SDL, however, there are no such limits.

Example 17: Array with infinite number of elements.

```
newtype RealArr Array (Real, Real)
endnewtype;
```

Having Real as index type means that there is an infinite number of elements in the array above. It has, however, the same properties as all other arrays discussed above. This kind of more advanced arrays sometimes can be a very powerful concept that can be used for implementing, for example, a mapping table between different entities.

Example 18: Array to implement a mapping table

```
newtype CharstringToPid Array (Charstring, Pid)
endnewtype;
```

The above type can be used to map a Charstring representing a name to a Pid value representing the corresponding process instance.

String

The `String` generator takes two generator parameters, the component sort and the name of an empty string value. A value of a String type is a sequence of component sort values. There is no restriction on the length of the sequence. The predefined sort Charstring, for example, is defined as an application of the String generator.

Example 19: String generator

```
newtype S1 String(Integer, empty)
endnewtype;
```

Above, a String with Integer components is defined. An empty string, a string with the length zero, is represented by the literal `empty`.

The following operators are available in instantiations of String.

```
mkstring  : Itemsort                   -> String
length    : String                     -> Integer
first     : String                     -> Itemsort
last      : String                     -> Itemsort
"//"      : String, String             -> String
substring : String, Integer, Integer -> String
append    : in/out String, Itemsort;
(. .)     : * Itemsort                 -> String
```

In this enumeration of operators, String should be replaced by the string newtype (`S1` in the example above) and Itemsort should be replaced by the component sort parameter (Integer in the example above). The operators have the following behavior, with the examples based on type `String (Integer, empty)`:

- `mkstring` :
  This operator takes one Itemsort value and converts it to a String of length 1.
  `mkstring (-3)` gives a string of one integer with value -3.

- `length` :
  The number of elements, i.e. Itemsort values, in the String passed as parameter.
  `length (empty) = 0, length(mkstring (2)) = 1`

- `first` :
  The value of the first Itemsort element in the String passed as parameter. If the length of the String is 0, then it is an error to call the first operator.
  `first (mkstring (8) // mkstring (2)) = 8`

- `last` :
  The value of the last Itemsort element in the String passed as parameter. If the length of the String is 0, then it is an error to call the last operator.
  `last (mkstring (8) // mkstring (2)) = 2`

- `//` (concatenation) :
  The result is a String with all the elements in the first parameter, followed by all the elements in the second parameter.
  `mkstring (8) // mkstring(2)` gives a string of two elements: 8 followed by 2.

- `substring`:
  The result is a copy of a part of the String passed as first parameter. The copy starts at the index given as second parameter (Note: first Itemsort element has index 1). The length of the copy is specified by the third parameter. It is an error to try to access elements outside of the true length of the first parameter.
  ```
  substring (mkstring (8) // mkstring(2), 2, 1)
  = mkstring(2)
  ```
- `append :`
  append is an IBM Rational extension and can be used to add a new component to the end of an existing String. append takes a variable as first parameter and a Itemsort value as second. The variable is then updated to include the second parameter as last component in the String. The reason for introducing this operator is that:
  ```
  task append(V, Comp);
  ```
  is much more efficient than performing the same calculation as
  ```
  task V := V // mkstring(Comp);
  ```
- `(. .)`:
  The `(. .)` expression, which is an IBM Rational extension, is an application of the implicit make operator that is present in all strings. The make operator takes a sequence of Itemsort values and returns a String that contains these value in the order they are given. `String_var := (. 1, 2, 3 .)` would give a string containing 1, 2 and 3.

It is also possible to access Itemsort elements in a String by indexing a String variable. Assume that C is a String instantiation variable. Then it is possible to write:

```
task C(2) := C(3);
```

This would mean that Itemsort element number 2 is assigned the value of Itemsort element number 3 in the variable C. NOTE that the first element in a String has index 1. It is an error to index a String outside of its length.

The String generator can be used to build lists of items of the same type, although some typical list operations are computationally quite expensive, like inserting a new element in the middle of the list.

Powerset

The `Powerset` generator takes one generator parameter, the item sort, and implements a powerset over that sort. A Powerset value can be seen as: for each possible value of the item sort it indicates whether that value is member of the Powerset or not.

Powersets can often be used as an abstraction of other, more simple data types. A 32-bit word seen as a bit pattern can be modeled as a Powerset over a syntype of Integer with the range 0:31. If, for example, 7 is member of the powerset this means that bit number 7 is set.

Example 20: Powerset generator

```
syntype SmallInteger = Integer
  constants 0:31
endsyntype;

newtype P1 Powerset(SmallInteger)
endnewtype;
```

The only literal for a powerset sort is `empty`, which represents a powerset containing no elements. The following operators are available for a powerset sort (replace `Powerset` with the name of the newtype, `P1` in the example above, and `Itemsort` with the Itemsort parameter, `SmallInteger` in the example):

```
"in"   : Itemsort,  Powerset  -> Boolean
incl   : Itemsort,  Powerset  -> Powerset
incl   : Itemsort, in/out Powerset;
del    : Itemsort,  Powerset  -> Powerset
del    : Itemsort, in/out Powerset;
length : Powerset             -> Integer
take   : Powerset             -> Itemsort
take   : Powerset,  Integer   -> Itemsort
"<"    : Powerset,  Powerset  -> Boolean
">"    : Powerset,  Powerset  -> Boolean
"<="   : Powerset,  Powerset  -> Boolean
">="   : Powerset,  Powerset  -> Boolean
"and"  : Powerset,  Powerset  -> Powerset
"or"   : Powerset,  Powerset  -> Powerset
(. .)  : * Itemsort           -> Powerset
```

These operators have the following interpretation (the examples are based on newtype `P1` of the above example, and it is supposed that variable `v0_1_2` of `P1` contains elements 0, 1, and 2):

- `in` :
  This operator tests if a certain value is member of the powerset or not.
  `3 in incl (3, empty)` gives `true`;
  `3 in v0_1_2` gives `false`, `0 in v0_1_2` gives `true`.

- `incl` :
  Includes a value in the powerset. The result is a copy of the Powerset parameter with the Itemsort parameter included. To include a value that is already member of a powerset is a null-action.
  `incl (3, empty)` gives a set with one element, 3,
  `incl (3, v0_1_2)` gives a set with elements, 0, 1, 2, and 3.

- `incl` (second operator) :
  This operator is an IBM Rational extensions added as it is more efficient than the standard incl. This operator updates a powerset variable with a new component value.
  `task incl(3, v0_1_2);` means the same as
  `task v0_1_2 := incl(3, v0_1_2);`

- `del` :
  Deletes a member in a powerset. The result is a copy of the Powerset parameter with the Itemsort parameter deleted. To delete a value that is not member of a powerset is a null-action.
  `del (0, v0_1_2)` gives a set with element 1 and 2;
  `del (30, v0_1_2) = v0_1_2`

- `del` (second operator) :
  This operator is an IBM Rational extensions added as it is more efficient than the standard del operator. This operator updates a powerset variable by removing a component value.
  `task del(3, v0_1_2);` means the same as
  `task v0_1_2 := del(3, v0_1_2);`

- `length` :
  The number of elements in the powerset.
  `length (v0_1_2) = 3, length (empty) = 0`

- `take` (one parameter) :
  Returns one of the elements in the powerset, but it is not specified which one.
  `take (v0_1_2)` gives 0, 1, or 2 (unspecified which of these three)
- `take` (two parameters) :
  Elements are implicitly numbered with in the powerset from 1 to length(). The IBM Rational-specific take operator returns the element with the number passed as second parameter. This operator can be used to “loop” through all elements of the set, as is illustrated in Figure 27.

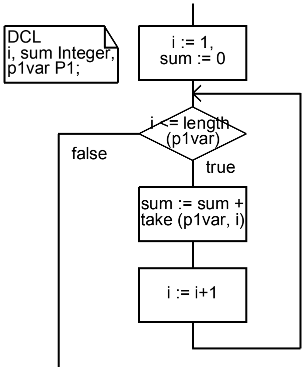


Figure 27: Computing the sum of all elements in a Powerset

- `<` :
  A<B, is A a true subset of B
  `incl (2, empty) < v0_1_2 = true,`
  `incl (30, empty) < v0_1_2 = false`
- `>` :
  A>B, is B a true subset of A
- `<=` :
  A<=B, is A a subset of B
- `>=` :
  A>=B, is B a subset of A

- `and` :
  Returns the intersection of the parameters, i.e. a powerset with the element members of both parameters.
  `incl (2, incl (4, empty)) and v0_1_2` gives a set with one element, visually 2.

- `or` :
  Returns the union of the parameters, i.e. a powerset with the element members of any of the parameters.
  `incl (2, incl (4, empty)) or v0_1_2` gives a set with elements, 0, 1, 2, and 4.

- `(. .)`:
  The `(. .)` expression, which is an IBM Rational extension, is an application of the implicit make operator, that is present in all powersets. The make operator takes a sequence of Itemsort values and returns a Powerset that contains these values.
  `v0_1_2 := (. 1, 2, 3 .)` would give a set including 1, 2 and 3.

Powerset resembles the Bag operator, and normally it is better to use Powerset. See also the discussion in "Bag" on page 79.

Bag

The Z.105-specific generator `Bag` is almost the same as Powerset. The only difference is that a bag can contain the same value several times. In a Powerset a certain value is either member or not member of the set. A Bag instantiation contains the literal `empty` and the same operators, with the same behavior, as a Powerset instantiation. For details please see "Powerset" on page 76.

A Bag contains one additional operator:

```
makebag : Itemsort      -> Bag
```

- `makebag` :
  Takes an Itemsort value and returns a Bag containing this value (length = 1).

It is recommended to use Powerset instead of Bag, except in cases where the number of instances of a value is important. Powerset is defined in Z.100, and is therefore more portable. Bag is mainly part of the predefined data types in order to support the ASN.1 `SET OF` construct.

Ref, Own, Oref, Carray

These generators are IBM Rational extensions to make it possible to work with pointers (Ref, Own, Oref) and with array with the same properties as in C.

Own and Oref is described in “Own and ORef Generators” on page 128 in chapter 3, *Using SDL Extensions*, while Ref and Carray is part of the package ctypes described in “C Specific Package ctypes” on page 108. The package ctypes also contains SDL versions of some simple C types, which might be helpful in some cases.

Literals

Literals, i.e. named values, can be included in newtypes.

Example 21: Literals in struct newtype

```
newtype Coordinates struct
    x integer;
    y integer;
  adding
    literals Origo, One;
endnewtype;
```

In this struct there are two named values (literals); `Origo` and `One`. The only way in SDL to specify the values these literals represent is to use axioms. Axioms can be given in a section in a newtype. This is not further discussed here. The SDL to C compilers provide other ways to insert the values of the literals. Please see the documentation in *chapter 56, The Cadvanced/Cbasic SDL to C Compiler, in the User's Manual*.

The literals can be used in SDL actions in the same way as expressions.

Example 22: Use of literals

```
dcl C1 Coordinates;

task C1 := Origo;
decision C1 /= One;
...
```

Please note the differences in the interpretation of literals in the example above and in the description of enumeration types, see “Enumeration Sorts” on page 63. In an enumeration type each literal introduces a new distinct value and the set of literals defines the possible values for the type. In the struct example above, the type and the set of possible values for the type is defined by the struct definition. The literals here only give names on already existing values.

An alternative that might be more clear, is to use literals in the case of an enumeration type and use operators without parameters (IBM Rational extension) in other cases, like the struct above.

Operators

Operators can be added to a newtype in the same way as literals.

Example 23: Operators in struct newtype

```
newtype Coordinates struct
    x integer;
    y integer;
  adding
    operators
    "+" : Coordinates, Coordinates -> Coordinates;
    length : Coordinates -> Real;
endnewtype;
```

IBM Rational has extended the operators with a number of new features to make them more flexible and to make it possible to have more efficient implementations. Extensions:

- in/out parameters
- operators without parameters
- operators without result

Example 24: Operators

```
operators
  op1 : in/out Coordinates;
  op2 : -> Coordinates;
  op3 : ;
```

In the example above `op1` takes one in/out parameter and has no result, `op2` has no parameters and returns a value of type `Coordinates`, while `op3` has neither parameters. nor result.

The behavior of operators can either be defined in axioms (as the literal values) or in operator diagrams. An operator diagram is almost identical to a value returning procedure (without states). An alternative to draw the operator implementation as a diagram is to define it in textual form. This might be appropriate as most operators performs calculations, and does not have anything to do with process control or process communication. In this case the algorithmic extension described in "Compound Statement" on page 138 in chapter 3, *Using SDL Extensions* could be of great value.

Example 25: Operator implementations

```
newtype Coordinates struct
    x integer;
    y integer;
  adding
    operators
    "+" : Coordinates, Coordinates -> Coordinates;
  operator "+" fpar a, b Coordinates
               returns Coordinates
  {
    dcl result Coordinates;
    result!x := a!x + b!x;
    result!y := a!y + b!y;
    return result;
  }
endnewtype;
```

In the SDL to C Compilers there is also the possibility to include implementations in the target language. The problem with this is that it is necessary to know a lot more about the way the SDL to C Compilers translate operators into C.

Default Value

In a newtype or syntype it is possible to insert a default clause stating the default value to be given to all variables of this type.

Example 26: Default value in struct newtype

```
newtype Coordinates struct
    x integer;
    y integer;
  default (. 0, 0 .);
endnewtype;
```

All variables of sort `Coordinates` will be given the initial value `(. 0, 0 .)`, except if an explicit default value is given for the variable in the variable declaration.

Example 27: Explicit default value in variable declaration

```
dcl
  C1 Coordinates := (. 1, 1 .),
  C2 Coordinates;
```

Here `C1` has an explicit default value that is assigned at start-up. `C2` will have the default value specified in the newtype.

A newtype which inherits another type does not inherit the default value from the original type.

Generators

It is possible in SDL to define *generators* with the same kind of properties as the pre-defined generators Array, String, Powerset, and Bag. As this is a difficult task and the support from the code generators is limited, it is not recommended for a non-specialist to try to define a generator.

The possibility to use user defined generators in the SDL to C Compilers is described in more detail in *“Generators” on page 2720 in chapter 56, The Cadvanced/Cbasic SDL to C Compiler, in the User's Manual*.

Using C/C++ in SDL

Introduction

To enable access to C or C++ declarations from an SDL specification, translation rules from C/C++ to SDL have been developed, that specify how C/C++ constructs may be represented in SDL. These translation rules have been implemented in the SDL Suite's CPP2SDL tool. CPP2SDL supports the translation of both C and C++ declarations.

When using CPP2SDL, it is possible to access C/C++ declarations and definitions in SDL. Figure 28 shows how CPP2SDL takes a set of C/C++ header files and, optionally, an import specification as input. Note that the import specification is only optional when CPP2SDL is executed from the command line. When using the utility from the Organizer, an import specification is created with a default configuration. An import specification holds CPP2SDL options, and may also specify which declarations in the header files are to be translated. CPP2SDL then translates the C/C++ declarations in the header files to SDL declarations. These resulting SDL declarations are saved in a generated SDL/PR file. See "Introduction" on page 764 in chapter 14, *The CPP2SDL Tool, in the User's Manual* for more details.

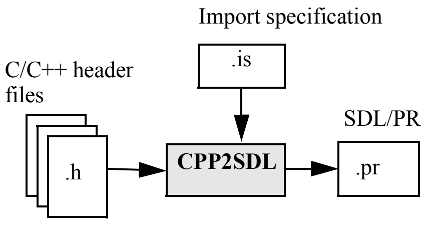


Figure 28: CPP2SDL input and output.

Workflow

The typical workflow involved when using CPP2SDL will be illustrated with an example based on the AccessControl system. The example can be found in `\IBM\Rational\SDL_TTCN_Suite6.3/sdt/exam-`

`ples/cpp_access`. Please note that the example currently runs on **Windows only**. However, the principles that are demonstrated are the same on all platforms.

The AccessControl system controls the access to a building. The building has a user terminal consisting of a display, a card reader and a keypad. To get access to the building, a valid card has to be inserted and a correct 4-digit code has to be typed.

In this version of the AccessControl system, information about cards and valid codes is stored in an external database. The database will be accessed through ODBC[1], which is a commonly used C/C++ API for accessing data from different kinds of databases.

The purpose of the example is to show how a C/C++ API can be accessed from SDL by means of the tools in the C/C++ Access. The example covers the most important issues regarding the usage of the C/C++ Access, and may serve as a basis for more advanced experiments.

The example described below is a walk-through of how to utilize CPP2SDL from within the Organizer. The different development phases illustrated are:

- A PR symbol is added to the Central process diagram.
- The PR symbol is refined to be an import specification, by double-clicking it and setting the document type to *C++ Import Specification*.
- A `TRANSLATE` section is added to the import specification, in which we list the names of all C/C++ declarations we need to access.
- The import specification is then saved in a file, and the import specification symbol is thereby automatically connected to this file.
- We use the CPP2SDL Options dialog to set various options for the import specification.
- Finally, we add a header file to be translated.

1.ODBC is a de facto standard **on Windows**, but it has also been implemented on other platforms.

Editing

The first step in accessing a C/C++ API from SDL is to insert a PR symbol at the place in the SDL specification where the C/C++ declarations of the API are to be used. The PR symbol represents the inclusion of an SDL/PR file, in general. In C/C++ Access this mechanism is used to include the SDL/PR file that is generated by CPP2SDL.

In the AccessControl example, we insert a PR symbol named ODBC in the process Central. The ODBC API is accessed from this process exclusively, thereby maintaining the narrowest possible scope.

Normally, an import specification should be placed at the highest level where declarations imported by the import specification are used. However, if C/C++ variables are imported, the import specification must be placed in a scope where external SDL variables are allowed to be declared.

Note:

External variables cannot be declared at system or block level. They can only be declared in processes, procedures, services or in operator diagrams.

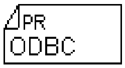
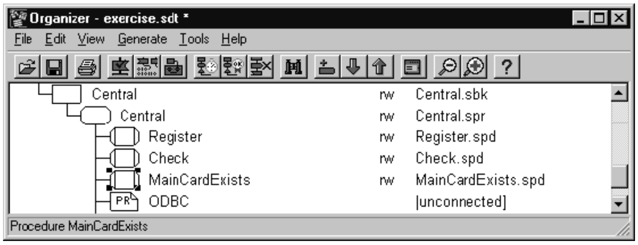


Figure 29: The PR symbol in the SDL Editor

When a PR symbol has been added in the SDL Editor it will initially appear in the Organizer as an unconnected reference, see Figure 30.


Organizer - exercise.sdt *
File Edit View Generate Tools Help

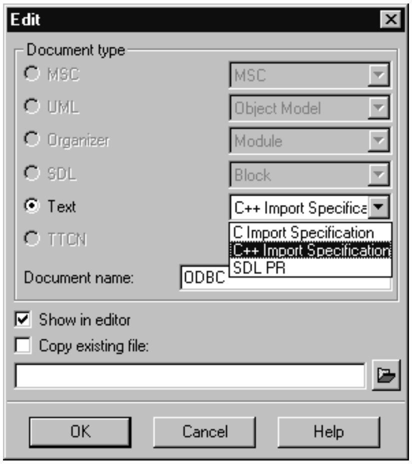
```
Central          rw  Central.sbk
Central          rw  Central.spr
Register         rw  Register.spd
Check            rw  Check.spd
MainCardExists   rw  MainCardExists.spd
ODBC                 [unconnected]
Procedure MainCardExists
```

Figure 30: The unconnected PR symbol in the Organizer

By default, the Organizer assumes that an unconnected PR symbol is to be connected to an ordinary user-defined SDL/PR file. In this case this is not what we want. By double-clicking the PR symbol, either in the SDL Editor or in the Organizer, an edit Document dialog is opened, see Figure 31 on page 87. If we change the document type from *SDL/PR* to *C++ Import Specification*, we specify that the SDL/PR file is generated from a set of C++ header files.

Note:

Ordinary PR symbols are connected to user-defined SDL/PR files, while import specification symbols are connected to generated SDL/PR files.


Edit

Document type

( ) MSC — MSC
( ) UML — Object Model
( ) Organizer — Module
( ) SDL — Block
(●) Text — C++ Import Specifica
( ) TTCN

C Import Specification
C++ Import Specification
SDL PR

Document name: ODBC

[✓] Show in editor
[ ] Copy existing file:

Figure 31: Edit the type of the PR symbol to be a C++ Import Specification

Since we want to create a new import specification, we leave the *Show in editor* check-box marked. If we already have an import specification to be used, there are two methods of connecting it. The first approach is to unmark all check-boxes and use the *Connect* command in the Organizer to connect to the existing import specification file. The second approach is to check the *Copy existing file* option and either browse for, or input the path to, the existing import specification. When using this method, it is necessary to view the file in the Text Editor, and then save in order to connect it.

An import specification can be edited manually by means of the Text Editor. However, an import specification can be left empty, and CPP2SDL options set from within the Organizer at a later stage. This will add a section called `CPP2SDLOPTIONS` where different options to CPP2SDL are stored. Often an import specification will contain a `TRANSLATE` section, with a list of the names of all declarations that you wish to be made accessible in SDL. For more information, refer to "Import Specification" on page 101.

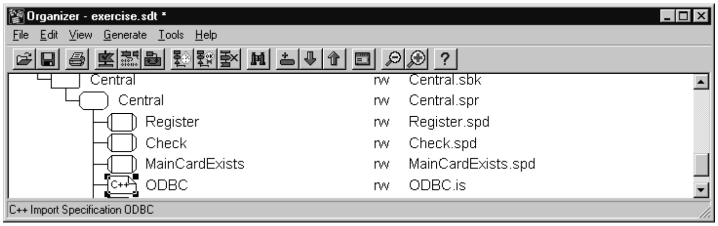
In our case we add a `TRANSLATE` section with the names of all ODBC functions and types that we will need to access from SDL. See Figure 32.

```
TRANSLATE {
  SQLHENV
  SQLHDBC
  SQLHSTMT
  SQLRETURN
  SQLCHAR
  SQLINTEGER
  SQLSMALLINT
  SQLPOINTER

  SQLAllocHandle
  SQLSetEnvAttr
  SQLSetConnectAttr
  SQLConnect
  SQLBindCol
  SQLExecDirect
  SQLFetch
  SQLCloseCursor
  SQLFreeHandle
  SQLDisconnect
  SQLGetDiagRec

  unsigned char.[6]
  unsigned char.[64]
  unsigned char.[256]

  char.[256]

  strcpy
  strcat
}
```

Figure 32 The `TRANSLATE` *section of the ODBC import specification*

When the import specification is saved to a file (called `ODBC.is`), the Organizer will automatically connect the import specification symbol to that file.

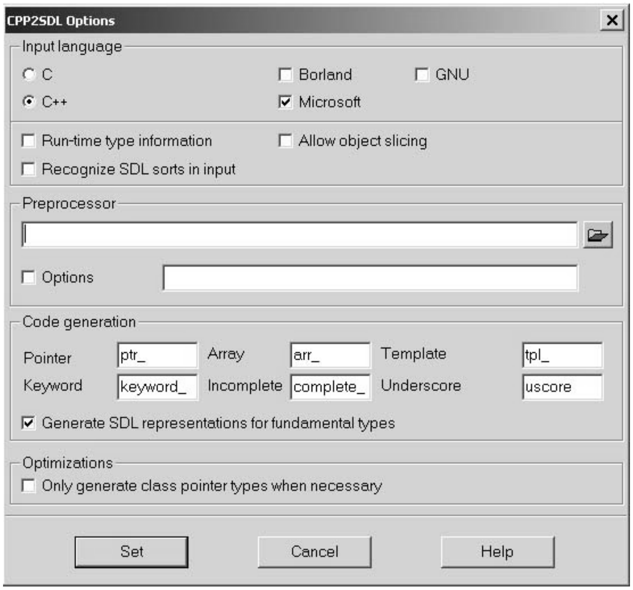
Figure 33 shows the connected import specification symbol.


Organizer - exercise.sdt *
File Edit View Generate Tools Help
C++ Import Specification ODBC

Central	rw	Central.sbk
Central	rw	Central.spr
Register	rw	Register.spd
Check	rw	Check.spd
MainCardExists	rw	MainCardExists.spd
ODBC	rw	ODBC.is

Figure 33: The connected import specification symbol in the Organizer

The next step is to set appropriate options for the translation of the C/C++ declarations that are specified in the import specification. This is best done by means of the *CPP2SDL Options* dialog (see Figure 34). This dialog is opened by right-clicking on the import specification symbol in the Organizer. For more detailed information about the CPP2SDL options, see "The CPP2SDL Tool" on page 763 in chapter 14, *The CPP2SDL Tool, in the User's Manual*.



Input language
- ( ) C
- (●) C++
- [ ] Borland
- [ ] GNU
- [✓] Microsoft
- [ ] Run-time type information
- [ ] Allow object slicing
- [ ] Recognize SDL sorts in input

Preprocessor
- [ ] Options

Code generation

Pointer	ptr_	Array	arr_	Template	tpl_
Keyword	keyword_	Incomplete	complete_	Underscore	uscore

- [✓] Generate SDL representations for fundamental types

Optimizations
- [ ] Only generate class pointer types when necessary

Figure 34: The CPP2SDL Options Dialog

The following options may be specified:

- *Language*

  The language option specifies the input language, i.e. if C or C++ declarations shall be translated. If C is selected as input language, CPP2SDL will assume that no C++ specific constructs are encountered in the input header files.

  Note that this option determines if the import specification is a C or C++ import specification. Refer to Figure 31 on page 87 where we selected which type of import specification to use.

- *Dialect*

  These check-boxes make it possible to specify what C/C++ dialects that are to be supported by CPP2SDL. If no check-boxes are marked, the ANSI C/C++ dialect is supported.

  In our example we use the ODBC implementation from the Microsoft Foundation Classes, so we need support for the Microsoft dialect.

- *Run-Time Type Information*

  If this check-box is set, Run-Time Type Information (RTTI) is assumed and dynamic casting is supported.

- *Allow Object Slicing*

  Set this check-box if generated SDL cast operators are to support slicing of C++ objects.

- *Recognize SDL Sorts in Input*

  When this check-box is set, SDL sorts will be recognized in the input.

- *Preprocessor*

  The preprocessor to be used for preprocessing the input can be set here. If no preprocessor is set, CPP2SDL will use Microsoft Visual C/C++ Compiler (`cl`) **in Windows** and the standard C/C++ Preprocessor (`cpp`) **on UNIX**.

Note:

It is normally recommended to preprocess the input C/C++ headers with a compiler rather than a plain preprocessor. The reason for this is that a compiler may set several useful preprocessor defines.

- *Preprocessor Options*

  The preprocessor options can be set in this field.

- *Pointer, Array, Template, Keyword, Incomplete, Underscore*

  These fields specify the prefixes and suffixes that are used when C/C++ names must be modified in the SDL translation.

- *Generate SDL Representations for Fundamental Types*

  Set this check-box if SDL representations for fundamental C/C++ types are to be included in the translation. These SDL representations are defined in SDL/PR files, which are described in detail in "SDL Library for Fundamental C/C++ Types" on page 850 in chapter 14, *The CPP2SDL Tool, in the User's Manual*.

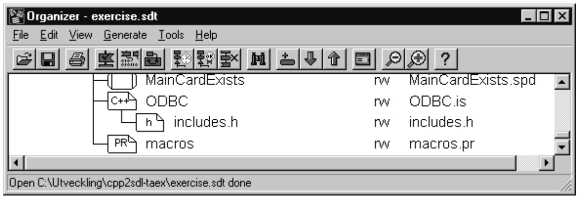
Note:

If the SDL type representation option is set at several levels, this will cause problems. SDL representations for fundamental types should only be included at the highest level at which the types will be used. For example, if two blocks in a system have import specifications for accessing C/C++ declarations, SDL representations for fundamental C/C++ types should be included in the system, and not in the blocks. This can be done by adding an empty import specification without input headers at system level, that includes the SDL representations for the fundamental C/C++ types.

- *Only Generate Class Pointer Types when Necessary*

  When this check-box is set, CPP2SDL will optimize the generation of class pointer types.

When appropriate CPP2SDL options have been set for an import specification, the next step is to add the C/C++ header files that are to be translated. This is done by selecting the import specification and, in the *Edit* menu, select *Add Existing...* Added header files will appear under the import specification symbol in the Organizer, see ***Figure 35***.



MainCardExists rw MainCardExists.spd
ODBC rw ODBC.is
includes.h rw includes.h
macros rw macros.pr

Open C:\Utveckling\cpp2sdl-taex\exercise.sdt done

Figure 35: Add header files to the import specification

An arbitrary number of header files can be added to an import specification. They will all be processed using the options that are specified for the import specification. In the AccessControl example only one header file is added (`includes.h`).

To see the contents of a header file double-click on its symbol in the Organizer. The Text Editor will then open and display the contents of the header file. If this is done on the `includes.h` header, we see that it actually includes several other header files. The reason for using a wrapper header like `includes.h` instead of adding the interesting headers under the import specification directly, is that we would like to avoid hard-coding the path to these files. By using `#include <file>` statements, and preprocessing the file with the Microsoft Visual C++ compiler, the location of these files will be known at compile-time.

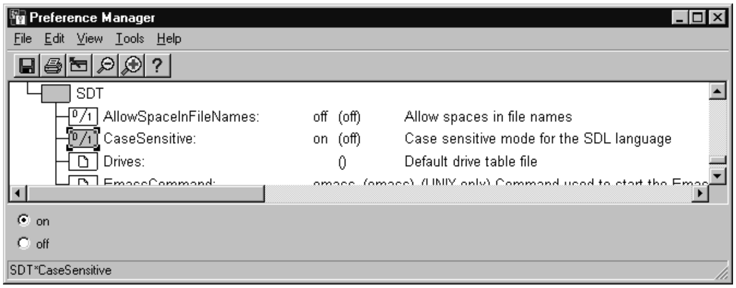
Let us summarize what we have done in the example so far. We have edited the SDL system by adding a PR symbol, changed the PR symbol to an import specification, added a `TRANSLATE` listing the needed declarations, saved the import specification connecting it to the system, configured CPP2SDL using the options dialog, and adding the header file to the system.

This concludes the editing phase. It is now time to analyze the system.

Analyzing

The SDL declarations that are generated by CPP2SDL must be analyzed as case-sensitive SDL. Before starting the Analyzer, a case-sensitivity option must therefore be set:

- Select *Tools* in the Organizer and start the *Preference Manager*.
- In the *Preference Manager*, double-click on the SDT symbol and set *CaseSensitive* to *on* (it is by default set to *off*).



SDT

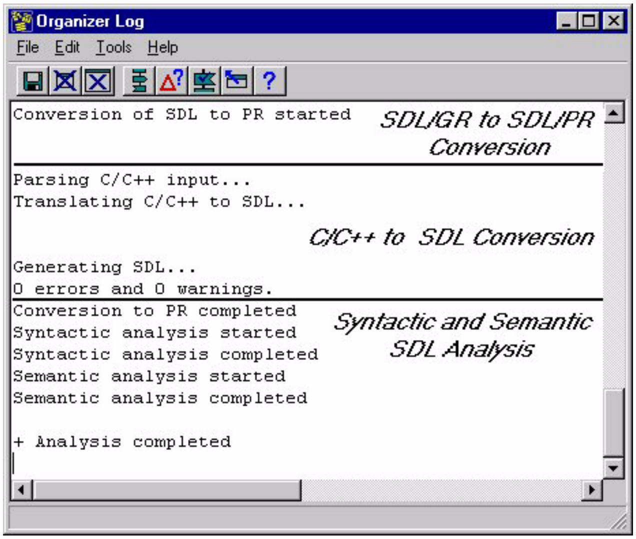
AllowSpaceInFileNames:	off (off)	Allow spaces in file names
CaseSensitive:	on (off)	Case sensitive mode for the SDL language
Drives:	()	Default drive table file

[✓ on] off

SDT*CaseSensitive

Figure 36: Set case-sensitive SDL in the Preference Manager

The Analyzer will perform three major steps during the analysis of an SDL system that contains C/C++ import specifications. During each step a message will be printed in the Organizer Log window to indicate the progress, see Figure 37.


Organizer Log
File Edit Tools Help
SDL/GR to SDL/PR Conversion
C/C++ to SDL Conversion
Syntactic and Semantic SDL Analysis

```
Conversion of SDL to PR started

Parsing C/C++ input...
Translating C/C++ to SDL...

Generating SDL...
0 errors and 0 warnings.
Conversion to PR completed
Syntactic analysis started
Syntactic analysis completed
Semantic analysis started
Semantic analysis completed

+ Analysis completed
```

Figure 37: The Organizer Log window for the analyze phase

1. **SDL/GR to SDL/PR Conversion**

   The Analyzer requests the SDL Editor to perform an SDL/GR to SDL/PR conversion. This means that all graphical SDL symbols are converted to their textual representations. In particular, every PR symbol will be represented by a `#include 'filename.pr'` in the SDL/PR, where `filename.pr` is the name of the file to which the corresponding import specification is connected.

   In our example we will thus get a `#include 'ODBC.pr'` in the SDL/PR representation of the process Central.

2. **C/C++ to SDL Conversion**

   This step is performed once for each import specification in the system. The header files associated with an import specification are parsed and analyzed by CPP2SDL. Errors that are reported during this phase may, for example, be due to differences in language support and inappropriate preprocessor settings. If so, you can set the correct language dialect and suitable preprocessor options in the CPP2SDL Options dialog. Syntax errors and some semantic errors in the header files will also be checked for during this phase. For more information about how CPP2SDL handles errors, see “Example usage of some C/C++ functionality” on page 855 in chapter 14, *The CPP2SDL Tool, in the User´s Manual*.
   If no errors are found, CPP2SDL will generate an SDL/PR file with the result of the translation. Finally, some warnings may be printed, for example to notify that certain declarations for some reason could not be translated.

   In our example we get a file called `ODBC.pr` when this step is finished.

Note:

CPP2SDL is not as good as a C/C++ compiler when it comes to error detection and error reports. It is thus strongly recommended to make sure that the header files are semantically correct by running them through a compiler, before they are translated to SDL.

3. **Syntactic and Semantic SDL Analysis**

When all SDL/PR code has been generated the SDL Analyzer will check for syntactic and semantic errors as usual. For example, it is likely that many errors will be reported if case-sensitive SDL was not set in the Preference Manager, see Figure 36. A common source for errors is that SDL representations for fundamental types were; not included at all, included at the wrong place in the SDL system, or included many times in the same SDL scope entity.

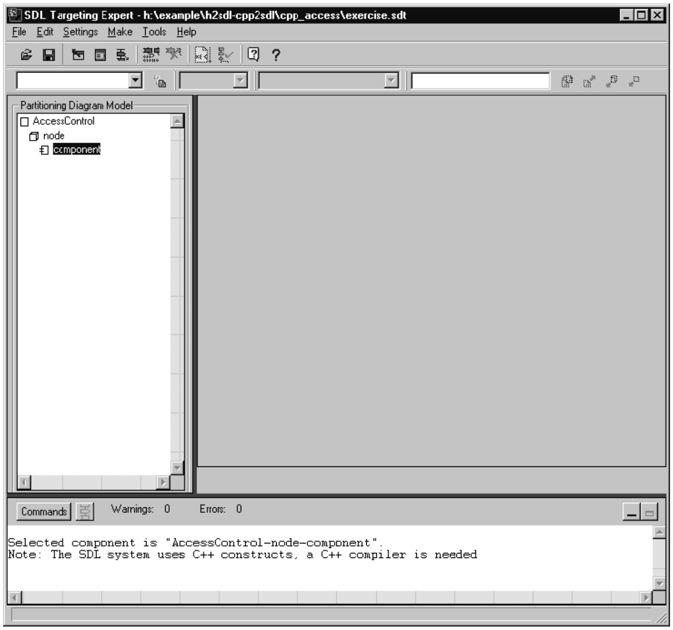
Once we have got a clean analysis of the system, it is time to proceed with code generation.

Generating

Code generation can be done either from the traditional *Make* dialog, or from the more powerful tool *SDL Targeting Expert*. To generate code for a system containing C or C++ import specifications, it is preferred to use the Targeting Expert. For example, it is much easier to link-in the object files that belong to the translated header files, using the Targeting Expert. The Make dialog will in the near future be discontinued in favor of the Targeting Expert.

A system that contains one or more C++ import specifications must be translated to C++ rather than C code. An option to the Code Generator controls whether C or C++ code is generated. This option is automatically set by the Analyzer if there are one or more C++ import specifications present in the Organizer view.

To start the Targeting Expert, select *Targeting Expert...* in the *Generate* menu in the Organizer. The Targeting Expert dialog will appear, see Figure 38. For full information about all the settings and options provided by Targeting Expert, see chapter 59, *The Targeting Expert, in the User´s Manual*.


SDL Targeting Expert - h:\example\h2sdl-cpp2sdl\cpp_access\exercise.sdt
File Edit Settings Make Tools Help
Partitioning Diagram Model
AccessControl
node
component
Commands
Warnings: 0
Errors: 0

```
Selected component is "AccessControl-node-component".
Note: The SDL system uses C++ constructs, a C++ compiler is needed
```

Figure 38: The SDL Targeting Expert

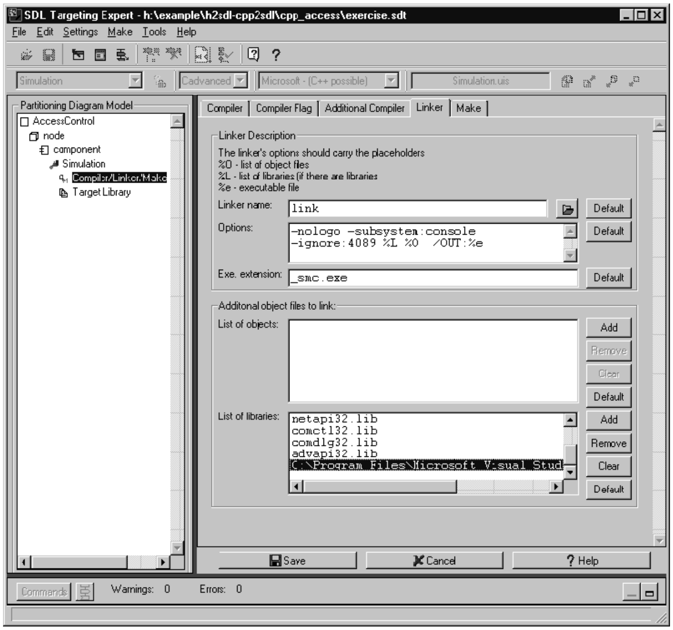
When one or more C++ import specifications are present in the SDL system, the Targeting Expert will issue a warning that a C++ compiler is needed to compile the generated code (see Figure 38). Next, right-click Component, select Simulations, and then Simulation. The compiler may be set by pressing the *Compiler/Linker/Make* icon, and then, under the *Compiler* tab, locating the compiler executable. Here we may also specify compiler options and preprocessor settings.

Note:

Make sure that the settings made in the CPP2SDL Options dialog for the preprocessor and preprocessor settings match the settings made in the Targeting Expert.

In the AccessControl example, you can use the C++ Microsoft Simulation kernel, and the generated code can be compiled with the Microsoft Visual C++ compiler.

To avoid getting loads of link errors, we also have to remember to link-in several required Microsoft libraries (e.g. the Odbc32.lib library). This is done under the *Linker* tab as shown in Figure 39. Simply add the file to the List of files and save.


SDL Targeting Expert - h:\example\h2sdl-cpp2sdl\cpp_access\exercise.sdt
File Edit Settings Make Tools Help
Simulation
Cadvanced
Microsoft - (C++ possible)
Simulation.uis
Compiler
Compiler Flag
Additional Compiler
Linker
Make
Default
Add
Remove
Clear
Save
Cancel
Help
Commands

Partitioning Diagram Model
- AccessControl
  - node
    - component
      - Simulation
        - Compiler/Linker/Make
        - Target Library

Linker Description

The linker's options should carry the placeholders
%O - list of object files
%L - list of libraries (if there are libraries
%e - executable file

Linker name: `link`

Options:
```
-nologo -subsystem:console
-ignore:4089 %L %O  /OUT:%e
```

Exe. extension: `_smc.exe`

Additonal object files to link:

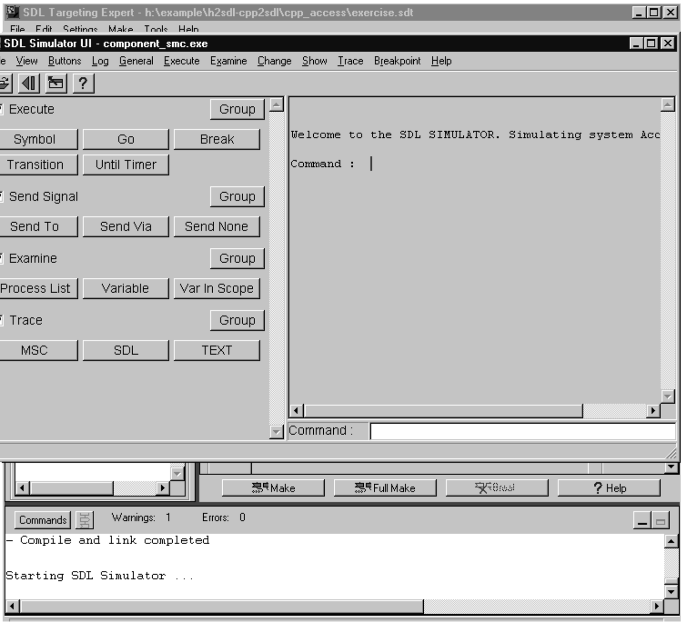
List of objects:

List of libraries:
```
netapi32.lib
comctl32.lib
comdlg32.lib
advapi32.lib
C:\Program Files\Microsoft Visual Stud
```

Warnings: 0 Errors: 0

Figure 39: Add libraries in the Targeting Expert

Now everything is ready for code generation. Press the *Make* or *Full Make* buttons and the Targeting Expert will instruct the Analyzer to analyze the SDL system (see "Analyzing" on page 92) and then invoke the C or C++ Code Generator. Finally the generated code will be compiled and linked as specified to create a simulator executable. Figure 40 shows what the Targeting Expert may look like when this has been done.



```
Welcome to the SDL SIMULATOR. Simulating system Acc

Command :
```

```
- Compile and link completed

Starting SDL Simulator ...
```

Figure 40: Generating a simulator from the Targeting Expert

Simulating

Naturally, it is possible to simulate and debug a system on SDL level even if it uses C or C++ declarations. The standard SDL simulator can be used for this.

A simulator will automatically start when making from Targeting Expert. At other times than making to start a simulation of a system, select *SDL* in the *Tools* menu in the Organizer or in the Targeting Expert. Then select *Simulator UI* and the SDL Simulator user interface will start. To load the simulator executable that was generated above, select *File* and *Open...* in the SDL Simulator UI.

For the AccessControl example, two customized buttons are available for the Simulator UI. They may be loaded by selecting *Buttons* and then *Load...* The “GUI” button starts a GUI for the AccessControl system and waits for you to single-step or go through the system by interacting with the GUI. The “GUI+MSC” button also activates the GUI, and in addition generates an MSC trace. In Figure 41 an example of an MSC trace of the AccessControl system is shown.

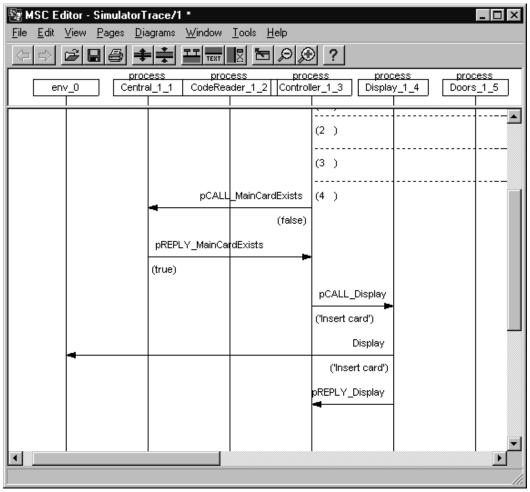


Figure 41: MSC trace of the AccessControl system

The Simulator will treat C++ classes as C structs, but with the additional possibility of invoking the constructors of the class. For example, when the value of a C++ class, that is instantiated in SDL, is changed from the Simulator, the following steps are performed:

- The Simulator pops up a dialog showing a list of available constructors. For example:

  ```
  0 /* No constructor */
  1 /* C() */
  ```
  or, for a class with a user-defined constructor,
  ```
  0 /* No constructor */
  2 /* C(int) */
  ```

  Type the number for the constructor that are to be invoked, if any.

- If a constructor was selected, the Simulator will prompt for its actual arguments.

- Finally, the Simulator allows public member variables to be explicitly set using either the SDL or ASN.1 syntax. For example:

  ```
  (. 1, true, 'x' .)              SDL syntax
  {mv1 1, mv2 true, mv3 'x'}      ASN.1 syntax
  ```
  Note that the ASN.1 syntax is more flexible since it contains the names of the member variables.

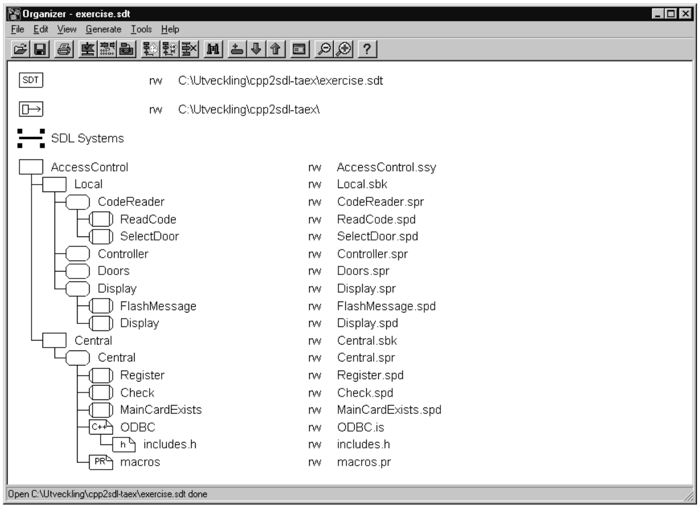
The steps for instantiating a C++ class from the Simulator (e.g. by sending a signal containing a parameter of class type) are similar to the ones shown above.

Summary of the AccessControl Example

The walk-through of the AccessControl example above has shown the typical workflow when using the C/C++ Access.

- The SDL specification is **edited** by adding PR symbols to it, and they are refined to become import specifications. C/C++ header files are added under each import specification, and appropriate translation options are set by means of the CPP2SDL Options dialog. A `TRANSLATE` section may also be added to the import specification listing the names of the declarations to be translated.
- The SDL specification is **analyzed** as case-sensitive SDL. Errors in the C/C++ headers or in the SDL specification are detected by CPP2SDL or the SDL Analyzer respectively.
- C or C++ code is **generated** by using the Make dialog or, preferably, the Targeting Expert. The generated code is compiled and linked together with additional object files.
- The SDL specification may be **simulated** using the Simulator UI.

Figure 42 below shows the Organizer view of how the AccessControl system may look like when it is completed.



```
SDT                    rw   C:\Utveckling\cpp2sdl-taex\exercise.sdt
                       rw   C:\Utveckling\cpp2sdl-taex\
SDL Systems
AccessControl                          rw   AccessControl.ssy
  Local                                rw   Local.sbk
    CodeReader                         rw   CodeReader.spr
      ReadCode                         rw   ReadCode.spd
      SelectDoor                       rw   SelectDoor.spd
    Controller                         rw   Controller.spr
    Doors                              rw   Doors.spr
    Display                            rw   Display.spr
      FlashMessage                     rw   FlashMessage.spd
      Display                          rw   Display.spd
  Central                              rw   Central.sbk
    Central                            rw   Central.spr
      Register                         rw   Register.spd
      Check                            rw   Check.spd
      MainCardExists                   rw   MainCardExists.spd
      ODBC                             rw   ODBC.is
        includes.h                     rw   includes.h
      macros                           rw   macros.pr
Open C:\Utveckling\cpp2sdl-taex\exercise.sdt done
```

Figure 42: Organizer view of the AccessControl system

As can be seen in the figure, the Central process has one PR symbol that has not been refined into an import specification. Instead this symbol is connected to an ordinary SDL/PR file, `macro.pr`, that contains external SDL synonyms that represent C/C++ macros that are needed in the calls to the ODBC API. The sorts of these synonyms are imported by the `ODBC.is` import specification. An alternative technique for accessing C/C++ macros, based on the `#CODE` operator, is described in "Accessing C/C++ Macros from SDL" on page 102.

Import Specification

The import specification is a text file written in a simple C/C++ style syntax. You can specify exactly which declarations in the input header files to access, by using an import specification. The specified subset of the declarations is translated by CPP2SDL. The import specification also enables access to e.g. class and function templates. For more information about import specifications, see "Import Specifications" on page 778 in chapter 14, *The CPP2SDL Tool, in the User's Manual*.

The example below shows a simple import specification where the identifiers `a_int`, `i_arr` and `func` are made available in SDL.

Example 28: A simple import specification

```
TRANSLATE {
        a_int
        i_arr
        func
}
```

If an identifier in an import specification refers to a declaration that depends on other declarations, CPP2SDL will translate all these declarations as well.

There are some more advanced constructs that can be used in an import specification:

- Type Declarators
- Prototypes for Ellipsis Functions

For more information about these constructs, see “Advanced Import Specifications” on page 781 in chapter 14, *The CPP2SDL Tool, in the User´s Manual*.

Templates

By using the CPP2SDL tool, instantiations of template declarations are supported.

To be able to instantiate a C++ template, CPP2SDL needs information about its actual template arguments. This information is given in an import specification.

The C++ template declaration is not itself translated to SDL. Instead an instantiation of the template is mapped to SDL.

Accessing C/C++ Constructs not Fully Supported by CPP2SDL

Accessing C/C++ Macros from SDL

Macros are used for conditional compilation, but can also be used for other purposes:

- To define constants: `#define PI 3.1415`

- To define types: `#define BYTE char`
- To define functions: `#define max(a,b) a>b?a:b`

Macros are not part of the C or C++ languages and are therefore not translated to SDL. Instead, the preprocessor expands all macros before CPP2SDL perform the translation.

To be able to access macro constants from SDL, the implicit `#CODE` operator or SYNONYM can be used, see example below.

Example 29: Constants defined as C++ macros ——————————

C++:

```
#define PI 3.1415;
```

SDL using #CODE:

```
dcl a double;

task a := #CODE('PI');
```

SDL using SYNONYM:

```
SYNONYM PI double = EXTERNAL 'C++';

dcl a double;

task a := PI;
```

———————————————————————————————

To be able to access macro definitions for types or functions, the macro `__CPP2SDL__` can be used. The `__CPP2SDL__` macro is defined when CPP2SDL executes, but not otherwise, and is used in a special header file (called `x.h` in the examples below). This header file must then be included in the set of header files that are translated by CPP2SDL.

The following examples illustrate how the `__CPP2SDL__` macro can be exploited to change C/C++ headers to make macro definitions for types and functions available in SDL.

Example 30: Macro "types" in C++ headers

```
#define BYTE char
```

In the C++ fragment above, the macro `BYTE` is used as if it were a type. The preprocessor will resolve all `BYTE` occurrences, which result in that `BYTE` cannot be available in SDL. To avoid this, the definition of `BYTE` can be changed to the following:

```
x.h:
#ifndef __CPP2SDL__
#ifdef BYTE
#undef BYTE
#endif
#define BYTE char
#else
typedef char BYTE;
#endif
```

The macro `BYTE` is now available as a type in SDL, since `__CPP2SDL__` will be defined during the C++ to SDL translation. In the generated C++ code, `BYTE` is a macro, since `__CPP2SDL__` will be undefined.

Example 31: Macro "functions" in C++ headers

```
#define max(a,b) a>b?a:b
```

By defining `max` as a macro, `max` can be used as if it were a function. The macro `max` can be used for any type for which `>` is defined. The following definition makes `max` available in SDL for `char` and `int`.

```
x.h:
#ifndef __CPP2SDL__
#ifdef max
#undef max
#endif
#define max(a,b) a>b?a:b
#else
int max(int a,int b);
char max(char a, char b);
#endif
```

With the above definition, `max` will be regarded as an operator by the SDL system, since `__CPP2SDL__` has been defined. When the C++ code generated from SDL is compiled, the C++ preprocessor will resolve the "function calls" to `max`, since the macro `__CPP2SDL__` then will be undefined.

Function Pointers

Function pointers are mapped to untyped pointers in SDL, `ptr_void(void*)`. This allows function pointers to be represented in SDL. However, it is not possible to work with this SDL representation. For example to call a function that the pointer points at or to assign the function pointer with the address of an SDL operator, you have to do as shown in the following example:

Example 32: Using function pointers ———————————————

C++:

```
int func1(int i, int j);
int con_sum(int a, int b, int (*F)(int,int));
```

Import Specification:

```
TRANSLATE {
func1
con_sum
}
```

SDL:

```
NEWTYPE global_namespace_ImpSpec /*#NOTYPE*/
  OPERATORS
    con_sum : int, int, ptr_void -> int;
    func1 : int, int -> int;
ENDNEWTYPE global_namespace_ImpSpec; EXTERNAL 'C++';
```

SDL using #CODE alternative 1:

```
dcl
   sum int,
   pfunc ptr_void;

task {
   pfunc := #CODE('(void*) &func1');
   sum := con_sum(1,4,#CODE('(int (*)(int,int))
#(pfunc)'));
};
```

SDL using #CODE alternative 2:

```
dcl
   sum int;

task {
   sum := con_sum(1,4,#CODE('&func1'));
};
```

Unsupported Overloaded Operators

In both C++ and SDL, there is a possibility to override predefined operators. In the table below, the overloaded C++ operators that CPP2SDL supports are listed.

C++ operator	Description	SDL operator
+	(binary) Addition	+
-	(binary) Subtraction	-
*	(binary) Multiplication	*
*	(unary prefix) Dereference	*>
/	(binary) Division	/
%	(binary) Modulo	rem
!	(unary prefix) Not	not
<	(binary) Less	<
>	(binary) Greater	>
<<	(binary) Left Shift	<
>>	(binary) Right Shift	>
==	(binary) Equal	=
!=	(binary) Not Equal	/=
<=	(binary) Less Equal	<=
>=	(binary) Greater Equal	>=
&&	(binary) And	and
||	(binary) Or	or

Both shift operators and the less/greater operators in C++ are mapped to < and > in SDL. This mapping implies that overloading is supported on either < and > or << and >> in SDL. If both these operator pairs are overloaded, CPP2SDL will issue a warning, and select the former pair to be represented in SDL.

Overloaded operators, that are not supported by CPP2SDL, can be handled using the operator `#CODE`.

C Specific Package ctypes

IBM Rational offers a special package `ctypes` that contains data types and generators that match C. It is described in detail in *chapter 62, The ADT Library, in the User's Manual*. The `ctypes` package should be used in the following cases:

- if you want to use pointers in SDL
- if you need a data type that matches some specific C type (for example short int) for which there is no corresponding SDL sort.
- if you use C headers directly in SDL In this case package `ctypes` **must** be used.

The tables below list the data types and generators in `ctypes` and their C counterparts.

SDL Sort	Corresponding C Type
`ShortInt`	`short int`
`LongInt`	`long int`
`UnsignedShortInt`	`unsigned short int`
`UnsignedInt`	`unsigned int`
`UnsignedLongInt`	`unsigned long int`
`Float`	`float`
`Charstar`	`char *`
`Voidstar`	`void *`
`Voidstarstar`	`void **`

SDL Generator	Corresponding C Declarator
`Carray`	`C array, i.e. []`
`Ref`	`C pointer, i.e. *`

The rest of this section explains how these data types and generators can be used in SDL.

Different Int Types and Float

`ShortInt`, `LongInt`, `UnsignedShortInt`, `UnsignedInt`, `UnsignedLongInt` are all defined as syntypes of Integer, so from an SDL point of view, these data types are really the same, and the normal Integer operators can be used on these types. The only difference is that the code that is generated for these types is different. `Float` is defined as a syntype of Real.

Charstar, Voidstar, Voidstarstar

`Charstar` represents character strings (i.e. `char *`) in C. Charstar is not the same as the SDL predefined type Charstring! Charstar is useful when accessing C functions and data types that use `char *`. In other cases it is better to use Charstring instead (see also "Charstring" on page 50). Conversion operators between Charstar and Charstring are available (see below).

`Voidstar` corresponds to `void *` in C. This type should only be used when accessing C functions that have `void *` parameters, or that return `void *` (in which case it is advised to "cast" the result directly to another type).

`Voidstarstar` corresponds to `void **` in C. This type is used in combination with the `Free` procedure described in "Using Pointers in SDL" on page 112. In rare cases this type is also needed to access C functions.

The following conversion operators in `ctypes` are useful:

```
cstar2cstring : Charstar   -> CharString;
cstring2cstar : CharString -> Charstar;
cstar2vstar   : Charstar   -> Voidstar;
vstar2cstar   : Voidstar   -> Charstar;
cstar2vstarstar : Charstar -> Voidstarstar;
```

These operators have the following behavior:

- `cstar2cstring`:
  Converts a C string to an SDL Charstring. For example if variable `v` of type Charstar contains the C string `"hello world"`, then `cstar2cstring(v) = 'hello world'`.
- `cstring2cstar`:
  Converts an SDL Charstring to a C string, i.e. the opposite of `cstar2cstring`.

- `cstar2vstar`:
  Converts a Charstar to a Voidstar. This operator is sometimes useful when calling C functions with `void *` parameters.
- `vstar2cstar`:
  Converts a Voidstar to a Charstar. This operator can for example be used if a C function returns `void *`, but the result should be "casted" to a `char *`.

The Carray Generator

The generator `Carray` in package `ctypes` is useful to define arrays that have the same properties as C arrays. Carray takes two generator parameters; an integer value and a component sort.

Example 33: Carray instantiation

```
newtype IntArr Carray(10, Integer)
endnewtype;
```

The defined type `IntArr` is an array of 10 integers with indices 0 to 9, corresponding to the C type

```
typedef int IntArr[10];
```

Two operators are available on instantiations of Carray; *modify!* to change one element of the array, and *extract!* to get the value of one element in the array. These operators are used in the same way as in normal SDL arrays, see "Array" on page 71. There is no (. ... .) notation provided for denoting values of whole CArrays.

```
    modify!  : Carray, Integer, Itemsort -> Carray;
    extract! : Carray, Integer           -> Itemsort;
```

Example 34: Use of Carray in SDL

```
DCL v IntArr, i Integer;

TASK v(0) := 3; /* modifies one element */
TASK i := v(9); /* extracts one element */
```

If a C array is used as parameter of an operator, it will be passed by address, just as in C. This makes it possible to write operators that change the contents of the actual parameters. In standard SDL this would not be possible.

The Ref Generator

The generator `Ref` in package `ctypes` is used to define pointer types. The following example illustrates how to use this generator.

Example 35: Defining a pointer type

```
newtype ptr Ref(Integer)
endnewtype;
```

The sort `ptr` is a pointer to Integer.

Standard SDL has no pointer types. Pointers have properties that cannot be defined in normal SDL. Therefore they should be used very carefully. Before explaining how to use the Ref generator, it is worthwhile to list some of the dangers of using pointers in SDL.

Pointers Will Lead to Data Inconsistency

If more than one process can read/write to the same memory location by means of pointers, data inconsistency can and will occur! Some examples:

- In a flight reservation system there is one seat left, and two reservation requests come in simultaneously. If pointers were used to check the availability of seats, both requests might be approved! In literature this is called the “writers-writers problem”.
- A process may update some array variable. If at the same time another process tries to read the variable by means of a pointer to the array, the reading process may get a value were some elements of the array are “new” while other elements are “old”, and the total result makes no sense. This is the classic “readers/writers problem”.

Even though tools such as the SDL Simulator and SDL Explorer will be able to detect a number of errors regarding pointers, there are situations that cannot be detected with these tools! This is because the Explorer and Simulator assume a scheduling atomicity of at best one SDL symbol at a time. This may not hold in target operating systems where one process can be interrupted at any time (pre-emptive scheduling). If pointers are used, data is totally unprotected, and data inconsistency may occur, even though the Explorer did not discover any problems! All these problems can be avoided by using SDL constructs for accessing data, like remote procedures and signal exchange.

Caution!

For the above stated reasons, **never pass pointers to another process!** Not in an output, not in a remote procedure call, not in a create, and not by exported/revealed variables!

And if you do not obey this rule anyway: after passing a pointer, release immediately the "old" pointer to prevent having several pointers to the same data area. For example (for some pointer `p`):

```
OUTPUT Sig(p) TO ...;
TASK p := Null;
```

Pointers Are Unpredictable

If you have an SDL system that always works except during demonstrations, then you have used pointers! Bugs with pointers may be very hard to discover, as a system may (accidentally) behave correctly for a long time, but then suddenly strange things may happen. Finding such bugs may take very long time; in rare cases you might not find them at all!

Caution!

Bugs caused by pointers may be hard to find!

Pointers Do Not Work in Real Distributed Systems

If an SDL system is "really" distributed, i.e. where processes have their own memory space, it makes no sense to send a pointer to another process, as the receiving process will not be able to do anything with it. Therefore, by communicating pointers to other processes, limitations are posed on the architecture of the target implementation.

Pointers Are Not Portable

The Ref generator and its operators are completely IBM Rational-specific. It is highly unlikely that SDL systems using pointers will run on other SDL tools.

Using Pointers in SDL

If you still want to use pointers in SDL after all these warnings, this section explains how to do this. A pointer type created by the Ref generator always has a literal value `Null` (corresponds to `NULL` in C), which is also the default value. The literal `Alloc` is used for the dynamic creation of new memory. Examples are given later.

Note:

It is up to the user to keep track of all dynamically allocated data areas and to free them when they are no longer needed.

The following operators are available for Ref types:

```
"*>"  : Ref, Itemsort -> Ref;
"*>"  : Ref -> Itemsort;
"&"   : Itemsort -> Ref;
make! : Itemsort -> Ref;
free  : in/out Ref;
"+"   : Ref, Integer -> Ref;
"-"   : Ref, Integer -> Ref;
vstar2ref : Voidstar -> Ref;
ref2vstar : Ref -> Voidstar;
ref2vstarstar : Ref -> Voidstarstar;
```

Furthermore, the following procedure is defined:

```
procedure Free; fpar p Voidstarstar; external;
```

These operators can be used in the following way:

- `*>` (postfix operator):
  Gets/changes the contents of a pointer. This is a postfix operator, so `p*>` returns the contents of pointer p. In SDL terminology this is the extract and modify operators for pointers.
- `&` (prefix operator):
  Address-operator. This is a prefix operator, so `&var` returns a pointer to variable `var`.
- `make!` or `(. .)`
  This constructor allocates new memory and assigns the parameter to make to the newly allocated memory.
- `free`
  This operator takes a pointer variable, frees the memory it refers to and sets the pointer variable to Null.
- `+`, `-`:
  used to add/subtract an offset to/from an address. This can be useful to access arrays in C. These operators are defined as in C, e.g. if `p`

is a pointer to some struct, then `p+1` points to the next struct (not to byte `p+1`).

- `vstar2ref`:
  Converts Voidstar to another pointer type. Should only be used to "cast" the result of C functions that return a `void *`.
- `ref2vstar`:
  Converts a pointer to Voidstar. This is useful when calling C functions that have `void *` parameters.
- `ref2vstarstar`:
  Returns the address of the pointer as a `void **`. This operator is needed when calling the `Free` procedure.
- Procedure `Free`: (NOTE: use `free` operator above instead)
  This procedure is used to release memory that has previously been allocated with `alloc`. This procedure is only provided for backward compatibility, use the `free` operator described above instead.

Example 36: Use of the Ref operators

```
NEWTYPE ptr Ref(Integer)
ENDNEWTYPE;

DCL p ptr,
    i, j Integer;

TASK p := alloc; /* creates dynamically a new
                    integer; p points at it */
/* here it should be checked that p != Null */
TASK p*> := 10;  /* changes contents of p */
CALL free(p);  /* releases the integer */
TASK p := (. 10 .); /* allocate and set to 10 */
CALL free(p);  /* releases the integer again */
TASK p := &i;  /* p now points to i */
TASK p*> := 5; /* changes contents of p, i.e. also
                  i is changed! */
TASK j := p*>; /* gets contents of p (=5) */
```

Using Linked Structures with Pointers

Pointers are useful when defining linked structures like lists or trees. In this section we give an example of a linked list containing integer values. Figure 43 shows an SDL fragment with data type definitions for a linked list, and part of a transition that actually builds a linked list. A list is represented by a pointer to an `Item`. Every `Item` contains a pointer `next` to the next item in the list. In the last item of the list, `next` = `Null`.

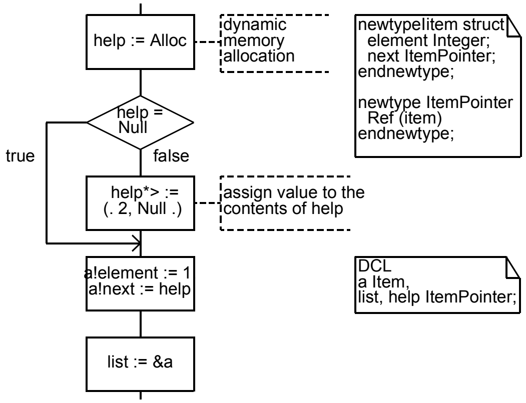


Figure 43: Building a linked list

Figure 44 shows an SDL fragment where the sum of all elements in a list is computed. Note that this computation would never stop if there would be an element that points back in the list, just to illustrate how easy it is to make errors with pointers.

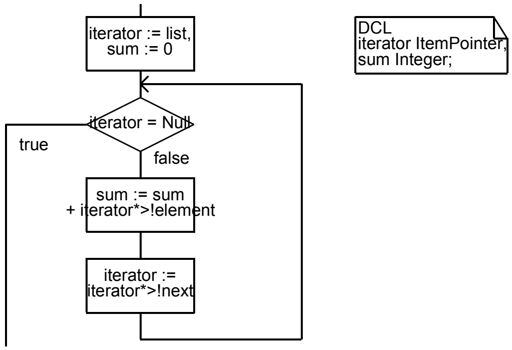


Figure 44: Going through the list

Using ASN.1 in SDL

ASN.1 is a language for defining data types that is frequently used in the specification and implementation of telecommunication systems. ASN.1 is defined in ITU-T Recommendations X.680-X.683. Recommendation Z.105 defines how ASN.1 can be used together with SDL. A subset of Z.105 is implemented in the SDL Suite.

This chapter explains how ASN.1 data types can be used in SDL systems. The following items will be discussed:

- How to organize ASN.1 modules in the SDL Suite
- How to use ASN.1 data types in SDL
- How to share ASN.1 data between SDL and TTCN

Organizing ASN.1 Modules in the SDL Suite

It is recommended to have a special chapter for ASN.1 modules (for example called *ASN.1 Modules*). If many ASN.1 modules are used, they may be grouped into Organizer modules (which is not the same as ASN.1 modules!), see "Module" on page 43 in chapter 2, *The Organizer, in the User's Manual*.

Figure 45 shows an example of the Organizer look of a chapter with two Organizer modules containing ASN.1 modules.

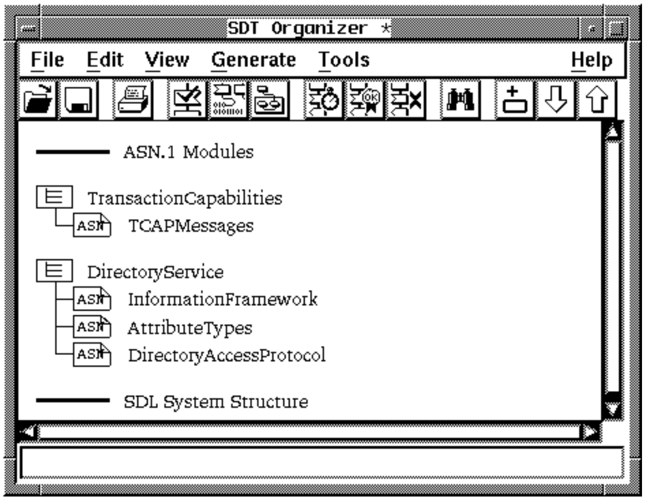


Figure 45: Example of ASN.1 modules in the Organizer

We will show with an example how to use an ASN.1 module in an SDL system. Suppose we have an ASN.1 module `MyModule` in file `mymodule.asn`:

```
MyModule DEFINITIONS ::=
BEGIN

Color ::= ENUMERATED { red(0), yellow(1), blue(2) }

END
```

This module contains one type definition, `Color`, that has three values, `red`, `yellow`, and `blue`.

We first add a new diagram of type *Text ASN.1* to the Organizer using *Edit/Add New* (without showing it in the Editor) and we connect it to the file `mymodule.asn` (using *Edit/Connect*). In order to use the ASN.1 module in SDL, we edit the system diagram and add `use MyModule;` in the package reference clause, as is illustrated in Figure 46 below.

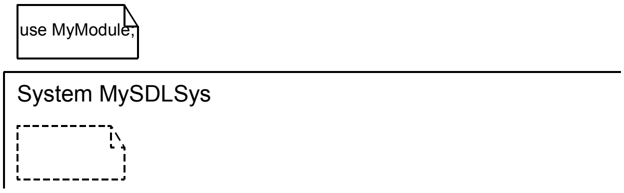


Figure 46: Using an ASN.1 Module in SDL

Figure 47 shows the resulting Organizer view. The symbol below the MySDLSys system symbol is a *dependency link* that indicates that the SDL system depends on an external ASN.1 module. Dependency links for ASN.1 modules that are used by an SDL system were previously required by the Analyzer, but now only serve as comments and are optional.

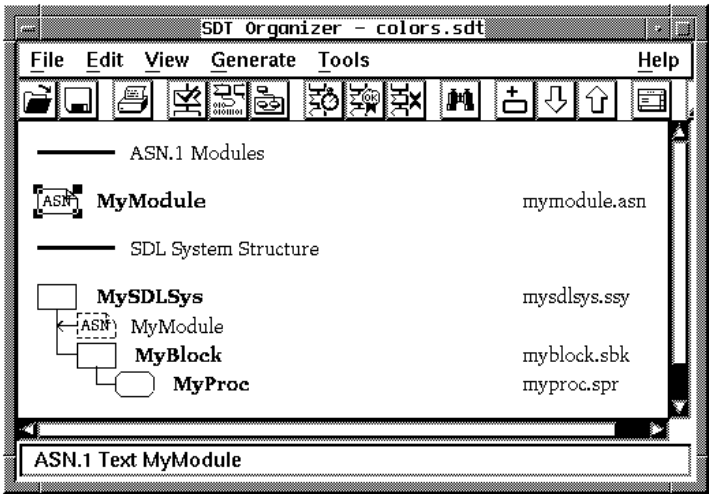


Figure 47: Organizer View of SDL System Using ASN.1 Module

If ASN.1 modules use other ASN.1 modules, dependency links between the ASN.1 modules should be created.

Using ASN.1 Types in SDL

After the above preparations, the data types in `MyModule` can be used in SDL. As an example, we will make an SDL system that converts a character string to the corresponding color. This is done by two signals:

- Signal `GetColor` has ASN.1 type IA5String as a parameter.
- When this signal is sent to the SDL system, the SDL system will reply with signal `ReturnColor`, that has a BOOLEAN parameter indicating whether there is a color that matches the string, and a `Color` parameter.

The system diagram including these signal definitions is shown in Figure 48 below.

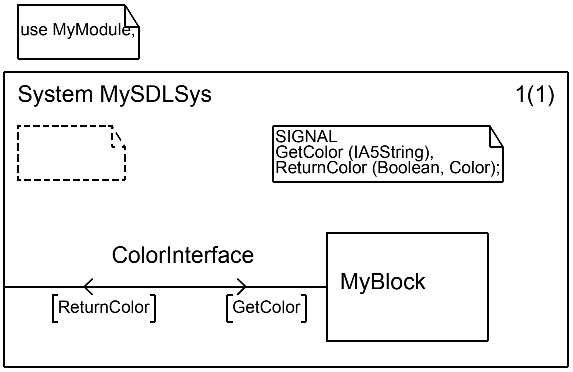


Figure 48: SDL system diagram

The MSC below illustrates how the system is intended to be used.

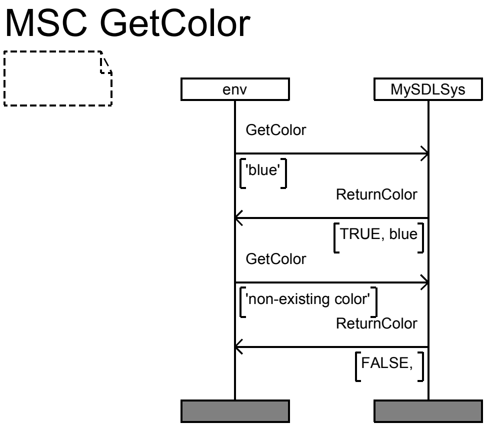


Figure 49: MSC illustrating GetColor

In order to know which values and which operators can be used on ASN.1 types, it is necessary to look in "Translation of ASN.1 to SDL" on page 705 in chapter 13, *The ASN.1 Utilities, in the User's Manual*.

For example, `Color` is defined as an `ENUMERATED` type. By looking at the mapping rules in "Enumerated Types" on page 717 in chapter 13, *The ASN.1 Utilities, in the User's Manual*, we see the list of operators that can be used on `Color`. These are in this case `num`, `<`, `<=`, `>`, `>=`, `pred`, `succ`, `first`, `last`, and also `=` and `/=`, which are always available.

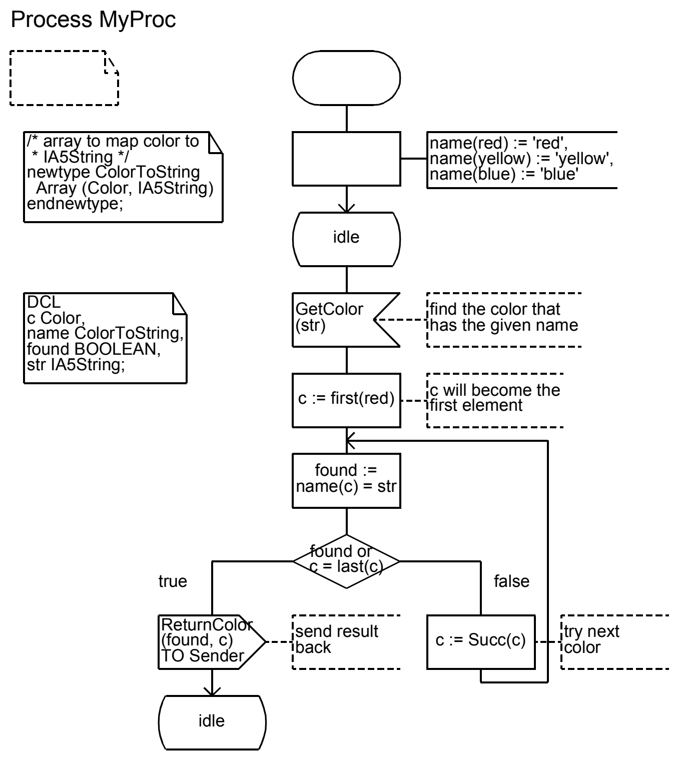


Figure 50: Using the Color type in SDL

Figure 50 shows a fragment of an SDL process that uses `Color`. It contains a loop over all values of `Color`, and illustrates how to declare variables of `Color`, how to use `Color` in new SDL sort definitions, and how to use the operators `first`, `last`, and `succ`. Some notes on the fragment:

- The type `ColorToString` is used to convert a color to an IA5String. In the fragment we do actually the opposite. An alternative solution would be to have a `StringToColor Array (IA5String, Color)` because then no loop would have been needed (see also "Array" on page 71). However, the purpose of the example was to illustrate how to loop through all elements.

- Operator `first` in `c := first(red)` returns the element with the lowest associated number. This ensures that we really get all elements when using the `Succ` operator. In this case we could just as well have written `c := red`.
- Note also the use of the predefined ASN.1 type `IA5String`, which is in fact a syntype of the predefined SDL sort `Charstring`.

Using Predefined ASN.1 Types in SDL

The predefined simple ASN.1 types can be used directly in SDL. In most cases, the ASN.1 type has the same name in SDL, for example ASN.1's type `NumericString` is also called `NumericString` in SDL. However, some predefined ASN.1 types contain white-space, like `BIT STRING`. In SDL, the white-space is replaced with an underscore, so the corresponding SDL sort is called `BIT_STRING`.

The operators on these predefined ASN.1 types are described in detail in section "Using SDL Data Types" on page 42.

Using ASN.1 Encoding Rules with the SDL Suite

The ASN.1 constructs defined in ITU-T Recommendation X.690 and X.691 are supported, which is explained in "ASN.1 Encoding and Decoding in the SDL Suite" on page 2831 in chapter 58, *ASN.1 Encoding and De-coding in the SDL Suite, in the User's Manual*.

Sharing Data between SDL and TTCN

One more advantage of ASN.1 is that TTCN is also based on this language. By specifying the parameters of signals to and from the environment of the SDL system with ASN.1 data types, this information can be re-used in the TTCN Suite for the specification of test cases for the system.

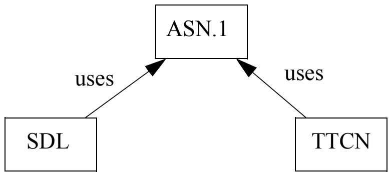


Figure 51: Sharing ASN.1 definitions between SDL and TTCN

This has the big advantage of making it easier to keep the SDL specification consistent with the TTCN test specification.

The use of external ASN.1 in TTCN is covered in more detail in the TTCN Suite manual. In this section we will briefly illustrate how to share data between SDL and TTCN using TTCN Link.

Supposing we have to write a test suite for the SDL system with the `Colors` example, we would add a new diagram – a TTCN Test Suite, for example called `ColorTest` – to the Organizer. In this test suite we want to use definitions from the ASN.1 module `MyModule` that contains the `Colors` data type. For this purpose we need to set a dependency link between the ASN.1 module and the test suite. We do this by selecting the ASN.1 module in the Organizer. By using *Generate/Dependencies* we connect it to the TTCN test suite `ColorTest`.

We can also use TTCN Link to generate declarations from SDL system `MySDLSys`. For this purpose, it is easiest to associate the SDL system with the TTCN test suite. This is done by selecting the SDL system diagram in the Organizer and associate it with the TTCN test suite using *Edit/Associate*. The Organizer View for the test suite now looks as in Figure 52 below:

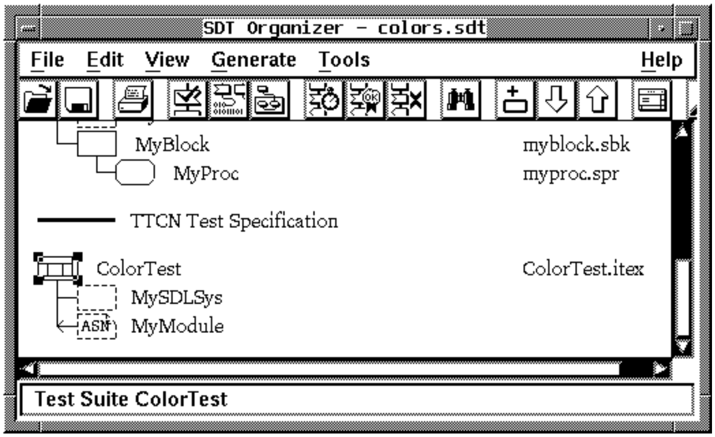
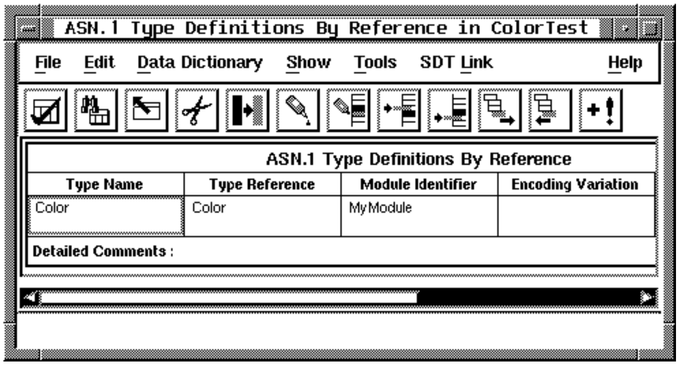


Figure 52: Organizer view of a test suite that uses ASN.1

We can generate a TTCN Link executable for the SDL system by selecting the SDL system in the Organizer and using *Generate/Make* select standard kernel *TTCN Link*. Now we can start the TTCN Suite by double-clicking on test suite `ColorTest`. By using *TTCN Link/Generate Declarations*, we can automatically generate the PCOs, ASP type definitions and ASN.1 type definitions. If we look at the result, we can see that `Color` is present as an ASN.1 Type Definition by Reference. This table is shown below.


ASN.1 Type Definitions By Reference in ColorTest
File Edit Data Dictionary Show Tools SDT Link Help

ASN.1 Type Definitions By Reference

Type Name	Type Reference	Module Identifier	Encoding Variation
Color	Color	MyModule	

Detailed Comments :

Figure 53: Resulting table in the TTCN Suite

Now this data type can be used in creating test cases, in constraints, etc. If at some point in time the definition of `Color` would be changed (for example if we would add a new color), then, in order to update the test suite accordingly, we can select the TTCN table for `Color`. In the Analyzer dialog, we should select both *Enable Forced Analysis* and *Retrieve ASN.1 Definitions*. Now the TTCN test suite will be updated with the new definition for `Color`.

Chapter

3

Using SDL Extensions

This chapter describes some of the extensions to SDL that are available in the SDL Suite, and that are not documented outside IBM Rational.

The extensions covered here are the Own and ORef generators, and the algorithmic extensions to SDL. There are other IBM Rational-specific extensions to SDL supported in the SDL Suite, mainly concerning data types, generators and operators. These extensions are covered in chapter 2, *Data Types*.

Own and ORef Generators

Introduction

A major problem to obtain fast applications generated from SDL is the data model, that requires copying of data in a number of places. In an interchange of a signal between two processes, the signal parameters are first copied from the sending process to the signal and then again copied from the signal to the receiver. If the two processes have access to common memory, it would be possible only to pass a reference to the data via the signal, and in that way there would be no need to perform the two copy actions.

The generator Ref can be used for this purpose (see "The Ref Generator" on page 111 in chapter 2, *Data Types*), but there is a number of important problems when using the Ref generator:

- The user has to deallocate memory when it is no longer used. If the user forgets this in any circumstance, memory will be lost.
- It is easy to, intentionally or unintentionally, access the same memory from several process instances. This is very bad practice in real-time programming (without protection for simultaneous access to the memory) and might cause unwanted behavior. These kinds of errors are usually very difficult to find.

Basic Properties of the Own Generator

The purpose of the Own generator is to solve the situation described above, i.e. it should be possible to limit the number of copy operations that are needed, at the same time as the user should not need to worry about memory deallocation, and simultaneous access to the memory from several processes should not be possible.

The basic property of the Own generator that makes this possible is that only one Own pointer at a time can refer to the same memory. This variable (of Own type) is referred to as the owner of the memory. Ownership is passed to another variable by assignment.

Example 37: Own variables

```
newtype Data struct
  a, b integer;
endnewtype;

newtype Own_Data Own(Data)
endnewtype;

dcl
  v1 Own_Data,
  v2 Own_Data;

task v1 := v2;
```

This assignment is interpreted as follows:

- If v1 refers to some memory this memory is deallocated.
- v1 is assigned the value of v2, i.e. refers to the same memory as v2.
- v2 is set to Null.

By this scheme the basic properties of the Own generator is preserved, i.e. all memory no longer accessible is deallocated and there is only one reference to the data.

To handle more complex cases, the order in which these operations are performed is a bit more complicated. With the same types and variables as in the example above and the procedure P, taking three Own_Data parameters:

```
task v1 := P(v2, v1, v2);
```

we get the following execution:

Evaluation of the right-hand side is performed from left to right, i.e. starts with the first actual parameter of P. The first formal parameter of P is assigned the value of v2 and takes the ownership of this memory. The variable v2 is assigned the value Null. The same thing happens for the second formal parameter and the variable v1. The third formal parameter of P get the value Null as v2 is Null at this point.

Now the procedure P is called and its return value is obtained. Before assigning this value to the variable on the left hand side, i.e. to v1, the memory currently referred to by v1 is deallocated. In this case v1 is Null at this point as the second formal parameter of P already have taken the ownership of this memory. Last the variable v1 is assigned the value returned by P.

Ownership is, as can be seen in the example above, passed not only in assignments, but in every case where the reference is assigned to some other place. This happens for example in assignments, input, output, set, reset, procedure call, and create. The only places where ownership is **not** passed is:

- when the explicit copy function is used, for example
  ```
  task v1 := copy(v2);
  ```
- in import, export, and view operations, which are interpreted as containing implicit copy operations.
- in calls to the standard functions '=' and "/=".

Note that `copy(v2)` is a "deep" copy, i.e. any Own pointers in the copied data structure are also copied. Otherwise we would end up with several references to the same memory.

Definition of Own Generator

The Own generator is defined in SDL according to the following:

```
GENERATOR Own (TYPE Itemsort)
  LITERALS
     Null;
  OPERATORS
    "*>"  : Own, ItemSort -> Own;
    "*>"  : Own -> ItemSort;
    make! : ItemSort -> Own;
  DEFAULT Null;
ENDGENERATOR Own;
```

Basically the Own generator is a way to introduce pointers to allocated memory. The Null value is as usual interpreted as "a reference to nothing". The operators "*>" are the Extract! and Modify! operators, i.e. the way to reference or modify the memory referred to by the pointer. Using the type and variables in the previous example the following statements are correct:

```
task v1*>!a := 1;
task i := v2*>!b;
   /* integer assignment, i is integer variable */
task d := v2*>;
   /* struct level assignment, d is of type Data */
```

The "*>" operator have the same properties as the '*' in C, i.e. "v1*>" has the same meaning as "*v1" in C. To make the syntax a bit easier there is a possibility to let the SDL Analyzer implicitly insert the "*>" in the expressions where it is needed. The example above would then become:

```
task v1!a := 1;
task i := v2!b;
   /* integer assignment, i is integer variable */
task d := v2;
   /* struct level assignment, d is of type Data */
```

which is a bit easier to read. More details about the implicit type conversions can be found in "Implicit Type Conversions" on page 134.

Before it is possible to start working with components in the data referenced by the Own pointer, the Own pointer must be initialized with a complete value (default is Null as can be seen in the definition). The Make! operator is a suitable way to initialize a variable. As usual the concrete syntax for Make! is "(. x .)", where x should be replaced by a value of the ItemSort for the current Own pointer.

Example of intializations using the data definitions in the examples above:

```
dcl v1 Own_Data := (. (. 1, 2 .) .);
task v1 := (. (. 5, 5 .) .);
```

The inner "(. .)" is for the constructing the struct value and the outer "(. .)" is for the Own make! function. It is, however, possible to avoid the double parentheses as there is an implicit type conversion from a type T to Own(T), by implicitly inserting "(. .)" around a value of type T. So the examples above could (and probably should) be written as

```
dcl v1 Own_Data := (. 1, 2 .);
task v1 := (. 5, 5 .);
```

Again, please see "Implicit Type Conversions" on page 134. The other operations available for own pointers, apart from "*>" and make!, are assignment, test for equality, and copy. The assign operator has already been described above. Test for equality ('=' and "/=") does NOT test for pointer equality as two Own pointers cannot be equal. Instead equality is "deep" equality, i.e. the values referred to by the Own pointers are compared.

An implicit copy operator has been inserted for every type. It takes a value and returns a copy of that value. For all types that are not Own pointers or contain Own pointers, this operator is meaningless as it just returns the same value. For Own pointers or for structured values containing Own pointers, the copy function, however, copies the values referenced by the Own pointers.

The ORef Generator

The ORef generator is intended to be used together with the Own generator to provide a way to temporary refer to owned data during some algorithm, without affecting the ownership of the memory. If, for example, Own pointers is used to create a linked list and we would like to write a procedure that calculates the length of the list, then we need a temporary pointer going through the list. If that pointer was a Own pointer the list would be destroyed while we traverse the list, as there may be only one Own pointer referring to the same memory.

Example 38: Own and ORef

```
newtype ListElem struct
  Data MyType;
  Next ListOwn;
endnewtype;

newtype ListOwn Own(ListElem)
endnewtype;
newtype ListRef ORef(ListElem)
endnewtype;

procedure ListLength; fpar Head ListRef;
returns integer;
dcl
  Temp ListRef,
  Len  Integer;
start;
  task Len := 0, Temp := Head;
  again :
  decision Temp /= null;
  (true) :
    task Len := Len+1, Temp := Temp!Next;
    join again;
  (false) :
  enddecision;
  return Len;
endprocedure;

dcl
  MyList ListOwn,
  L integer;

task L := call ListLength(MyList);
```

Note the use of the ListRef type both in the formal parameter Head and in the local variable Temp. If Head would be of ListOwn type, the variable MyList would be null after the call of ListLength, which is not what we intended. If ListOwn was used as type for the variable Temp, as the statement `Temp := Temp!Next` would unlink the complete list.

Another example of a typical application of ORef, is to introduce backward pointer in a linked list, to make it doubly linked. If the forward pointers are Own pointers then the backward pointers cannot be Own pointers as then we would have two Own pointers on the same object.

The ORef generator is defined as:

```
GENERATOR ORef (TYPE Itemsort)
  LITERALS
     Null;
  OPERATORS
    "*>"  : ORef, ItemSort -> ORef;
    "*>"  : ORef -> ItemSort;
    "&"   : ItemSort -> ORef;
    "="   : ORef, ItemSort -> Boolean;
    "="   : ItemSort, ORef -> Boolean;
    "/="  : ORef, ItemSort -> Boolean;
    "/="  : ItemSort, ORef -> Boolean;
  DEFAULT Null;
ENDGENERATOR;
```

Where “*>” is used for dereferencing and ‘&’ is an address operator.

Run-Time Errors

There are four situations, concerning Own and ORef, that can lead to a run-time error. These situations are:

- Dereferencing of a null pointer.
- An ORef pointer that refers to an object that has been deallocated.
- An ORef pointer that refers to an object that is owned by another process.
- A cycle of Own pointers is created, as this memory can never be deallocated.

These problems are all found during simulation and validation, except that if an ORef pointer refers to a data area that is first deallocated and then allocated again, the ORef pointer is no longer invalid.

Examples of run-time error situations assuming the data types in the previous section.

```
dcl
  L1, L2 ListOwn,
  R1, R2 ListRef,
  I      Integer;

task
  L1 := null,
  R1 := null,
  L1!Data := 1,
    /* ERROR: Dereferencing of Null pointer */
  I := R1!Data;
    /* ERROR: Dereferencing of Null pointer */

task
  L1 := (. 1, null .),
  R1 := L1,
  L1 := null,
  I := R1!Data;
    /* ERROR: Reference to deallocated memory */

task
  L1 := (. 1, null .),
  R1 := L1;
output S(L1) to sender;
task
  I := R1!Data;
    /* ERROR: Reference to memory not owned by
       this process */

task
  L1 := (. 1, null .),
  L1!Next := (. 2, null .),
  L1!Next!Next := L1;
    /* ERROR: Loop of own pointers created */
```

Implicit Type Conversions

Note:

Implicit type conversions are by default off in the SDL Analyzer. It can be turned on in the Analyze dialog in the Organizer. Implicit type conversions will, however, influence the time needed for semantic analysis. The more complex an expression is, the more effect on time the implicit type conversions will have, as the number of possibilities increases (often exponentially) with the length of the expression.

The purpose of the implicit type conversions is to simplify the use of the Own generator. The code that operate on data structures should be the same if you use a data structure T or if you use a own pointer to T, own(T). The only thing that a user has to think about is if ownership should be passed or if a copy should be made, when passing data to somewhere else.

The implicit conversions never change the type of, for example, an assignment. If there is an assignment:

```
task R1 := L1;
```

no implicit type conversions are applied on R1, as that would change the type of the assignment. Type conversions might be applied on L1 in the right hand side, to obtain the correct type. In an assignment:

```
task Q(a) := ...;
```

the implicit type conversions might also be applied to the index expression, i.e. to a.

In a test for equality and in similar situations, e.g. in:

```
L1 = R1
```

implicit type conversions are first applied to the left expression, i.e. to L1. If that yields a correct interpretation, that one is selected. Otherwise implicit type conversions are tried on the right expression, i.e. to R1.

Assume a type T and a two generator instantiations `Own_T = Own(T)` and `ORef_T = ORef(T)`. Assume also the variables:

```
dcl
  t1 T,
  v1 Own_T,
  r1 ORef_T;
```

Then the following implicit type conversions are possible:

```
1. Own_T  -> T,       by v1 -> v1*>
2. T      -> Own_T,   by t1 -> (. t1 .)
3. Own_T  -> ORef_T,  by v1 -> demote(v1)
4. ORef_T -> Own_T,   by r1 -> (. r1*> .)
5. T      -> ORef_T,  by t1 -> &t1
6. ORef_T -> T,       by r1 -> r1*>
```

Type conversions 1 and 6 make it possible to exclude "*>" in component selections. Instead of writing

```
a*>!b*>(10)*>!c
```

it is possible to write

```
a!b(10)!c
```

This possibility also exists for ordinary Ref pointers.

Type conversion 2 makes it possible to assign an Own pointer a new value of the Own pointer component type. If A is a Own pointer to a struct containing two integers, then it is possible to write:

```
A := (. 1, 2 .);
```

which means the same as

```
A := (. (. 1, 2 .) .);
```

where the inner "(. .)" is the make! function for the struct and the outer "(. .)" is the make! function for the Own pointer.

This possibility also exists for ordinary Ref pointers.

Type conversion 3 makes it possible to assign an ORef pointer to an Own pointer. This is already used in the examples above, but is not directly possible, as an ORef and an Own pointer are two distinct types. The `demote` operator converts a Own pointer to the corresponding ORef pointer. (Corresponding means the first ORef with the same component type in the same scope unit as the Own pointer type is defined).

Type conversion 4 makes it possible to construct a new Own value, starting from a ORef value. The conversion is performed in two steps, first going from ORef_T to T by applying conversion 6, and then from T to Own_T by applying conversion 2.

Type conversion 5 makes it possible to let a ORef_T pointer refer to a DCL variable by writing:

```
task r1 := t1;
```

which means the same thing as

```
task r1 := &t1;
```

where '&' is the address operator (as in C).

Algorithms in SDL

A former problem in SDL is the lack of support for writing algorithms. For pure calculations, not involving communication, the graphical form for SDL tends to become ordinary flow charts, which is usually not a good way to describe advanced algorithms. Such, often large, parts of an SDL diagram also hide other, from the SDL point-of-view, more important parts of the diagram, namely the state machine and the communication aspects.

The algorithmic extensions described here addresses these problems by introducing the possibility to write algorithms in textual form within a Task symbol, and also to define procedures and operators in textual form in text symbols. There are two major advantages with this approach, compared to ordinary SDL:

- The algorithms are written in a compact form, similar to ordinary programming languages, and will therefore not hide other important aspects of an SDL diagram.
- The language used within the algorithms contains more powerful algorithmic constructs than ordinary SDL, like if-then-else, and loop statements.

In addition, the algorithmic extensions make it possible to now define procedures and operators in textual form in text symbols in SDL/GR.

These algorithmic extensions to SDL have been approved by ITU Study Group 10 to be incorporated into the official Master List of Changes that will affect the next ITU recommendation for SDL. There are a few minor differences in the support for SDL algorithms in the SDL Suite compared with the ITU definition – these are noted in the descriptions below.

The constructs that are part of the extensions are:

- Compound Statement
- Local Variables
- If Statements
- Decision Statements
- Loop Statements
- Label Statements
- Jump Statements
- Empty Statements

Compound Statement

The basic concept in the algorithmic extensions is the *compound statement*. A compound statement starts with a '{', which is followed by a sequence of variable declarations and a sequence of statements, and it then ends with a '}'.

A compound statement may be used in three places:

- as the contents of a TASK,
- as the body of a procedure or operator definition in a text symbol,
- as a statement within an enclosing compound statement.

Note:

According to the ITU language definition the body of a procedure or operator is allowed to be a statement. In the SDL Suite, however, a compound statement is required. This means that if the body consists of only one statement, the enclosing "{ }" are required anyhow.

Note also that the enclosing "{ }" should not be included in a Task symbol in the SDL Editor. These braces will be added when the SDL system is converted to SDL/PR for analysis.

Example 39

Contents in Task symbol in SDL/GR:

```
a := b+1;
if (a>7) b := b+1;
```

Corresponding code in SDL/PR:

```
task {
  a := b+1;
  if (a>7) b := b+1;
};
```

Example 40

A procedure in a text symbol in SDL/GR, or in SDL/PR:

```
procedure p fpar i integer returns integer
{
  if (i>0)
    i := i+1;
  else
    i := i-1;
  return i;
}
```

Local Variables

Within a compound statement it is allowed to define a number of local variables. These variables will be created when the compound statement is entered and will be destroyed when the compound statement is left. The semantics of a compound statement is very much like a procedure without parameters, which is defined and called at the place of the compound statement.

A variable declaration within a compound statement looks that same as ordinary variable declarations, except that “exported” and “revealed” are not allowed. Example:

```
dcl
  a, b integer := 0,
  c    boolean;
```

Statements

A statement within a compound statement my be of any of the following types:

- compound statement
- output statement
- create statement
- set statement
- reset statement
- export statement
- return statement (only in procedures and operators)
- procedure call statement
- assignment statement
- if statement
- decision statement
- loop statement
- label statement
- jump statement
- empty statement

Note that each statement (and each variable declaration statement) ends with a ‘;’. The following statement types use the same syntax as in ordinary SDL/PR:

output, create, set, reset, export, return, call, assignment

Example 41: Ordinary SDL/PR statements

```
output s1(7) to sender;
output s2(true, v1) via sr1;
create p2(11);
set(now+5, t);
reset(t);
export(v1);
return a+3;
call prd1(a, 10);
a := a+1;
```

Note:

According to the ITU language definition the keyword `call` in a procedure call is optional. In the SDL Suite it is, however, required.

If Statements

The structure of an if statement is:

```
if ( <Boolean expr> )
  <Statement>
else
  <Statement>
```

where the else part is optional. The Boolean expression is first calculated. If it has the value true, the first statement is executed, otherwise the else statement, if present, is executed.

Example 42

```
if (a>0)
  a := a+1;

if (a=0) {
  a := 100;
  b := b+1;
} else {
  a := a+1;
  b := 0;
}
```

If there are several possible if statements for an else path (the “dangling else” problem), the innermost if is always selected.

Example 43

```
if (a>0)
  if (b>0)
    a := a+1;
  else
    a := a-1;
```

means:

```
if (a>0) {
  if (b>0)
    a := a+1;
  else
    a := a-1;
}
```

Decision Statements

A decision statement has much in common with the ordinary decisions found in SDL, i.e. it is a multi-branch statement. The major differences between decision statements and ordinary statements is that all paths in a decision statement ends at the enddecision.

Example 44

```
decision (a) {
(1:10) : {call p(a); a := a-5;}
(<=0)  : a := a+5;
else   : a := a-5;
}
```

The decision question and the decision answers follows the same syntax and semantics as in ordinary decisions. Following an answer there should be a statement, which might be a compound statement.

Loop Statements

A loop statement is used to repeat the execution of a statement (or usually a compound statement), a number of times. The loop is controlled by a loop variable, which either can be locally defined in the loop or defined somewhere outside of the loop.

The loop control part contains three fields:

- the loop variable indicator with a start value,
- the loop test expression,
- the new loop variable value.

Example 45

```
for (a := 1, a<10, a+1)
  sum := sum+a;
```

should be interpreted as (in C-like syntax):

```
a = 1;
while (a<10) {
  sum = sum+a;
  a = a+1;
}
```

Note the difference between SDL and C when it comes to the variable update. In C this is a statement, in SDL it is an expression to be assigned to the variable mentioned in the loop variable indicator.

In the loop variable indicator, either a new variable can be defined or a previously defined variable can be used. Example:

```
for (a := 1, ...
for (dcl a integer := 1, ...
```

Other possibilities in loop statements:

- One or more of the loop control part fields can be empty. If, however the loop variable indicator (first field) is empty, then the loop variable update field (third field) must also be empty. Example:

  ```
  for (a := 1, , a+1) ..
  for ( , , ) ...
  ```

- A loop may contain several loop control parts. Example:

  ```
  for (a := 1, a<10, a+1; b := 1, b<5, b+1)
    sum := sum+a+b;
  ```

This should be interpreted as (in C like syntax):

```
a = 1;
b = 1;
while ( (a<10) and (b<5) ) {
  sum = sum+a;
  a = a+1;
  b = b+1;
}
```

- Break statements can be used to break out of a loop. See “Label Statements” on page 143 and “Jump Statements” on page 144.
- A loop statement may end with a “then” statement, which is executed if the loop is terminated because of the loop test expression becomes false. The “then” statement is not executed if the loop is terminated due to a break statement. Example:

```
ok := false;
for (a:=1, a<10, a+1) {
  sum := sum+arr(a);
  if (sum > limit) break;
}
then
  ok := true;
```

Label Statements

A label statement is just a label followed by a statement. These labels are only of interest if the statement following the label is a loop statement. The label name can be used in break statements (see below) to break out of a loop statement. Example:

```
L:
for (i:=0, i<10, i+1)
  sum := sum+a(i);
```

Note:

There are no “join” or “goto” statements allowed in the algorithmic extensions to SDL.

Jump Statements

Jump statements, i.e. `break` and `continue`, are used to change the execution flow within a loop.

A continue statement, which only may occur within a loop, is defined as: “skip the remaining part of the loop body and continue with updating the loop variable to its next value.”

Example 46

```
for (a:=1, a<10, a+1) {
  if (sum > limit) continue;
  sum := sum+arr(a);
}
```

should be interpreted as (in C like syntax):

```
a = 1;
while (a<10) {
  if (sum > limit) goto cont;
  sum = sum + arr[a];
cont :
  a = a+1;
}
```

A break statement can be used to stop the execution of the loop and directly goto the statement after the loop.

Example 47

```
ok := false;
for (a:=1, a<10, a+1) {
  sum := sum+arr(a);
  if (sum > limit) break;
}
then
  ok := true;
```

should be interpreted as (in C like syntax):

```
ok = false;
a = 1;
while (a<10) {
  sum = sum + arr[a];
  if (sum > limit) goto brk;
  a = a+1;
}
ok = true;
brk:
```

A break statement breaks out of the innermost loop statement. By using labeled loop statements breaks out of outer loops can be achieved.

Example 48

```
L: for (x:=1, x<10, x+1) {
     a := 0;
     for (y:=1, y<10, y+1) {
       a := a+y;
       if (call test(x,y)) break L;
     }
   }
```

The break statement in the inner loop breaks out from both loops as it mentions the label for the outer loop.

Empty Statements

It is allowed to have an empty statement, represented by just writing nothing. This is sometimes useful, for example as loop statement:

```
for (i:=1, Arr(i)/=0 and i<Limit, i+1) ;
  /* This loop sets i to the index of the first zero
     element in the Array Arr. */
```

Grammar for the Algorithmic Extensions

Meta grammar:

`'dcl', ')', ';'`	are examples of terminal symbols.
`<Stmt>, <Name>`	are examples of non-terminal symbols.
`::=`	means defined as.
`$`	means used for empty.
`*`	means 0 or more occurencies.
`+`	means 1 or more occurencies.
`|`	means or.

Start of Grammar

```
<CompoundStmt> ::=
   '{' <VarDefStmt>* <Stmt>* '}'

<VarDefStmt>   ::=
   'dcl'     <Name> (',' <Name>)* <Sort> (':=' <Expr> | $)
       ( ',' <Name> (',' <Name>)* <Sort> (':=' <Expr> | $) )*
   ';'

<Stmt>         ::=
   <CompoundStmt>          |
   <Outputx> ';'           |
   <CreateRequest> ';'     |
   <Setx> ';'              |
   <Resetx> ';'            |
   <Export> ';'            |
   <Return> ';'            |
   <ProcedureCall> ';'     |
   <IfStmt>                |
   <LabelStmt>             |
   <AssignmentStatement> ';' |
   <DeciStmt>              |
   <LoopStmt>              |
   <JumpStmt> ';'          |
   <EmptyStmt> ';'

<IfStmt>       ::=
   'if' '(' <Expr> ')' <Stmt> ('else' <Stmt> | $)

<DecisionStmt> ::=
   'decision' '(' <Expr> ')' '{'
   ( <Answer> <Stmt> )+
   ( 'else'   <Stmt> | $)
   '}'
```

```
<Answer>        ::= same as answer in ordinary decisions

<LoopStmt>      ::=
   'for' '(' (<LoopClause> (';' <LoopClause>)* | $) ')'
     <Stmt>
   ('then' <Stmt> | $)

<LoopClause>    ::=
   (<LoopVarInd> | $) ',' (<Expr> | $) ',' (<Expr> | $)

<LoopVarInd>    ::=
   'dcl' <Name> <Sort> ':=' <Expr> |
   <Identifier> (':=' <Expr> | $)

<LabelStmt>     ::=
   <Label> <Stmt>

<JumpStmt>      ::=
   <Break> (<Name> / $) |
   <Continue>

<EmptyStmt>     ::=
   $
```

End of Grammar

Algorithms in SDL Simulator/SDL Explorer

The textual trace in the SDL Simulator and the SDL Explorer for the new algorithmic extensions will be according to the table below.

Statement	Textual trace	Comment
Compound		No trace

If	`IF (true)`
	`IF (false)`	

Decision	`DECISION  Value: 7`	Same trace as for ordinary decisions

Loop	`LOOP variable b := 3`
	`LOOP test TRUE`
	`LOOP test FALSE`
		For loop variable assignments
		For loop tests

Jump	`CONTINUE`
	`BREAK`
	`BREAK LoopName`	

Empty		No trace

A compound statement without variables declarations is seen as just a sequence of statements, while a compound statement with variable declarations is seen as a procedure call of a procedure with no name, without parameters. However, no trace information is produced for this implicit procedure call or procedure return.

When it comes to variables, these are available in the simulator interface just in the same way as if compound statements where true procedures. That is, the commands *Up* and *Down* can be used to view variables in different scopes. Note that a variable defined in a loop variable indicator introduces a scope of its own.

There is one exception of this general treatment of variables in local scopes and that is procedures defined as a compound statement.

In this procedure:

```
procedure p
  fpar in/out a integer
{
  dcl b integer;
  ...
}
```

the parameter `a` and the variable `b` will be in the same scope, the procedure scope. For compound statements within the outermost procedure scope the general rules above apply.

Execution Performance in Applications

Cadvanced

All concepts in the algorithmic extensions have efficient C implementations, except variable declarations in local scopes (including in a loop variable indicator), as such compound statements will become SDL procedure calls.

Cmicro

All concepts in the algorithmic extensions have efficient C implementations. Compound statements containing variables are implemented using C block statements.

Chapter
4

Organizing a Project

This chapter explains how diagrams and other documents that can be handled in the Organizer, may be managed in a project including several project members. The functionality of the SDL Suite related to management of diagrams in a project environment is provided by the Organizer. A reference manual for the Organizer can be found in chapter 2, *The Organizer, in the User´s Manual*.

Chapter 4 Organizing a Project

Introduction

General

Software development projects are typically staffed by a large number of software engineers, each working simultaneously on different parts of the same system. This requires careful coordination and version control of the different parts of the system. Normally, stable (well-defined) versions of a system are stored in a central storage area (in this chapter named the *original area*) accessible for each user. The original area is typically either:

- A directory containing different versions of the system managed by a version or revision control system, e.g. RCS, CM SYNERGY or ClearCase.
- An ordered set of directories, where each directory represents one version of the system and contains the information that has changed as compared to the previous version. For example, version 1 of the system PBX may be stored in the directory PBX-1 which contains version 1 of all parts of the system PBX. Version 2 of the system PBX is then stored in the directory PBX-2 which contains only the parts of the system that has changed in comparison to PBX-1.

In addition, each project member may have his own *work area* where he temporarily stores the parts of the system he is currently working on. Furthermore, a project may also use a reference area which contains one or more packages of information that are shared between different projects.

This way of working is supported by the Organizer tool. It allows the members of a project to work independently, but in a coordinated and structured way. In particular, the tool supports:

- Individual customization of the user environment.
- Easy visualization of the SDL system's contents, in terms of SDL packages and diagrams, Message Sequence Charts and physical files.
- Updating of versions from each individual user's work area to the shared original area.

- Copying of SDL diagrams and Message Sequence Charts between different storage areas.

Improved Support

To facilitate the management of more complex projects, the SDL Suite can be integrated to control systems that are used to manage versions and revisions. Example of such systems are RCS, CM SYNERGY and ClearCase. The Organizer can easily be customized in order to extend the menus commands with support for “check-in”, “check-out” and “update” operations. The extensions are defined in textual files that contain the definitions of the names of the menus and commands, what operations they will call in the control system, what objects are the subject of the operation, etc.

The rest of this chapter is organized as follows:

- “Diagram Binding” on page 152 describes the principles for how diagrams and other documents are connected to files.
- “How to Manage the Diagrams in a Project” on page 155 describes how the tools can be used to manage the SDL diagrams in a project environment.

Diagram Binding

The Organizer provides the ability to manage the diagrams and other documents that build up a project, without the need for an external version or revision control system.

Important is that one diagram is represented by one file, and the mapping between diagrams and files are kept in the *system file*. The system file is the file that is opened when issuing the *Open* command in the Organizer. The system file has by default the extension `.sdt`.

A diagram may be bound to a physical file using the Organizer by:

- Automatic Binding
- Manual Binding

Automatic Binding

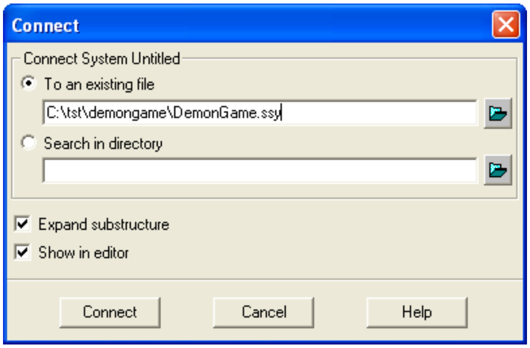
Diagrams may be bound automatically when the diagram is saved for the first time. The name of the physical file will by default be `<diagram name>.<ext>`, where <*diagram name*> is the name of the diagram and <*ext*> is a suggested extension of three letters indicating what type of diagram is contained in the file. See "Save" on page 11 in chapter 1, *User Interface and Basic Operations, in the User's Manual* for information on default file names and extensions.

Example 49

The block diagram with the name DemonBlock will by default be bound to the file `demonblock.sbk`

Manual Binding

Diagrams and files may be bound manually, on demand, via the command *Connect*. A file selection dialog appears and the selected diagram may be connected to an existing file. Another possibility is to look for the diagram file in a certain directory.



Connect

Connect System Untitled

[✓ To an existing file]

C:\tst\demongame\DemonGame.ssy

Search in directory

[✓ Expand substructure]

[✓ Show in editor]

Figure 54: Connecting a diagram to an existing file

When connecting a diagram, there is a possibility to automatically connect all diagrams in a possible substructure, the *Expand Substructure* option. The names of the files are not important when mapping files automatically, using this feature, since the diagram name is stored in the file. What is important though, is the extension of the file name which must correspond to the type of diagram stored in the file.

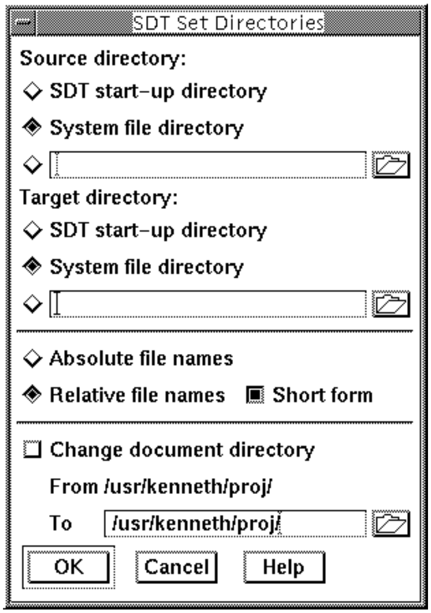
Source and Target Directories

Two settings in the Organizer are related to where diagrams are stored. The two settings are:

- *Source Directory*
  The directory where new diagrams are stored.

- *Target Directory*
  The directory where generated files are stored. Generated files do not include the source files, i.e. the files included in the Organizer Diagram Structure.

The source directory and the target directory may be changed in the Organizer by using the *Set Directories* command. In the *Set Directories* dialog there is also an option to either show the absolute path to files, or to show the path relative the Source directory. For a complete description of the functionality of this command, please see the “Set Directories” on page 71 in chapter 2, *The Organizer, in the User´s Manual*.



SDT Set Directories

Source directory:
- SDT start-up directory
- [✓ System file directory]
-

Target directory:
- SDT start-up directory
- [✓ System file directory]
-

- Absolute file names
- [✓ Relative file names] [✓ Short form]

- Change document directory

From /usr/kenneth/proj/

To /usr/kenneth/proj/

Figure 55: Specifying the Source and Target directories

This affects the way paths to files are stored in the system file:

- If the option *Absolute file names* is chosen, then the absolute path will be stored in the system file.
- If the option *Relative file names* is chosen, then only the relative path will be stored in the system file. This is important if the source directory should be moveable, in which case this option should be selected.

How to Manage the Diagrams in a Project

One way to manage the diagrams in a project is to have the original area managed by some external configuration or version control system, e.g. RCS (Revision Control System), which is available on most computer systems. In that case, every member of the project has his own work area. In each work area, there is a system file (or perhaps multiple system files, each one containing a different part of the system). All diagrams in the system file are bound to files in the users' work area. The relationship between files in the users' work area and files in the original area is managed by the external configuration or version control system.

The rest of this section describes how to use RCS as a version control system. For more information about SDL Suite/RCS Integration, see Readme files in `<inst dir>\examples\cm\win32\rcs\` or `<inst dir>/examples/cm/unix/rcs/`. How to use CM SYNERGY is described in "Using CM SYNERGY Together with an SDL System" on page 162. How to instead use ClearCase is described in "Using CM SYNERGY Together with an SDL System" on page 162.

The control is based on the following mechanisms:

- The locking functionality in RCS, by using the check in and check out commands.
- The multi user support in the Organizer, facilitated by the *Control Unit Files.*
- The possibility to add own menu commands to the SDL Suite, commands that in this case should connect the Organizer to RCS.

Note:

The following instructions and descriptions only addresses a **UNIX** system with the **UNIX** version of RCS.

Starting to Use RCS Together with an SDL System

When handling of files managed by RCS is to be introduced for an existing SDL system, the following approach can be used:

1. Have all the files related to your system placed in one directory hierarchy which we call the *work area*.
2. Set the *Source Directory* in the Organizer to the work area directory.
3. Now we create the *original area* for the system as the RCS file database. The work area and the original area should have the same directory structure for a given diagram system. The original area will contain the RCS directories that hold the revision files, but these directories are empty for now. (Note that the work area does **not** have any RCS directories.)

Example 50: Work and original area for the DemonGame system——

The work area has three directories:

- One for the files related to the system.
- One for each of the two blocks (GameBlock and DemonBlock):

```
demonblock      gameblock       system

demonblock:
demon.spr       demonblock.sbk

gameblock:
game.spr        gameblock.sbk   main.spr

system:
demongame.msc   demongame.ssy   systemlevel.msc
```

The original area for the Demon Game system (created in a different place in the file system) will then have the following structure:

```
RCS      demonblock      gameblock       system

demongame/RCS:

demongame/demonblock:
RCS

demongame/demonblock/RCS:
```

```
demongame/gameblock:
RCS

demongame/gameblock/RCS:

demongame/system:
RCS

demongame/system/RCS:
```

4. Define the root directory of the original area (**`setenv rcsroot <directoryname>`**) and load the RCS menu-set into the Organizer (run **`$telelogic/sdt/examples/RCS/sdtrcs.mnu`**). A new menu titled *RCS* appears in the Organizer menu bar.
   - The *RCS* menu in the Organizer is an example of how check in and check out can be done on one diagram and a hierarchy of diagrams. (The commands that are defined in the example could easily be tailored to fit specific needs).
5. Check in the first version of each diagram file, using the Organizer commands *Recursive Check In* or *Check In*.
   - To visualize the results, make sure the file permissions are visible in the Organizer (Choose the *View Options* command in the *View* menu and select the option *File access permissions*.
6. After a successful check in, the original area is now populated with the RCS files.

Example 51: The Original Area, now populated

```
RCS       demonblock       gameblock        system

RCS:

demonblock:
RCS

demonblock/RCS:
demon.spr,v              demonblock.sbk,v

gameblock:
RCS

gameblock/RCS:
game.spr,v       gameblock.sbk,v main.spr,v

system:
```

```
RCS

system/RCS:
demongame.msc,v          demongame.ssy,v
systemlevel.msc,v
```

Using the SDL Suite and RCS in a Multi User Environment

If the system is developed by a group of developers it is useful to partition it and assign *Control Unit Files*, for the different partitions. For the DemonGame example, let us assume that the DemonBlock is developed by one developer and the GameBlock by another. The procedure here is to create:

- a top level control unit file,
- a control unit file for the DemonGame system,
- a control unit file for the DemonBlock, and
- a control unit file for the GameBlock.

1. In order to assign a control unit file for an Organizer object we select the object and execute the *Configuration > Group File* command from the *Edit* menu.

2. When the four control units above are assigned, we save the system from the Organizer (which also saves the control units on their respective `.scu` files).

3. Now, we can check in the created `.scu` files. It is wise to have the control unit file located in the same directory as the corresponding object file.

Example 52: The Original Area with Control Unit Files

The DemonGame original area with checked in control unit files:

```
RCS                demonblock      gameblock
system

alfa/RCS:
demongame.scu,v

alfa/demonblock:
RCS

alfa/demonblock/RCS:
DemonBlock.scu,v        demon.spr,v
```

```
demonblock.sbk,v

alfa/gameblock:
RCS

alfa/gameblock/RCS:
GameBlock.scu,v game.spr,v      gameblock.sbk,v
main.spr,v

alfa/system:
RCS

alfa/system/RCS:
DemonGame.scu,v          demongame.ssy,v
demongame.msc,v          systemlevel.msc,v
```

Make Local Changes Global Using RCS

When a user has made changes in files that have been checked out and locked and wants to make these changes global for the complete project, the following steps should be performed.

- Use *Check In* or *Recursive Check In* from the Organizer *RCS* menu, on the diagrams that have been modified. Note that *Recursive check In* will automatically handle control unit files as well.

Example 53: Adding a Diagram to the Control Unit and Saving

Say that a new process diagram is added to the GameBlock:

1. The user must therefore check out the files `gameblock.sbk` and `GameBlock.scu`.
2. With the SDL Editor, the diagram GameBlock is updated to contain the new process reference symbol.
3. When saving from the Organizer, `GameBlock.scu` will automatically be updated with the new process diagram file name. Note also that all changes will be local to the partition of the diagram system that represents the GameBlock and that no global update of the system file is needed.

Make Global Changes Local Using RCS

If a user is notified that there have been changes to the system, the following should be done in order to update the users' local system:

- Use *Check Out* or *Recursive Check Out* to update the diagrams.
  - From Example 53 above, if the GameBlock has been modified a *Recursive Check Out* on the GameBlock object will bring in the newest checked in version.

Building and Populating a Work Area from RCS based Original Area

Say that we have an original area with its RCS directories containing the checked in diagrams and the control unit files. A developer that is new to the project wants to make a complete update of a system in his work area to get an up-to-date view.

1. The developer must first create/update his work area directory tree. The environment variable `rcsroot` should be defined to point to the original area.
2. In an empty Organizer: the RCS menus are loaded (use for instance the command **`$telelogic/sdt/examples/RCS/sdtrcs.mnu`**), the *Source Directory* is set to the work area, and the top level control unit file is associated with the SDT symbol (using the Organizer *Configuration > Group File* command).
   - Note that the top level control unit file must be assigned the correct file name that is checked in. For the DemonGame example, the top level control unit file name would be `DemonGame.scu` in the work area top directory.
3. To populate/update the work area, select the SDT symbol in the Organizer and perform *Recursive Check Out.*
4. The new developer saves the system file to get a personal view of the diagram system.
   - When saving the system file, the Organizer tries to save the top level control unit file; this file is read-only and the Organizer will hence issue an error message which can be disregarded.

Endpoint Handling with RCS

Presently there is **one** global link file (the *master link file*) that holds the link information for a given system. A group of developers can coordinate changes to this file with the help of the RCS check out and lock facility, where only one developer can update the file at a time. A *local link file* is used for temporary storage of changes made to the endpoint and link information, until the user decides to make his changes global. Say we have checked in the first version of the link file into the original area. In order to modify the link file, the procedure would follow these guidelines:

1. Issue the command *Add Local Link File* in the RCS menu.
2. All changes to the endpoint and link information will now be stored in the local link file, and the master link file will be left unchanged.
3. When it is time to update the global link information, check out the master link file by issuing the command *Check Out And Lock Link File*.
4. Issue the command *Merge Local Link File*, which will update the master link file with the information from the local link file.
5. Check in the master link file by issuing the command *Check In Link File*.

Simultaneous Editing of an SDL Diagram

There is a possibility for several users to work simultaneously on the same SDL diagram by utilizing the commands *Split* and *Join*, available in the *Tools* menu of the Organizer.

By the Split command, a diagram with several pages can be saved in several files, with disjoint sets of pages, thus enabling several users to edit them independently. The Join command simply merges two SDL diagrams of the same kind.

Using CM SYNERGY Together with an SDL System

This section describes one way of using the SDL Suite and CM SYNERGY in an integrated way.

For more information about the SDL Suite/CM SYNERGY Integration see the `Readme` file in the installation `<SDL Suite inst dir>\examples\cm\win32\cmsynergy\` or `<SDL Suite inst dir>/examples/cm/unix/cmsynergy/`.

For a more general description of using a configuration or version control system, please see "How to Manage the Diagrams in a Project" on page 155.

Introducing CM SYNERGY with the SDL Suite - Migration

When handling of files by CM SYNERGY is to be applied on a system being developed by the SDL Suite, the following approach can be used.

1. Make sure that all files (related to the system) are configured in one directory hierarchy outside CM SYNERGY.
2. Install the CM SYNERGY menu into the Organizer. Add `cmsynergy.ini` to your `org-menus.ini` file.

   The SDL Suite will search for the `org-menus.ini` first in the directory where the SDL Suite was started, then in a directory pointed to by the HOME environment variable and finally in the directory in which the SDL Suite was installed.

   If you do not already have a `org-menus.ini` file, the `cmsynergy.ini` can serve as one. Just copy `cmsynergy.ini` to either your HOME directory or to where you have the SDL Suite installed and rename it to `org-menus.ini`. For more information on dynamic menus, see "Defining Menus in the SDL Suite" on page 18 in chapter 1, *User Interface and Basic Operations*. (The CM SYNERGY menu that comes with the distribution of the SDL Suite is an example and could be tailored by the user.)

3. Set your path environment variable to include
   - `<CM SYNERGY inst dir>\bin` or `<CM SYNERGY inst dir>/bin`, and
   - `<SDL Suite inst dir>\bin\wini386` or `<SDL Suite inst dir>/bin`.

   This is necessary to be able to start the tools from each other directly.

4. Start the CM SYNERGY Client.

5. Change role to ccm_admin.

6. Display the *Admin>Type Definition* dialog to define an SDL file type. Enter the following values:

   Type Name: SDL
   Description: Binary file
   Super Type: binary
   Initial Status: working
   Require Task at:<none>

   Verify Comment Existence on Promote / Check In: Off
   Allow Update during Model Install: Off
   Associate with a File in the File System: On
   Can be a Product File: Off

   Icon Color / Fil: binary.bmp
   File Name Extension: .ssy

7. Display the *Type>Modify File Operations* dialog, and enter the following values:

   (If necessary, replace `sdt` below with the command you use to start SDL Suite. If the sdt script is not in your PATH, you should specify a full path.)

   Type Name: SDL
   Description: Binary file
   Command Templates:
   Graphical User Interface:

   Edit: sdt %file1

View: sdt %file1
Compare: sdt -fg -grdiff %file1 %file2
Merge: sdt -fg -grdiff %file1 %file2 -mergeto %outfile

Command line interface:
Edit: sdt %file1
View: sdt %file1
Compare: sdt -fg -grdiff %file1 %file2
Merge: sdt -fg -grdiff %file1 %file2 -mergeto %outfile

Print:
Compare Attribute: source

8. Click *OK*.
9. Click *Update Type*.
10. Define a project file type (*.sdt) by entering the following values:

    Type Name: SDT
    Description: SDT project file
    Super Type: ascii
    Initial Status: working
    Require Task at:<none>

    Verify Comment Existence on Promote / Check In: Off
    Allow Update during Model Install: Off
    Associate with a File in the File System: On
    Can be a Product File: Off

    Icon Color / Fil: ascii.bmp
    File Name Extension: .sdt

11. Display the *Type>Modify File Operations* dialog, and enter the following values:

    (If necessary, replace `sdt` below with the command you use to start SDL Suite. If the sdt script is not in your PATH, you should specify a full path.)

    Type Name: SDT
    Description: SDT project file

Command Templates:
Graphical User Interface:

Edit: sdt %file1
View: sdt %file1
Compare: %FAIL
Merge: %FAIL

Command line interface:
Edit: sdt %file1
View: sdt %file1
Compare: %FAIL
Merge: %FAIL

Print:
Compare Attribute: source

12. Click *OK*.
13. Click *Update Type*.
14. Close the dialog. (*File>Close*).

    Now you have set up CM SYNERGY for SDL system diagram and SDT project files.

15. To manage the other SDL diagram types select *Tools>Migrate>Options>Set>Edit* to open up the migrate rules file (`migrate.rul`) in a text editor.
16. After the entry for your new type which should look something like (might differ depending on platform):

```
MAP_FILE_TO_TYPE .*Ž[Ss][Ss][Yy]$ SDL # Created auto-
matically ...
MAP_FILE_TO_TYPE .*Ž[Ss][Dd][Tt]$ SDT # Created auto-
matically ...
```

add the following lines:

```
MAP_FILE_TO_TYPE .*Ž[Ss][Bb][Kk]$ SDL
MAP_FILE_TO_TYPE .*Ž[Ss][Pp][Rr]$ SDL
MAP_FILE_TO_TYPE .*Ž[Ss][Pp][Dd]$ SDL
MAP_FILE_TO_TYPE .*Ž[Ss][Uu][Nn]$ SDL
MAP_FILE_TO_TYPE .*Ž[Ss][Oo][Pp]$ SDL
MAP_FILE_TO_TYPE .*Ž[Ss][Ss][Tt]$ SDL
MAP_FILE_TO_TYPE .*Ž[Ss][Bb][Tt]$ SDL
MAP_FILE_TO_TYPE .*Ž[Ss][Ss][Uu]$ SDL
MAP_FILE_TO_TYPE .*Ž[Ss][Pp][Tt]$ SDL
MAP_FILE_TO_TYPE .*Ž[Ss][Vv][Tt]$ SDL
MAP_FILE_TO_TYPE .*Ž[Ss][Ss][Vv]$ SDL
MAP_FILE_TO_TYPE .*Ž[Mm][Ss][Cc]$ SDL
```

and save the file. Be very careful with the syntax in this file as it is sensitive to syntax errors. Close the open dialogs.

17. Use the *Migrate* facility to load the files into CM SYNERGY, via *Tools>Migrate*.

    – Put in the path to your Project directory in the *From Directory* field, or use the browse button to locate them.

    – In the *Project* field use the syntax ProjectName-Version, make sure to set the version to a meaningful value.

    – Next click on the menu item *Options>Set>Set Object State To>released* and click *OK*.

    – **Windows only:** To be able to change files you must set the option *Make Copies Modifiable* in *Work Area Properties* and then click *OK*.

    – To ensure that the correct files are being migrated it is suggested that you click *Preview*. If the results are what you expected go ahead and click *Load*.

    Now you have a project that is baselined in a released state which you can start to work from.

18. Each developer will now need to set up their own working environment before using CM SYNERGY with SDL Suite and TTCN Suite.

Introducing CM SYNERGY with the SDL Suite - Set up your working environment

After the build manager has migrated your SDL system into CM SYNERGY, each developer will need to set up their own CM SYNERGY work-area. This should be done using the CM SYNERGY client to ensure that all the correct options are selected.

1. Select the correct project baseline for your SDL system as directed by your build manger.
2. Check out your own personal working version of the project hierarchy using the correct options as directed by your build manager, check out the system file (this will be your own personal version).

Note:

You will normally only have to set up a working environment in this way once for each major release of your software.

3. Load the CM SYNERGY menu into the Organizer <org-menus.ini>

Introducing CM SYNERGY with the SDL Suite - Day-to-Day Working with CM SYNERGY

These steps assume that you will be using the (recommended) CM SYNERGY Task Based-CM methodology. They may be executed from the Organizer.

1. Select your SDL system and start CM SYNERGY from the Organizer.
2. Set your default task (also known as the current task), create one if required.
3. Check out the file(s) you wish to work on.
4. Edit and test as required.
5. Check in your (default) task.

Note:

If you are working as part of a team you may pick up your colleagues' latest work by selecting the top level directory and using the *Update* command (from time to time you may need to do a full update from the CM SYNERGY client - ask your build manager).

Using ClearCase Together with an SDL System

This section describes one way of using the SDL Suite and ClearCase in an integrated way. For more information about SDL Suite/ClearCase Integration, see `Readme` files in
`<inst dir>\examples\cm\win32\clearcase\` or
`<inst dir>/examples/cm/unix/clearcase/`.

For a more general description of using a configuration or version control system, please see “How to Manage the Diagrams in a Project” on page 155.

Introducing ClearCase with the SDL Suite – Checking in Files

When handling of files by ClearCase is to be applied on a system being developed by the SDL Suite, the following approach can be used.

1. Make sure that all files (related to the system) are configured in one directory hierarchy outside ClearCase. See the work area part in Example 50 on page 156.
   – Note that when working with ClearCase, the original area and work area point to the **same directory** – the top level directory for the diagram system.
2. Install the ClearCase menu into the Organizer. Please add clearcase.ini to your org-menus.ini file. The SDL Suite will search for the org-menus.ini first in the directory where the SDL Suite was started, then in a directory pointed to by the HOME environment variable and finally in the directory in which the SDL Suite was installed.If you do not already have a org-menus.ini file, the clearcase.ini can serve as one. Just copy clearcase.ini to either your HOME directory or to where you have the SDL Suite installed and rename it to `org-menus.ini`. For more information on dynamic menus, please see “Defining Menus in the SDL Suite” on page 18 in chapter 1, *User Interface and Basic Operations*. (The ClearCase menu that comes with the distribution of the SDL Suite is an example and could be tailored by the user.)

3. Set an appropriate ClearCase view. Copy the system file to the top level directory of the ClearCase file system. You may have to edit it in order to remove the line defining `SourceDirectory`.
4. *Open* the system file to bring up the structural view of the system (the diagrams are marked as invalid in the Organizer – this is OK for now).
5. The *MkDir for Object* menu command can be used to create the directory structure in a ClearCase VOB. Select an object in the Organizer and execute the *MkDir for Object* menu command to create the directory for the selected object in the ClearCase VOB.
6. Now you can populate the ClearCase directory structure with the diagram files, by copying the files from the directory in step 1. above.
7. Re-open the system file in the Organizer. All diagrams should now be connected to their files.
8. Select the *System File* icon and do the *Recursive MkElem* menu choice to create all the objects in the ClearCase VOB.
   – Note that no ClearCase element is created for the system file.
9. Select the *System File* icon and the *Recursive Check In* will check in all objects into the ClearCase VOB.
10. The directories must be checked in separately. Use the command *Check In Directory* to do that.

The system file for the diagram system can be checked in but it is not suitable for version control since the Organizer wants to update it in situations unrelated to revision changes. The system file should be regarded as one developer's personal view of the system being developed. On the other hand, the top level control unit file should be checked in and is suitable to be put under revision control.

Introducing ClearCase with the SDL Suite – Opening a System

The top level control unit file allows to load the system into the Organizer if a user starts from scratch. Say that there is a checked in diagram system in a ClearCase VOB. A developer that wants to start working on that diagram system has to do the following.

1. Mount the ClearCase VOB containing the diagram system.
2. Set the appropriate ClearCase view and start the SDL Suite in it on a new system.
3. Load the ClearCase menu into the Organizer
   `$telelogic/sdt/examples/ClearCase/sdtcc.mnu`
4. Set the *Source Directory* to the top level directory for the diagram system.
5. *Connect* the top level control unit file with the *System File* icon and run the *Recursive Update ClearCase* menu command. This loads the system into the Organizer.
6. *Save* the system file. (The Organizer will warn that the top level control unit file is read only but this can be disregarded.)

A

Access Control system (example): 2
Algorithmic extensions to SDL: 137
Array (SDL generator): 71
ASN.1 encoding rules: 122
ASN.1, using in SDL: 116
B

Bag (SDL generator): 79
Bit (SDL sort): 44
Bit field (in SDL structs): 65
Bit_String (SDL sort): 45
Boolean (SDL sort): 48
BREAK statement (in SDL): 144
C

C, using in SDL: 84
CArray (SDL generator): 110
Character (SDL sort): 49
CharStar (SDL sort): 109
Charstring (SDL sort): 50
Choice (SDL): 68
CM Synergy, integrating with the SDL Suite: 162
Compound statement (in SDL): 138
Configuration management systems, integrating with the SDL Suite: 151
CONTINUE statement (in SDL): 144
Control unit files, using: 158
CPP2SDL: 84
ctypes package: 108
D

Data types (ASN.1 in SDL): 116
Data types (C in SDL): 84
Data types in SDL: 42
Decision statement (in SDL): 141
Default value (in SDL sorts): 82
Diagram, binding to physical file: 152
Duration (SDL sort): 53
E

Encoding rules (ASN.1): 122
Enumeration sorts (in SDL): 63
Exported procedure (SDL concept): 22
F

Files, binding to diagrams: 152
Float (SDL sort): 109
FOR statement (in SDL): 142

G

Generators (in SDL): 71
H

H2SDL: 84
I

IA5String (SDL sort): 52
IF statement (in SDL): 140
Import specification: 101
Inherits (SDL sorts): 69
Integer (SDL sort): 54
J

Jump statements (in SDL): 144
L

Label statement (in SDL): 143
Literals (in SDL sorts): 80
Local link file: 161
Local variables (in SDL): 139
LongInt (SDL sort): 109
Loop statement (in SDL): 142
M

Master link file: 161
Multiuser support: 158
N

Natural (SDL sort): 54
Null (SDL sort): 55
NumericString (SDL sort): 52
O

Object_Identifier (SDL sort): 55
Object-orientation in SDL: 13
Octet (SDL sort): 58
Octet_String (SDL sort): 59
Operator diagrams: 81
Operators (in SDL sorts): 81
Original area (storage area): 150
P

Package (SDL concept): 39
PId (SDL sort): 61
Powerset (SDL generator): 76
PrintableString (SDL sort): 52
Procedures in SDL, usage: 19
Project organization in the SDL Suite: 150

R

range condition (in SDL data types): 62
RCS, integrating with the SDL Suite: 156
Real (SDL sort): 61
Ref (SDL generator): 111
Remote procedure (SDL concept): 22
Revision control systems, integrating with the SDL Suite: 151

S

SDL algorithms: 137
ShortInt (SDL sort): 109
Simulation, textual trace: 147
Sorts in SDL: 42
Source directory, using: 153
Specialization (SDL concept): 27
String (SDL generator): 73
Struct (SDL): 64
Syntypes (in SDL): 62

T

Target directory, using: 153
Time (SDL sort): 53
Trace (Explorer): 147
Trace (Simulator), algorithms in SDL: 147
TTCN Link, sharing data between SDL and TTCN: 123
Type conversions, implicit in Analyzer: 134

U

UnsignedInt (SDL sort): 109
UnsignedLongInt (SDL sort): 109
UnsignedShortInt (SDL sort): 109
User-defined sorts (SDL): 62

V

Value-returning procedures (in SDL): 25
Version control systems, integrating with the SDL Suite: 151
VisibleString (SDL sort): 52
VoidStar (SDL sort): 109
VoidStarStar (SDL sort): 109

W

Work area (storage area): 150