

## *Object Design*

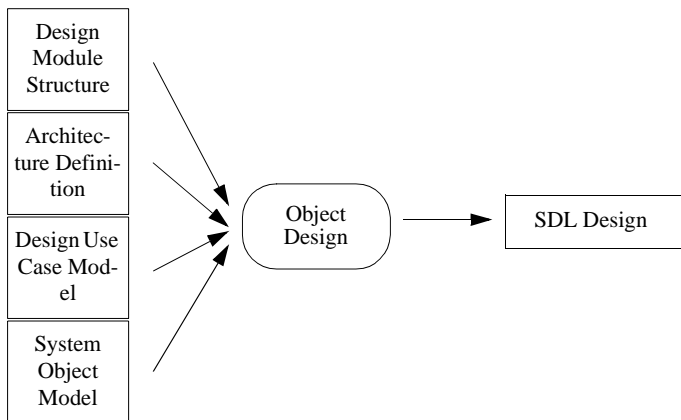
**This chapter gives you the rules for how to transform object models to SDL design models. Mappings for active and passive objects and associations are described and exemplified. How to define the behavior of an object is also discussed as well as a section on how to test the design.**

**The chapter requires that you are at least reasonably familiar with SDL.**

## Object Design Overview

The object design is the activity that completes the definition of the system. It carries on where the system design finished and fills in the details about object behavior and system structure. The object design is a creative activity that refines the object definitions and system structure taking at least three aspects into account:

- The system structure: where in the system is an object localized
- The reuse structure: is the object a subject for reuse internally within the system or externally
- The precise definition of internal object structure, the behavior and data aspects



*Figure 655: Overview of the object design activity*

There is a close relation between the system design and the object design, in the sense that both activities are dealing with the structure of the system. The difference between the activities is mainly a difference in the point of view. The system design takes a top-down look at the system to identify subsystems and overall structures. The object design on the other hand focuses on the objects and uses them as a starting point for the refinement and structuring.

# Object Design Overview

---

In practice one of the major tasks of the object design is to define the behavior of the objects. Essentially the object design can be viewed as three sequential tasks that must be performed.

1. Map all the classes in the relevant part of the analysis object model to suitable SDL concepts.
  - Active classes are mapped to processes or process types and passive classes are mapped to data types. This mapping is described in “Mapping Object Models to SDL Design Models” on page 3774.
  - The localization of the process (type) or data type in terms of the SDL package/system structure should be done according to the design module structure and architecture definition from the system design.
2. Choose a set of essential use cases and define the behavior of the SDL processes and data types that implements these use cases. Concentrate on the normal behaviors and leave the exceptions for a later step. The design of SDL processes is further discussed in “Describing Object Behavior” on page 3803.
  - Note that this also includes a testing activity that verifies that the SDL design implements the requirements from the use cases. This is further described in “Design Testing” on page 3810.
3. Elaborate the design by introducing more use cases and refine the SDL design to handle also these cases. Take care of exceptional situations like error handling etc.
  - This elaboration is an iterative process where the design is incrementally developed until all requirements are implemented.
  - Each iteration also includes a testing step where both newly implemented and old functionality is checked.
  - The elaboration can include both refining existing processes/data types and introducing new process/types.

In a project where the design is split on several development teams the iterations in the design will often have to be synchronized. This is further discussed in chapter 77, *SOMT Projects*.

## Mapping Object Models to SDL Design Models

In SOMT object design, an important step is the transformation of (parts of) the analysis object model to an SDL system. This section describes this transformation.

There are several aspects important in the transformation:

- How to map objects to SDL, taking into consideration e.g. whether the object is active or passive
- How to represent associations in SDL
- How to preserve the inheritance/aggregation relations in SDL

The mapping to ASN.1 will also be discussed briefly.

### Mapping an Active Object

Consider an object in the analysis model that has been found to be an active object. How should this be mapped into the design model? The default choice is to map this object as an SDL process type. The attributes of the object will then be mapped to variables and the operations will correspond either to remote procedures defined in the type or signals handled by the process type. As an example consider the DisplayInterface object in the Access control system. This object is responsible for interfacing a display that is capable of showing a line of text to a user. In the analysis object model this object may look like [Figure 656](#). It has one attribute, *Text*, and one operation, *Display*.

DisplayInterface
Text
Display

*Figure 656: The DisplayInterface object*

When mapped to an SDL diagram as a process type the result will be a process type reference, as shown in [Figure 657](#).



Figure 657: The `DisplayInterface` process type

Now consider the case where the `Display` operation was defined to be an asynchronous operation which is the default. If we now take a look at the `DisplayInterface` process type definition we can see that it has a variable called `Text` and a gate with the signal `Display`. This process type definition is shown in [Figure 658](#).

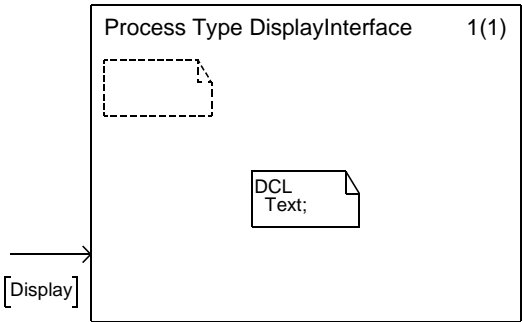
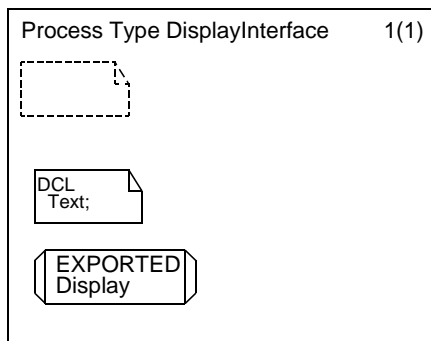


Figure 658: The `DisplayInterface` process type definition

If, on the other hand, the `Display` operation was defined to be synchronous (by giving the keyword `sync` after the operation name), then the definition of the `DisplayInterface` process type would now also be different. It would contain a definition of the `Display` remote procedure instead of a gate with a signal. See [Figure 659](#).



*Figure 659: The DisplayInterface process type when mapped to a process type with RPCs*

When mapping an object like the DisplayInterface, the design issue of reuse shows up as a question of *where* to define the SDL object. If the DisplayInterface class is considered to be a general class that is to be reused in different projects or by different development teams, the most natural choice is to put the process type in a package that includes this type definition and other related types that are to be reused in different contexts. The design choice of what packages to use in a project was one of the issues of system design.

If the DisplayInterface only is intended to be used in this project but it is used in several of the blocks representing subsystems then there is also a possibility to put the process type on the system level. This would allow the type to be used in the entire system but it would not be convenient to use it in other projects.

If the DisplayInterface is only to be used within the local block it can be defined anywhere in the scope in question, even if it probably is a good idea to keep all process type references on one page in the block diagram.

Finally, if the DisplayInterface object is only to be used in one particular place in the system, it is possible to define it as a process directly where it will be used instead of as a process type. This would be done the same way that it was mapped to a process type except that a process would be generated instead of a process type. Sometimes this strategy is a good idea since it slightly reduces the complexity of the system, however it is only possible if the object is only used in one place in the system.

### Mapping Active Objects with Inheritance

Inheritance among active objects is common in object models and is mapped directly to inheritance between the corresponding process types. For example, when mapping the EnglishDisplay object from [Figure 660](#) to a process type, the result is as shown in [Figure 661](#).

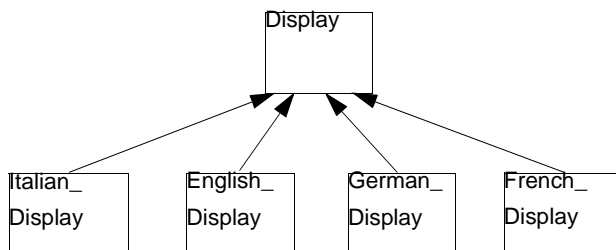


Figure 660: Object model with inheritance

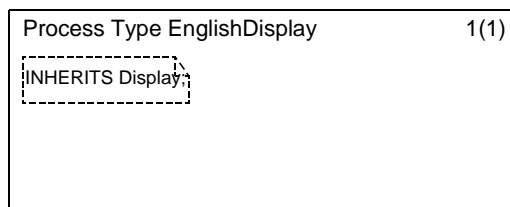


Figure 661: Process type with inheritance

### Mapping Aggregations of Active Objects

A fairly common structure is an aggregation with one assembly class that contains a number of other object classes, the part classes<sup>1</sup>. This is common for example when using container classes as abstract interfaces to a certain functionality. One example is given in a description of the software in one part of the access control system: the part that controls the access to one particular door in the building.

---

1. Terminology (assembly and parts classes) after [\[20\]](#) the “Object Model Notation Basic Concepts”

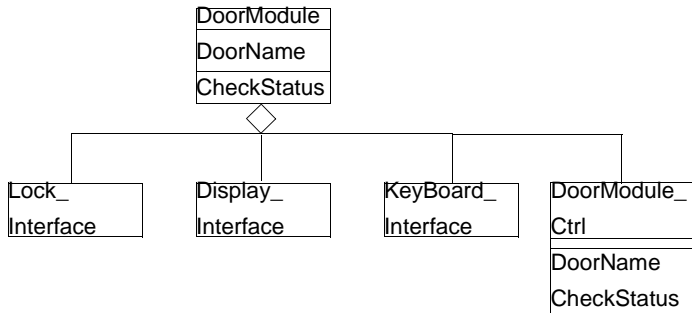


Figure 662: An example of an aggregation of active objects

The most common mapping for this type of structure is to map the assembly class to a block or a block type, depending on the strategy for reuse that has been decided upon for the object.

One special, but fairly common, case is when the assembly object class contains attributes. In this case there are two possibilities: either the aggregation is considered as a “subsystem”, in which case the attributes should be implemented by one of the parts classes, or the object should in addition to the mapping to a block (type) also be mapped to a process within the block that acts as a data server. This mapping follows the same principles as was discussed in section [“Mapping an Active Object”](#) on page 3774.

Another possibility is that the assembly object class contains operations. This implies that these operations should be included in the interface of the block/block type. In the same way as when pasting as process types there is a design choice involved here. Are the operations intended to be synchronous remote procedures or are they asynchronous signals? In this case it is also important to define where the operations are to be implemented. Also in this case there are two choices: either the operations are implemented in one of the parts classes or a process is introduced in the block that handles the operations the same way a data server process handles the attributes.

As an example consider the analysis model in [Figure 662](#). The block type DoorModule has both an attribute and an operation. A design counterpart for this is shown in [Figure 663](#) and [Figure 664](#). The process DoorModuleCtrl is the server process that handles the data and operations defined in the DoorModule analysis object.



The choice where to implement the operations and the attributes of the aggregate class is depending on the intended meaning with the aggregation. In SOMT the recommended practise is to use aggregation in object model that describes the architecture of a system as a “subsystem” or “parts vs. whole” relation, meaning the aggregate class is completely determined by its parts classes. In this case the operators and attributes of the aggregate class should be implemented by the parts classes.

Note that this is an example where the attributes and operations of the aggregate class DoorModule have been implemented by one of the part classes, the DoorModuleCtrl class.

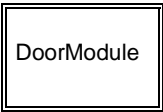


Figure 663: A block type reference for the DoorModule object

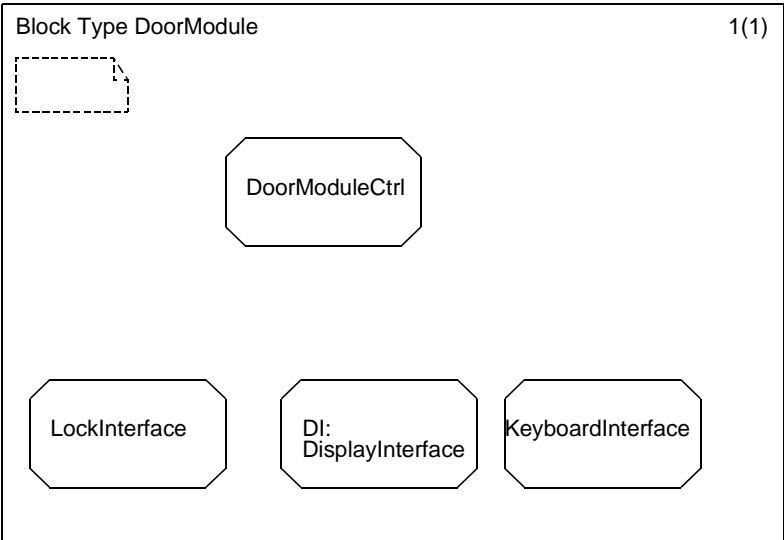


Figure 664: A block type with a data/operations server process

If the object that is mapped to a block type is a subtype of another object in the object model, then the resulting block type will be defined as a

subtype that inherits a block type that corresponds to the supertype in the object model.

## Mapping State Charts to SDL Process Graphs

The principle of reusing as much information as possible of the information gained during the analysis activities when working in the design activities implies that an effort should be made to translate state chart descriptions into SDL process graphs. For a thorough description of the mapping rules please refer to: *“Converting State Charts to SDL” on page 1658 in chapter 40, Using Diagram Editors, in the User’s Manual.*

State charts may be converted into SDL. Converting state charts without any hierarchical states is very straight forward, but converting a state chart containing hierarchical states requires flattening since the concept of hierarchical states does not exist in SDL.

The state chart LocalStation [Figure 665](#) contain a hierarchical state with sub states and need to be flattened when converted to SDL.

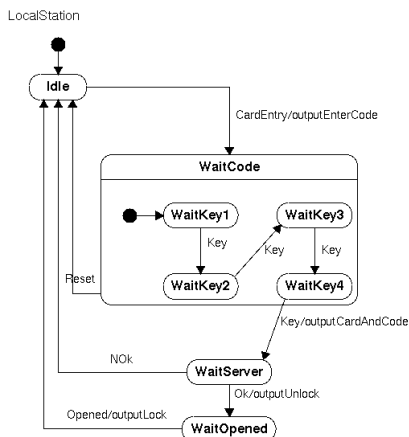


Figure 665: The State Chart Local Station

The behavior of the SDL process LocalStation [Figure 666](#) is the behavior defined in the state chart LocalStation. The process is flattened and gives a straight forward view of the behavior. To increase traceability

between the state chart and the SDL process the converter functionality provides comments on states and transitions which are related to any hierarchical state.

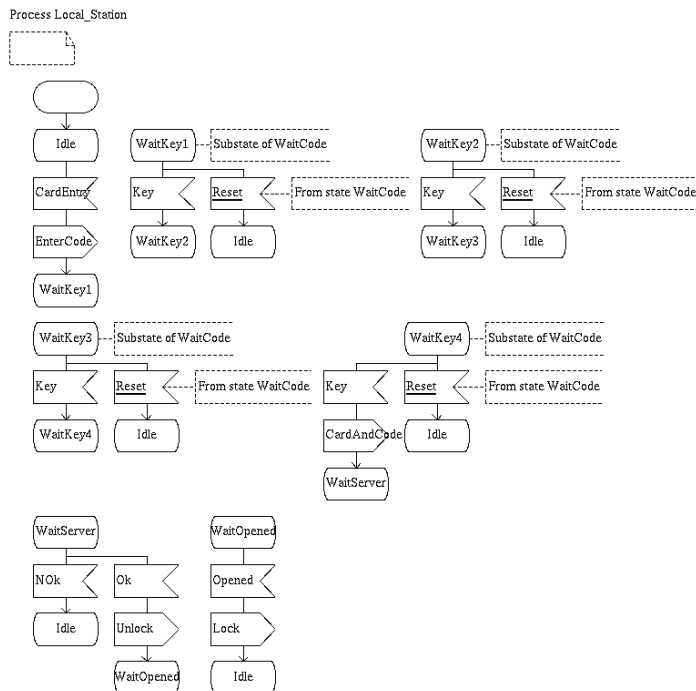


Figure 666: The SDL process LocalStation

## Mapping a Passive Object

A passive object is an object that does not have any thread of control or spontaneous behavior of its own. A passive object is often used to encapsulate a certain amount of information that is needed in the system. In the context of distributed systems it is often useful to classify the passive data descriptions into two broad classes depending on the way the data is used: *external* data and *internal* data

- The external data is focused on describing the data units that will be transported across the system when that application executes. In an application where the different components execute in different

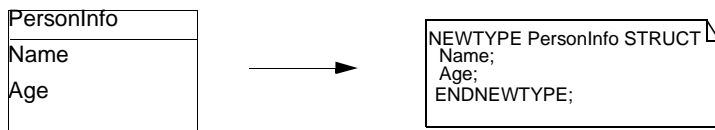
memory spaces, and maybe even on different hardware, there is a need to consider transportation format and maybe even coding/decoding of the data. Often the description of external data is focused on the information contents of the data units and not very much on the operations that can be performed on the data units. Typical example applications where external data is very common includes protocols in telecom applications. A common property of this kind of data is that they are usually structured in trees, in an object model often in aggregation hierarchies, and not in graph structures.

- The internal data is characterized by the fact that it is used to describe information that the application needs to do its work. Typically the data is localized to one concurrent execution unit. The internal data units are thus used to store rather than to transport data. Some typical examples are data bases and complex data structures in conventional program units. A local data unit is typically not copied from one component to the other. Instead operators are used to access the data unit.

As will be seen in the next sections the mapping of passive objects to SDL data types is slightly biased towards the external data view of passive objects, the mapping to ASN.1 data types is very biased to external data view, and the mapping to C data types is slightly biased towards the internal data view of passive objects.

### Mapping Objects to SDL Structs

In general, the SDL construct that can be used as the design representation for a class is an SDL STRUCT. This is exemplified in [Figure 667](#) below.



*Figure 667: Mapping a passive object to a STRUCT type*

In this mapping all attributes of the class are mapped to fields of the STRUCT.

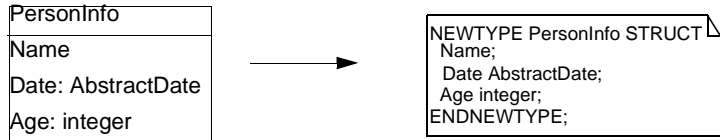
## Attribute Data Types

The attributes of a class may have data types associated with them. In general there are three ways to handle these data types:

- Defining the data type in the analysis model.
- Giving an abstract definition in the analysis model and making a final decision in the design model.
- Consider it to be a design issue and give no definition of the data type in the analysis model.

As usual the choice is a matter of personal taste but should be directed by the general idea that the analysis model must be complete enough to be understandable by itself, but as small as possible to facilitate overview and ease of use.

The default mapping in SOMT is a simple literal mapping of whatever exists in the analysis model to the design model in SDL. See [Figure 668](#).



*Figure 668: A data type mapping example*

A special case which might be of interest in the analysis is when more complex data types like lists or sets are used. SOMT treats this type of data in the same way as elementary types are treated.

## Mapping Operations

When a class is mapped to a data type in SDL there exists several ways to map the operations of the class. Operations could be mapped to:

- Operators described by (textual) operator definitions
- Operators described by operator diagrams
- Operators described using C code
- Procedures described in SDL
- External procedures

The default mapping used by SOMT is to map operations to SDL operators with operator diagrams.

### Operators Described by Operator Definitions

Operators described by operator definitions is the simplest choice. Consider the PersonInfo class as defined in [Figure 669](#).

PersonInfo
Name : charstring
Age: integer
IncreaseAge
Retired: boolean

Figure 669: A class with operations

This object has two operations: IncreaseAge and Retired which are intended to increase the age of the person with one, and to check if the person has reached the age where he/she has retired from his job.

[Figure 670](#) shows how a mapping of operations to operator definitions can be done.

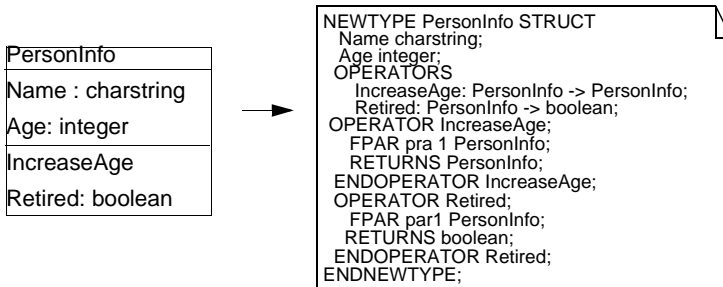


Figure 670: Data type with operator diagrams implementing the class operations

# Mapping Object Models to SDL Design Models

## Operators Described by Operator Diagrams

Operators described by operator diagrams is another alternative.

If a mapping to SDL is done using operators with operator diagrams, the SDL will look like in [Figure 671](#).

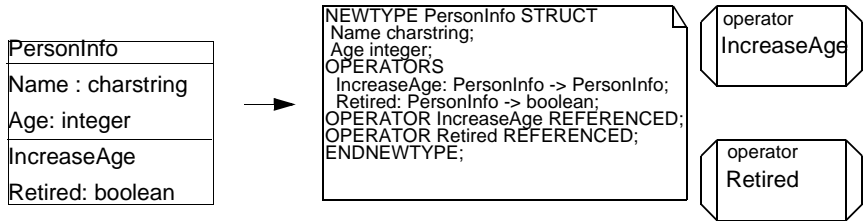


Figure 671: Data type with operator diagrams implementing the class operations

## SDL Operators in General

Some things worth noting about SDL operators are:

- An SDL operator can only be used within an expression, e.g. “var:= oper( 1 ) + 1”.
- All operators must return a result.
- Operators can not have IN/OUT parameters. All parameters are IN parameters.

A consequence of this is that an operator cannot both modify an object and return a result. The operators can thus be divided into two classes:

- Modifiers, that modify the object
- Extractors, that extract information from the object but does not modify it

The modifiers are defined as:

```
ModOp: ObjType, P1type, P2type -> ObjType;
and used as
```

```
MyObj := ModOp( MyObj, p1, p2 )
```

An extractor would be defined as:

```
ExtrOp: ObjType, P1type, P2type -> ResultType;
and used as
```

```
Result := ExtrOp( MyObj, p1, p2 )
```

However, SOMT allows a way to overcome the third aspect above. If a C implementation is used as discussed below, the restriction that operators cannot modify the parameters is somewhat relaxed, since the data type itself can be defined using a pointer and then whatever is pointed at can of course be modified by the operator.

### Operators Described Using C Code

The second mapping possibility for object model operations is to map them to SDL operators with C implementation. See [Figure 672](#).

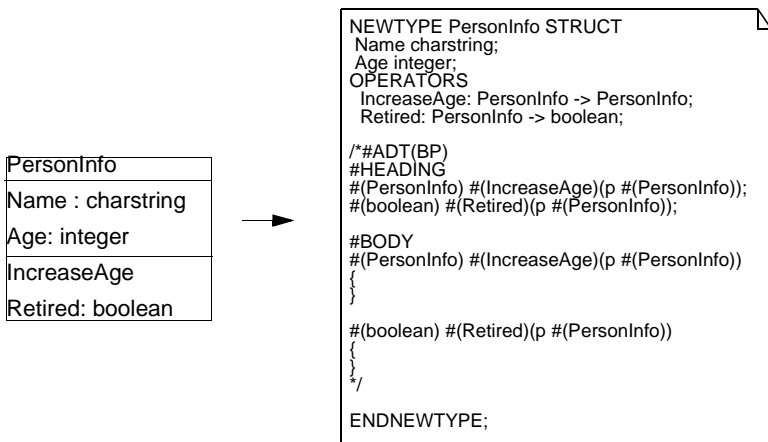


Figure 672: Data type with C implemented operators

### Procedures Described in SDL

The third way to implement operations of a class is to use SDL procedures. In this case the definition of the PersonInfo class would be like in [Figure 673](#).



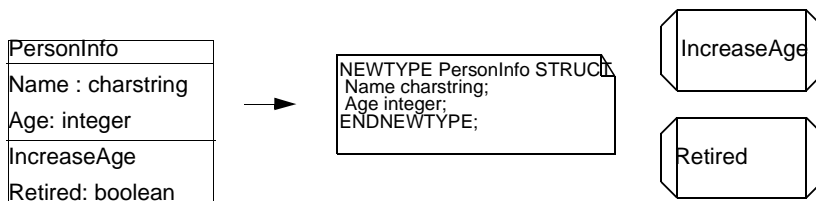


Figure 673: Data type with procedure implementation of operations

Some differences when procedures are used compared to when operators are used:

- Procedures can use both IN parameters, IN/OUT parameters and a return value.
- There is no syntactic relationship between the procedure and the data type definition.
- Procedures can be used in two different ways:
  - using special procedure call symbols (like in [Figure 674](#)).

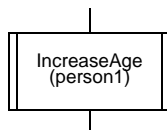


Figure 674: Procedure call symbol

- in expressions with a syntax as the following example:  
`ret:= CALL retired(p1)`

## External Procedures

The fourth way is to implement the operations as normal C functions. These functions correspond in SDL to *external procedures*. For example, to implement the `IncreaseAge` operation, one could have the following C function (the `PersonInfo` data type would have to be specified as a C type):

```
void IncreaseAge (PersonInfo *p)
{
    (p->age)++;
}
```

External procedures can be declared in text symbols, and they are called as if they were normal SDL procedures, as shown in [Figure 675](#). “[Mapping Passive Objects to C](#)” on page 3795 gives more details about mapping operations to C.

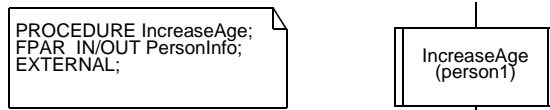


Figure 675: External procedures in SDL

### Mapping Aggregations

A common situation is that the information used by a system has to be structured into some kind of tree structure. In the analysis this will appear as an aggregation hierarchy of passive objects. An example is shown in [Figure 676](#).

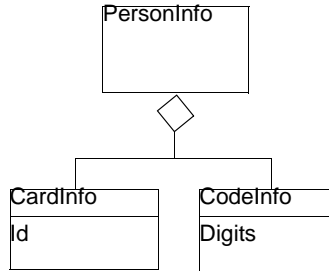


Figure 676: An aggregation of passive objects

If the `PersonInfo` object is mapped to a struct the aggregation would be visible in the design model as fields in the struct the same way attributes would be mapped. The mapping of the `PersonInfo` object from [Figure 676](#) into an SDL diagram is shown in [Figure 677](#).

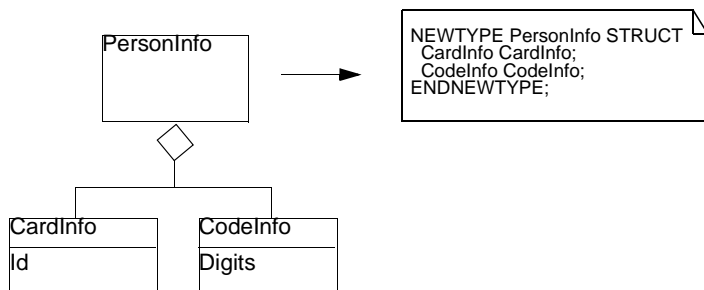


Figure 677: The SDL mapping of a passive object with aggregation

A common special case is when there is a multiplicity associated with the aggregation as in where one person can have more than one card.

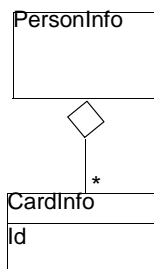


Figure 678: An aggregation with associated multiplicity

In general the multiplicity implies that there is a list or set of elements associated with the aggregation. There are several mappings to SDL possible, for example based on:

- Standard SDL generators like:
  - *Array*
  - *String*
  - Other data types as described in chapter 2, *Data Types*
- A user-defined C implementation of lists

The default mapping in SOMT is a mapping to a type called “xxxList”, where “xxx” is the name of the class. In [Figure 679](#) the mapping of the PersonInfo type is illustrated.

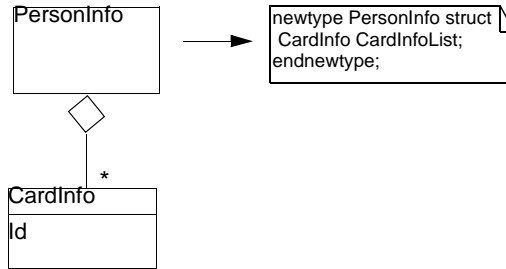


Figure 679: An SDL mapping an aggregation with multiplicity other than one

The “xxxList” type will then have to be designed separately. One simple way is to use an SDL *string* as in [Figure 680](#) for the CardInfoList type.

```

newtype CardInfoList
  string( CardInfo, EmptyCardInfoL )
endnewtype;

```

Figure 680: An SDL implementation of a list using the string generator

Another special case is given by recursive data structures that are used to describe tree structures. In [Figure 681](#) a simple recursive tree is illustrated. In general this type of tree can of course also include a multiplicity other than one.

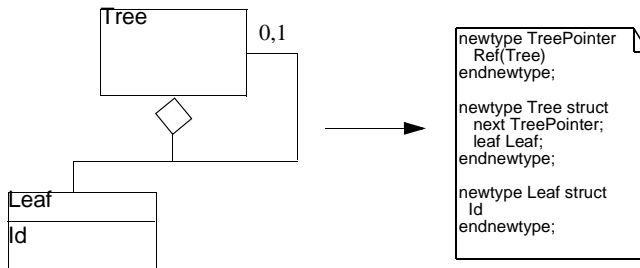
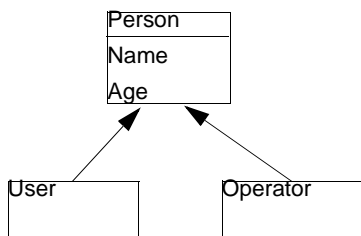


Figure 681: A recursive object model

Due to the lack of pointers in SDL, this type of structure can normally not be defined in standard SDL-92. However, SDL tools may offer tool-specific pointer generators that can be used to represent the above tree. [Figure 681](#) shows how the recursive data structure can be represented with help of the `Ref` pointer generator of the SDL suite.

### Inheritance

Inheritance in an object model is used to model “is-a” relationships. In practise the inheritance shows how attributes, operations and associations are inherited from a superclass to the subclasses. An example is shown in [Figure 682](#) which models the fact that both the users and operators are persons.



*Figure 682: Inheritance relations*

When mapping classes that inherit other classes to SDL data types there are three mechanisms that can be used:

- Flattening
- Delegation
- The SDL inheritance concept

Flattening means essentially that all operators, attributes and associations are copied from the superclass to the subclasses. In the example above this strategy would imply that the SDL representation of the `User` class might look like in [Figure 683](#).

```
newtype User struct
  Name charstring;
  Age integer;
endnewtype;
```

*Figure 683: Representing inheritance using flattening*

Using delegation to represent inheritance is essentially to replace the inheritance hierarchy with aggregation hierarchies. When using this strategy the SDL representation of the classes in [Figure 682](#) will be as illustrated in [Figure 684](#).

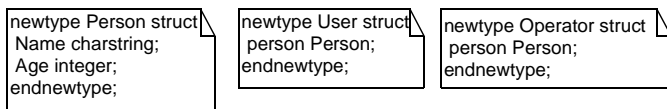


Figure 684: Using delegation to represent inheritance

When using the data types defined as in [Figure 684](#), note that the syntax for accessing the attributes will of course show the delegation strategy used. For example, in order to access the name attribute of a User a construction like “`uservar!person!name`” will have to be used.

SDL includes an inheritance concept for inheritance between data types. Unfortunately the inheritance between data types is an inheritance of operators only which limits the usefulness of the SDL inheritance. For more information about inheritance between SDL data types see “[Inherits](#)” on page 70 in chapter 2, *Data Types*.

Multiple inheritance implies that attributes, operations and association are inherited from more than one superclass as illustrated in [Figure 685](#), that models the fact that a user is both a person and a card holder.

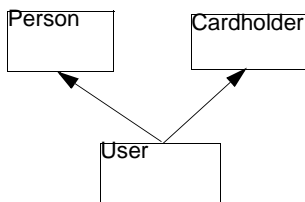
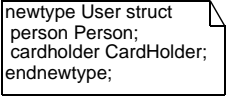


Figure 685: Multiple inheritance example

When mapping classes with multiple inheritance to SDL both the flattening and delegation strategy works fine (except for name clashes when using the flattening strategy) but the SDL inheritance does not include multiple inheritance so it can not be used. In [Figure 686](#) the delegation strategy is used to map the User class from the previous example to SDL.



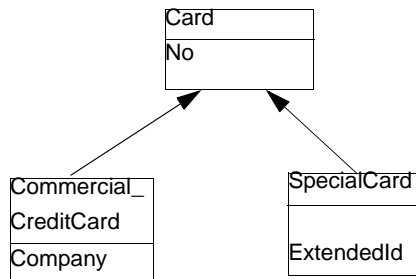
```
newtype User struct
  person Person;
  cardholder CardHolder;
endnewtype;
```

*Figure 686: Mapping multiple inheritance using delegation*

Delegation is the default mapping of inheritance used in SOMT.

## Mapping Passive Objects to Signals

A special kind of passive objects in the analysis object model are the objects that are used only for communication, either between the system and its environment or between modules within the system. For example, when defining the use cases it is useful to show the relations among the events using an object model. It is especially useful if there are inheritance relations among the events as shown in [Figure 687](#). The corresponding SDL signal definitions are shown in [Figure 688](#).



*Figure 687: Object model describing communication events*

There is often a choice of whether to map a passive object used for communication to a struct data type or to a signal. This is one of the design decisions that has to be taken during the object design.

```

signal Card( No );
signal CommercialCard inherits Card adding (Company);
signal SpecialCard inherits Card adding (ExtendedId);
  
```

*Figure 688: SDL signal definitions corresponding to the object model in [Figure 687](#)*



## Mapping Passive Objects to C

Passive objects can also be mapped to C data types and functions. In C, classes correspond to types, attributes of a class to fields in struct types, and operations to functions.

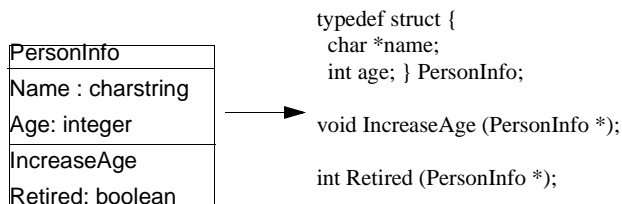


Figure 689 Mapping a passive object to C

Figure 689 shows a possible mapping of class PersonInfo to C. Note that in the mapping of the operations, a parameter is present to identify the PersonInfo object.

To represent aggregations in C, the same mechanisms as described in “Mapping Aggregations” on page 3788 can be used. When mapping inheritance to C, the flattening and delegation mechanisms described in “Inheritance” on page 3791 can be used.

When mapping associations to C, pointers are very powerful. Figure 690 shows an example of a one-to many association that is mapped to a linked list in C. Note how the role name has been used in the mapping.

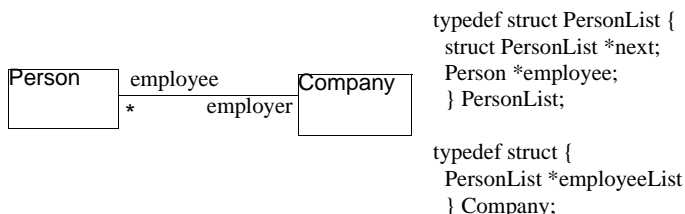


Figure 690 Mapping an association to C

The SDL suite has facilities to instantiate classes that have been mapped to C, and access their attributes and operations from SDL. This is described more detailed in [chapter 2, Data Types](#).

**Note:**

Try to avoid pointers to data in other SDL processes! Data inconsistency will occur if the same data can be read/written by more than one SDL process at the same time. This can be achieved by avoiding pointers in parameters of signals and remote procedures.

Figure 691 shows an SDL fragment that uses the C types for PersonInfo of Figure 689. The CString2CStar operator converts an SDL Char-string to C's char \*. Operations are called by means of SDL procedure calls. Note also the address operator '&' in the call to IncreaseAge.

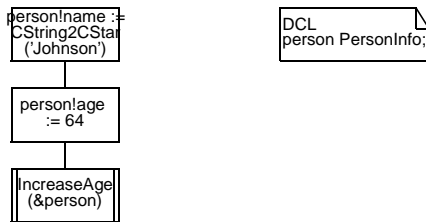


Figure 691 Using C type PersonInfo in SDL

The choice between the SDL struct representation of classes and the C code representation is to some extent a matter of taste. There is a trade-off between simplicity and expressiveness. The SDL struct definition is very simple and all SDL based tools can completely analyze and manipulate this type of data. When using a C implementation, you take over some of the responsibilities from the tools. This means more work for you, but also a possibility to define in detail what you want.

Mapping Passive Objects to ASN.1 Data Types

An analysis object can also be mapped to an ASN.1 data type. The ASN.1 SEQUENCE construct corresponds best to a passive object with attributes. The most basic example is illustrated in [Figure 692](#) below.

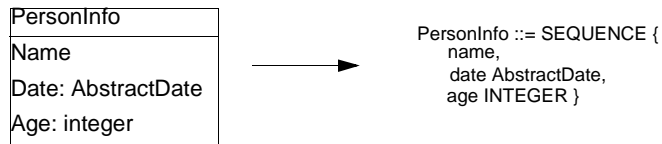


Figure 692: Passive object mapped to ASN.1

The ASN.1 SEQUENCE can be compared with the SDL STRUCT, so most of chapter “[Mapping a Passive Object](#)” on page 3781 on mapping classes to STRUCTs is also valid for ASN.1. The few differences are treated here.

The largest difference is that ASN.1 has no possibility to specify operators. Therefore the operations of a passive object should be inserted in some dummy SDL type, while the attributes are mapped to a different ASN.1 type, as illustrated below.

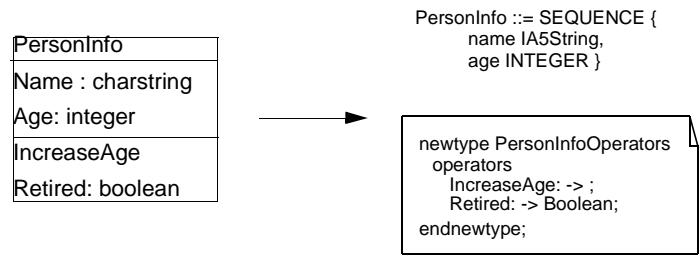


Figure 693: Passive object with operators mapped to ASN.1

The second difference is that ASN.1 has no concepts for inheritance. Therefore only flattening and delegation can be used to represent inheritance. ASN.1 has a special construct, COMPONENTS OF, that is useful to represent flattening. An example of this is given in [Figure 694](#) below.

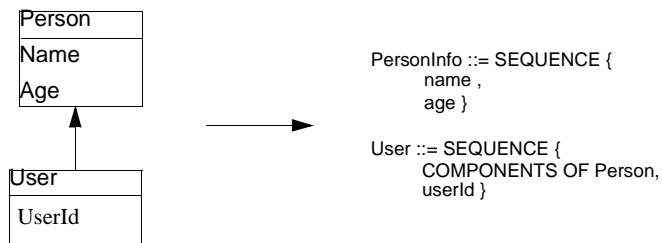


Figure 694: The use of *COMPONENTS OF* to represent inheritance

The SDL suite only allows use of ASN.1 in separate ASN.1 modules, that have to be included into SDL with the **IMPORTS** construct.

## Mapping Associations

Associations are in object models used to represent relationships between objects of different classes. The instances of associations are called *links*. An example is given in [Figure 695](#). In this example the association is a relationship between Cards and Codes that describes that a card must have exactly one valid code. In the object model this is represented by an association called *Valid* between the classes *Card* and *Code*.

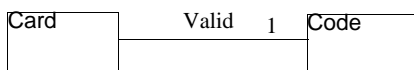


Figure 695: An example of an object model with associations

In the mapping to SDL, the information about the association is kept in one or both of the involved classes, in the example in *Card*, in *Code*, or in both.

There are several aspects of associations that are important to consider when mapping an association to SDL:

- Whether involved objects are active or passive
- The structure of the associations: graph vs. tree
- Intrusive vs. non-intrusive representation
- The traversal direction, one-way vs. two-way
- The multiplicity of the association
- Role names vs. association names

- Association attributes/association class

The rest of this section will discuss these issues and how they influence the mapping to SDL.

In general, associations form a graph structure among the objects where each object may have associations to a number of the other objects. Mapping associations to SDL implies in most cases that some kind of object references or pointers must be used.

In general, associations between active objects indicates that there is a possibility for the objects to communicate. In SDL this means that there must exist a communication path (signal routes and channels) between the corresponding SDL concepts (usually processes). In addition, an association between two active objects may also need to be represented by a Pid variable in one (or both) of the processes.

There are however cases when PIDs should not be used: as Pid is a concept that only exists in the SDL world, associations with objects outside the SDL system cannot be represented with PIDs. Instead, data types defined in protocols define how to refer to external objects. For example in the TCP/IP protocol, TCP services are addressed by a host address (for example 3.1.29.1) and a local port number. In an SDL implementation of a TCP service it would be impossible to refer to a service outside the SDL system by means of a Pid; the host number and the port number must be used instead.

Since strict SDL does not include a pointer concept, associations with passive objects must be represented by other data types. Alternatively a C representation can be used when mapping the objects to SDL as discussed in [“Mapping Objects to SDL Structs” on page 3782](#).

There is one special case where there is no need for special data types. This is the case when the associations form a pure tree structure and furthermore all associations are one-way one-to-one associations with a traversal direction from the root to the leaves. In this case a strategy based on mapping classes to SDL structs can be used, i.e. a reference to an object is in SDL represented by the object itself.

Sometimes an association is implicitly represented in the class definitions themselves, which is called an intrusive representation. In this case the association does not have a representation of its own in the SDL model. For example, if the classes are represented by SDL structs then the association can be represented by a field in one (or both) of the

structs. As an example the Valid association from [Figure 695](#) is mapped in [Figure 696](#) to a field in the struct that represents the Card.

```
newtype Card struct
  Valid Code;
endnewtype;
```

*Figure 696: Mapping an association to a field in a struct*

A non-intrusive representation of the association, on the other hand, does not rely on fields in structs that represent classes or any similar strategy. Instead the association is explicitly represented in SDL. A convenient way to accomplish this is to use the SDL array generator as exemplified in [Figure 697](#).

```
newtype Valid array( Card, Code ) endnewtype;
```

*Figure 697: A non-intrusive mapping of an association using an SDL array*

The design choice to make when choosing a mapping to SDL depends on the way it will be traversed in the application: is the traversal of the association always in one direction or is it traversed in both directions. This is usually not relevant in the analysis model but influences the SDL representation. For example if the Valid association in [Figure 695](#) is a one-way association only traversed from the Card to the Code then the SDL representation in [Figure 696](#) is all that is needed. If, on the other hand, the association is also traversed from the Code to the Card then the Code representation will also have to contain an element corresponding to the association (note that this requires a C implementation due to the lack of pointers in SDL) or alternatively a non-intrusive representation can be used.

The multiplicity of the association defines how many instances of one class can be associated with an instance of the other class. A one-to-many association requires a list or set representation. This is essentially the same as an aggregation with a multiplicity greater than one, which is described in [“Mapping Aggregations” on page 3788](#).

Role names are an alternative that can be used instead of or in combination with association names. A role is one end of an association, and the role name uniquely identifies the object from the perspective of the ob-

ject at the other end. An example is given in [Figure 698](#) where the association is identified using role names instead of an association name.

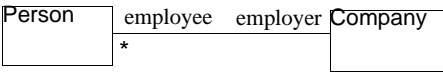


Figure 698: An association that uses role names instead of an association name

In many cases the role names are more convenient and less confusing than an association name. In a mapping to SDL it is preferred to use the role name as e.g. the name of a field in a struct rather than the association name. As an example consider [Figure 699](#) where the Person object from [Figure 698](#) is mapped to a struct in SDL and the role name is used as the field name.

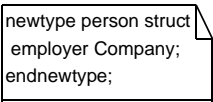


Figure 699: SDL mapping of a an association using the role name instead of the association name

An association may have an associated class as illustrated in [Figure 700](#).

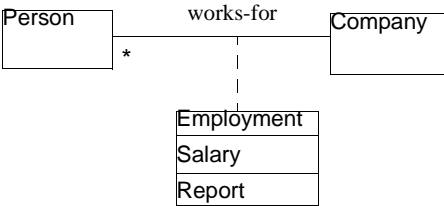
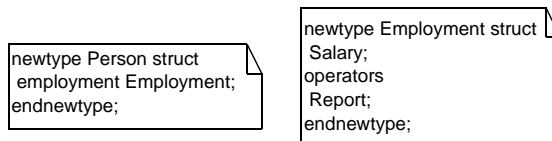


Figure 700: An example of an association class

The implication of this is that there are some attribute values and/or operations associated with each link of this association class that exists between the two objects. When mapping this construct to SDL there are two strategies: an intrusive representation and an explicit representation of the association class. If an intrusive strategy is used there is no explicit representation of the association so the attributes/operations must

instead be associated to one of the objects involved in the relation. This strategy is possible only if at least one of the end points involved has the multiplicity “1”. In the example in [Figure 700](#) it would be possible to incorporate the association class into the Person object since there is only one company for each person in this model. A structured way to represent this in SDL is illustrated in [Figure 701](#).



*Figure 701: The association class from [Figure 700](#) mapped into the Person object with an intrusive strategy*

The second alternative is essentially to view the association class as a regular class and map it to SDL using any of the strategies that can be used to map regular classes to SDL. This is necessary for many-to-many associations.

The default mapping of associations between passive objects in SOMT is to use the intrusive strategy where classes are mapped to SDL structs and thus associations are mapped to fields in these structs. The name of the field in the struct is the corresponding role name if a role name exists, otherwise it is the association name. If there is no name specified for one of the roles and furthermore no name for the association, then no field is generated in the corresponding struct, and the association is considered to be a one-way association.

## Summary of Mappings from Object Models to SDL

There are a number of possible SDL target concepts that an analysis object can be mapped to. The choice of target depends mainly on properties of the object:

- Active objects are mapped to processes or process types
- Aggregations of active objects are mapped to blocks or block types
- Passive objects are mapped to struct data types (or to signals if they are used for communication only)



- Associations between passive objects are mapped to fields in struct data types, or to new data types that explicitly represent the association
- Associations between active objects are mapped to communication paths (and possibly to variables within processes)

## Describing Object Behavior

Once the type of SDL target concept has been chosen, the behavior of the object can be defined. Processes and process types are defined by creating the process graphs and ADTs are preferably defined by giving operator diagrams for the operators.

In practise, the major task of the object design activity is the definition of the behavior of SDL processes since they are the SDL representation of active objects that tend to have a more complex behavior than the passive objects. SDL process graphs provide a graphical notation for extended finite state machines, i.e. finite state machines with variables. In [Figure 702](#) an example of a small process graph that illustrates some of the constructs possible in an SDL process is shown. More details about SDL and SDL process graphs are provided in section “[SDL](#)” on [page 3685](#).

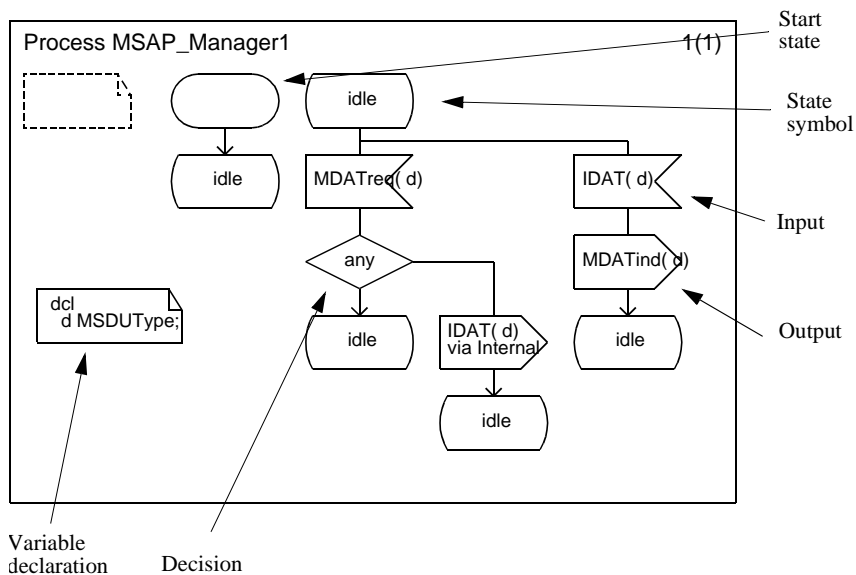


Figure 702: An SDL process graph

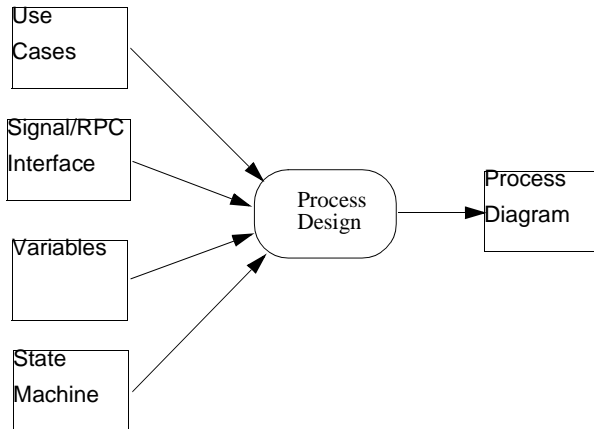
The possible inputs to the process design task are illustrated in [Figure 703](#). The possible inputs directly from the system analysis/design are:

- MSC use cases from the design use case model or analysis use case model
- Possibly a state machine, described by a state chart, giving an overview of the behavior if this was defined in the system analysis

The inputs resulting from the mapping of object models to SDL are:

- The signal and/or remote procedure call interface of the process
- Some process variables defined by the attributes of the object model class

Based on these inputs the goal of the process design is to create a process diagram that defines the behavior of the process.



*Figure 703: Inputs and output of the process design task*

A useful way to structure the tasks that are needed to create a complete process are:

1. Create a first version of the process that defines the control structure in terms of states and transitions. There are two subtasks that in practise are performed more or less in parallel:
  - Defining the control structure
  - Defining data aspects.
2. Elaborate the process by e.g. considering secondary use cases and exceptional cases.

### The First Version – Defining the Control Structure

The control structure of an SDL process is defined by the states and the transitions of the process. Ideally the states of a process represent what the environment might expect of the process. Different states represent different stable phases in the life-time of a process and depending on what state a process is in it will respond differently to requests and inputs from the environment.

One straight-forward way to find the states of a process is to analyze the use cases. Since the use cases show an external view of the processes

they reflect the expectations the environment can have on the behavior of the process. Both directly in terms of sequences of inputs and outputs the process has to conform to but also indirectly in terms of states since each input of a process must be preceded by a state.

So, the task of designing the control structure of the process starts with selecting a set of typical and essential use cases. If the use cases are in MSC format and on an appropriate abstraction level that includes the process to be defined, the use cases can be analyzed directly. If not they should be rewritten to clearly show the responsibilities of the process in question.

Analyze each use case in terms of states and transitions for the process. Incrementally build up the process graph by adding states and transitions. Start in the beginning of the use case and figure out what state the process must be in. Manually walk through the use case, checking the process defined so far and incrementally adding states and transitions to the process graph. Check for each transition in the use case (i.e. input followed by one or more outputs by the process in question) that the transition exists in the process graph. If it does not exist, add it. Take a look in the use case to see if there is an external need for a state change in the process: has the expectations on the process changed after this transition in the use case? If no: go back to the same state again. If yes: is there an already defined state that may fit these expectations? If there is one, use it. If there is no such state, create a new one and give it a name that describes the expectations. Continue until all use cases has been analyzed.

When adding transitions do not forget to check with the mapping from the object model if it is a regular transition with input and outputs or a remote procedure that is to be added. Also check if state lists (and “\*” states) can be used as the starting state of the transition.

During the design of the process also consider what part of the control to put in the process graph states and what to put in variable values. In general it is recommended to define the control flow using the process graph states, but there are cases when it is better to put parts of the control in data values instead of as explicit process states. One example is loop variables that count the number of occurrences of something and is used to exit the loop after a certain number.

One problem that might occur during the use case analysis is that two of the use cases seem to be very difficult to combine in the same process

since they require different states of the process. This is an indication that some restructuring is needed and that maybe the process should be divided into different processes or into a set of services.

When finished with the analysis of the selected use cases the result should be a skeleton process graph that contains states and transitions with mainly inputs, outputs, remote procedures, timer actions and a few tasks and decisions that deal with control variables. Make sure that the state/transition structure makes sense. The states should represent external expectations on the behavior and their names should reflect it. This is an important issue, in particular for the possibilities to maintain the process.

The next step is now to consider the data aspects of the transitions.

### The First Version – Data Aspects

Often there will be three kinds of variables in an SDL process: temporary variables used to handle the parameters of signals, control variables like loop counters as discussed above, and “real” variables that store information about some entity that will be accessed later during the execution. Most of the “real” variables should have been identified during the analysis are given by the mapping from the object model to SDL. The task now is to define how the “real” variables are affected by the transitions. Add temporary variables handling the parameters of the signals when needed and tasks with expressions that define the needed computations. If complex computations are needed it is good practise to hide them in procedures or operators. The transitions should preferably stay fairly simple.

The first version of the process is considered to be finished when it is possible to verify that the process fulfills the selected use cases, e.g. by running a simulator or verifying MSC use cases (compare with “Design Testing” on page 3810). Both the control and the data aspects should be dealt with.

Now it is time to start with the elaboration of the process.

### Elaboration of the Process

The purpose of the first version of the process was to define the control structure of the process and make sure that this is able to cope with the requirements from the most typical and important use cases. The pur-

pose of the elaboration is to complete and refine this structure to make the process definition reliable and facilitate the maintenance. There are several aspects to cover in this elaboration:

- Secondary use case and exceptional cases in primary use cases
- Simplification e.g. using state lists and procedures
- Robustness and completeness of the process
- Restructuring for inheritance and reuse

The major topic for the elaboration task is to consider the secondary use case that was not treated in the first version and also the exceptional cases of the already treated use cases. This is done essentially the same way as when creating the first version as described in “The First Version – Defining the Control Structure” on page 3805. The use cases are walked through by hand and the process graph is checked and possibly extended to cope with the new cases.

To enable the understanding of the process and thus also to make it possible to maintain it, it is important that the definition is as simple as possible. This is a topic that is dealt with in the elaboration task. Procedures can be used to simplify process definitions considerably by defining a particular piece of code in one place and using it in several. Procedures can also be used on a higher level to indicate different phases in the lifetime of the process. Using state lists and “\*” states it is also possible to simplify the definition of a process by defining transitions that are common to many states in one place.

The robustness and completeness of the process must also be handled in the elaboration. The strategy is essentially to make sure that the appropriate action is taken by the process, not only for the expected cases but also for unexpected cases. So, for each state in the process and each input signal/ remote procedure call possible; check that the action taken makes sense. Also check the treatment of unexpected parameter values.

Another topic to be treated in the elaboration is to consider how to facilitate reuse of the created process. Is it possible to create a more general process type by factoring out some parts of the definition and defining a more general process type that can be specialized in other situations?

The elaboration is in practise an iterative process where all the aspects above are treated more or less in parallel. When the elaboration is fin-

ished, the process definition is completed and is ready for integration test. The module test should preferably already have been done at this stage.

### Operator Diagrams

When defining the behavior of passive data types defined in SDL the preferred way to define the operator is using operator diagrams. An operator diagram is essentially a flow graph with a start symbol, symbols defining the actions performed by the operation and one or more return symbols. The symbols may for example be tasks with assignments or decisions. An example is given in [Figure 704](#) that shows the operator diagram for an operator *BirthDay* that increases the age of a person by 1.

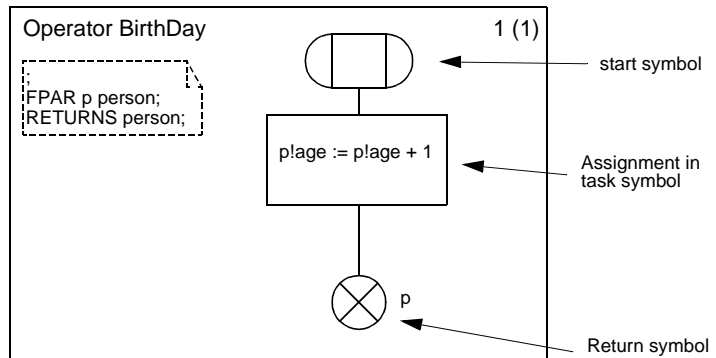


Figure 704: An operator diagram

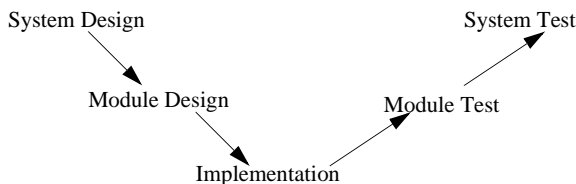
## Design Testing

One of the major benefits with a design notation like SDL that has a well-defined and complete semantics, is the possibility to test the application already in the design activity. This is feasible since the completeness of an SDL design makes it possible to simulate the design taking distribution and concurrence into account.

It is important to emphasize that the output of the object design activity is not only an SDL design but it is a *tested* SDL design that has been shown to fulfil its requirements. This implies that the design testing is an important task in the object design activity.

### Testing Strategy

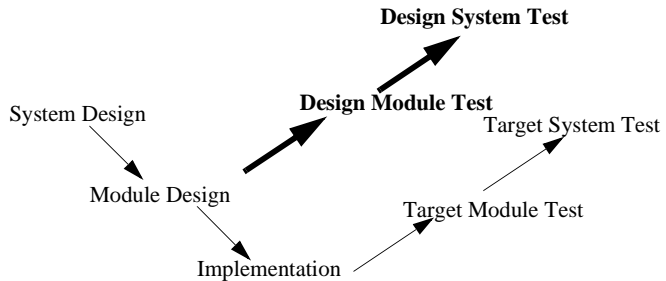
A traditional development/test strategy can be described by a “V” as in [Figure 705](#). The design is performed top down, starting with a system design where the major components and their interfaces are defined, followed by a module design and an implementation phase where the application is implemented. The implementation of each module is then tested separately in a module test and finally the entire system is tested. The system test can also sometimes be divided into two parts, one focusing on the integration of the different modules and the other focusing on testing the functionality of the complete system.



*Figure 705: The “V” model of a traditional design/test strategy*

This model works in practise fairly good but it has one problem: the complete functionality is not tested until the system tests are performed in the end of the development/testing process. To some extent this problem can be overcome by an iterative process that includes more than one “V” in a development project. Using techniques like SDL this can be even more improved by introducing one more line of testing in the model as in [Figure 706](#).





*Figure 706: The improved “V” design/test strategy*

The extra line of testing represents the design level testing that is the focus of this section. There are two practical aspects that differentiates the design testing from the traditional target testing:

- It is performed using a simulation of the design instead of on the real application in its target environment:
- It is performed as part of the design activity instead of as a separate testing activity performed after the implementation.

The second aspect is very important. Problems that otherwise would not show up until system test time is now found and solved during design at a much lower cost.

In the figure it may look like the testing effort has doubled in the improved “V” strategy since there are now two lines of testing instead of one. This is fortunately not the case. What has happened is that the effort of testing has switched from the target testing to the design testing, making the total amount of testing more or less the same. There are several reasons for this:

- The cost of testing is smaller for design testing than for target testing: the design testing on SDL level is very easy to perform. Since a large portion of the testing is performed at the SDL level this reduces the total testing cost.
- The target testing can focus on the integration and targeting aspects since the logic of the design has already been tested in the design tests. In practice the target module testing can even be skipped if automatic code generation is used when producing the implementation.

## Test Case Sources

There are several sources from where the test cases may come:

- From Design Use Cases
- Internally developed during design
- From external sources.

The most important input, both to module and system test, is the design use case model from the system design. This model should capture most of the requirements on the system and using the implinks it is also possible to trace the dependencies from the original requirements all the way to the design use cases. Furthermore, the design use case should be in a format that is possible to test more or less automatically against the SDL system, like MSC or TTCN.

However, during the object design there are usually more test cases developed that tests other aspects of the design, and these form also an important part of the module testing tests.

The third source of tests is external sources. For example, in the telecommunication area it is common to have standardized tests suited for certain types of applications or interfaces. In other cases the customer may have specified acceptance tests that the system must comply to before it is approved. These type of tests should of course also be part of the design tests, in particular for system testing the external sources are important.

It is convenient, but not necessary, if the same notation is used for the design use cases, the tests developed during object design and the external test suites.

## Tools for Testing

To perform the testing there is a need for tools, and fortunately there are several tools available that make a number of testing strategies possible:

- Manual or batch simulation of an SDL system using a standard SDL simulator
- MSC verification and automatic testing using a state space exploration tool
- Co-simulation of SDL and TTCN, using standard SDL and TTCN simulators

Using a standard SDL simulation tool it is of course possible to manually simulate the test cases and check that the system performs as expected, but it requires quite a lot of manual work. A better way is to produce test scripts that contain the simulator commands that are needed and then execute them automatically in a batch mode and log the results on a log file. To check the outcome of the test either the log files are manually inspected or checked by a post processing tool that e.g. compares the new log files with old, manually inspected log files.

Another approach to testing is to use a state space exploration tool that can automatically check if an MSC is consistent with the SDL system. The benefit with this method is that MSCs can be directly input to the tool and checked and furthermore there is no need for a manual inspection of the results: a verdict can be automatically generated by the tool. The drawback is that some features like the combination of user-written and automatically generated code can not always be handled by state space exploration tools.

A third approach that is useful if the tests are defined using TTCN is to use a co-simulation of an SDL system and a TTCN test suite as the means to perform design testing. Essentially this is similar to using an SDL simulator alone, but instead of specifying the input as simulator commands a TTCN simulator specifies the input to the SDL simulator and checks the outcome of the test.

## Test Practices

A common situation in particular when performing module tests is that there is a need to test one part of an SDL system in isolation from the rest of the system. In an SDL system the part may be for example a block, a process or a data type definition. The simplest way to accomplish this in SDL is to use a package as a container of e.g. the block (which in this case will have be a block type) or data type and then use a special test system to specify the test environment.

As an example consider the DoorCtrl part of the access control system. Assume that this part is designed as a block type “DoorCtrlT” in a package DoorCtrlPack. To perform a module test on this block the simplest way is to create a special test system DoorCtrlTest that instantiates the DoorCtrlT and connects all its gates to the environment.

## Consistency Checks

This sections describes some consistency checks that are useful to perform on the models produced in the object design.

- Check that all objects from the analysis object model has been implemented in the design.
- Check that the design model is complete according to the SDL rules, e.g. that all processes have a defined behavior.
- Check that the design model correctly implements the requirements from the design use cases using design level testing.

## Summary

The object design activity should produce a complete and tested design of the system. The precise system and internal object structure as well as the reuse structure are defined in this activity. Relevant parts of the object models are mapped to SDL concept and then completed in an SDL process design activity with design use cases as the main input. This is done iteratively by starting with a initial set of essential use cases, making that part of the design complete and then testing the design (verifying it against the use cases). The activity is finished when all use cases have been implemented and the final design has been tested against all the use cases.