

System Design

This chapter gives a thorough description of the different models in the system design activity and some guidelines on how to create these models. A recommendation on consistency rules that are relevant for the models in this activity as well as for the consistency between models from previous activities and this activity, is also included.

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

System Design Overview

One of the most important issues of software development, if not the most important of all, is to define the architecture of the system. Define how the system is built up of smaller parts that in turn may be composed of even smaller parts until each part is manageable by itself. The architecture is proposed in the system analysis architecture but the precise definition of this structure is the major task of the system design activity.

The components of a system may have several different important functions to fulfill:

- They act as a unit for work division. Different development teams can be responsible for different components.
- They form a decomposition of the functionality. Each component may be responsible for one aspect of the total functionality of the system.
- They act as distribution units. The components can define how the system is distributed in the physical world.
- They may act as technology units. The design of the different components may use different notations and tools and, although SOMT has its main focus on SDL, the system design activity also takes other possibilities into account.

The major inputs to the system design activity are the analysis object model and the analysis use case model produced in the system analysis activity. The system design is the process that based on these inputs define in detail how the system is decomposed into components and to define the interfaces between the different parts. This is illustrated in [Figure 646](#) that also shows the three major artifacts developed in the system design; the design module structure, the architecture definition and the design use case model. In addition to these formalized descriptions there is often a need to specify non-functional aspects of the components in a textual design documentation.

System Design Overview

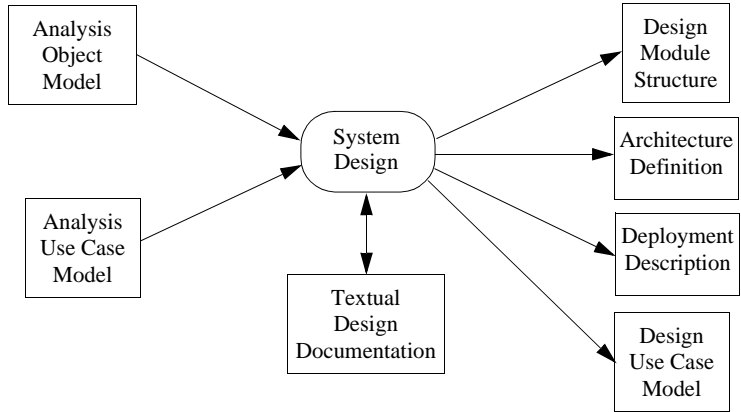


Figure 646: Inputs and outputs of the System Design activity

The architecture is in the system design formalized primarily using SDL. In SDL the major structuring concept is the *block* and the notation is the block diagram. Seen from an object-oriented point of view a block is a container of objects. The block can either be directly described by the objects that it contains or it is decomposed into lower level blocks. The block structuring mechanism is discussed more in [“Architecture Definition” on page 3755](#).

Where the logical architecture defines the decomposition of logical functionality the *design module structure* defines the decomposition into work items. It defines the different modules the design teams can start working on and also provides a mapping from the logical architecture to design modules. In SDL the design modules are usually SDL *packages*. The design module structure also takes a slightly broader perspective of the system to be built and describes how existing frameworks, tools and components are incorporated into the development structure.

The *deployment description* is a way to describe the physical distribution structure of the SDL system. It is also the place where the implementation strategy for different parts of the system can be described.

There should be a simple (if possible one-to-one) mapping between the top levels of the architecture definition and some of the modules in the design module structure. The benefit gained from a simple mapping is

that the design modules define the possibilities to divide the work on different development teams and the logical blocks comprise well-defined sets of functionality and responsibilities. If they do not map to each other there is an obvious risk for complex interfaces between the development teams.

As always when a system is decomposed into smaller parts one very important issue is how the interface between the parts are defined. In particular if the components are used as division of work load and designed by different development teams the interface definition is the means to communicate between the different groups and a common understanding of the interface is crucial. There are two aspects of the interface:

- A static aspect, defining the operations or services that a block offers
- A dynamic aspect, that defines how the different blocks cooperate to solve a common task

Both aspects are important and the definition of them is a vital part of the system design activity.

In SOMT the major concepts used to define the static interface are the SDL concepts *signals* and *remote procedures*, and the dynamic aspect is a continued usage of use cases. However, since there very often is a need to design parts of a system using other techniques than SDL or to use existing modules, other interface definitions techniques are also used in SOMT.

The major tasks to be performed in system design in an SDL based project can thus essentially be summarized as the following:

1. Create an (incomplete) SDL system that is a starting point for the formalization of the architecture of the application. This is further discussed in [“Architecture Definition” on page 3755](#).
2. Define the design module structure. Draw a diagram that illustrates the structure and create the necessary packages etc. as described in [“Design Module Structure” on page 3757](#).
3. Define the physical distribution strategy for the SDL system, see [“Deployment Description” on page 3760](#).
4. Define the static interfaces as discussed further in [“Static Interface Definitions Using SDL” on page 3761](#).
5. Define the dynamic aspects of the interfaces by a continued use of use cases. See section [“Design Use Case Model” on page 3764](#).

There is, as we will see in [“Object Design” on page 3771](#), a close relation between the system design activity and the object design activity in the sense that the object design activity is concerned with the representation and behavior of the objects and the system design deals with the distribution of the objects into blocks and defining the communication paths between the objects.

The rest of this chapter will discuss the system design activity. SDL will frequently be used to define the architecture and interfaces and examples of SDL diagrams will be used throughout the chapter. A complete presentation of the SDL language is however outside the scope of this document. For more information, please consult either the Z.100 standard itself [\[23\]](#), or a text book about SDL like [\[27\]](#).

Architecture Definition

When using SDL to design a system the architecture of the system is defined by the block diagrams. They define how the system is decomposed into blocks and how these blocks either form the leaves of the block hierarchy or are further decomposed into smaller blocks. Essentially this block structure is a formalization of the logical architecture from the system analysis.

As an example, consider once again the access control system. The system controls the doors of a building to unlock the doors when an authorized user wants to enter or exit the building. The task in system design is to define how to structure this system. One natural choice is a distributed structure where the control of each door is localized close to the door and a central controller keeps all common information about authorized users, cards and codes. Furthermore one special block is responsible for the handling of an operator panel. A logical architecture that described this was illustrated in [Figure 641 on page 3733 in chapter 72, *System Analysis*](#). The SDL diagram that shows a beginning of a formalization of this architecture is depicted in [Figure 647](#).

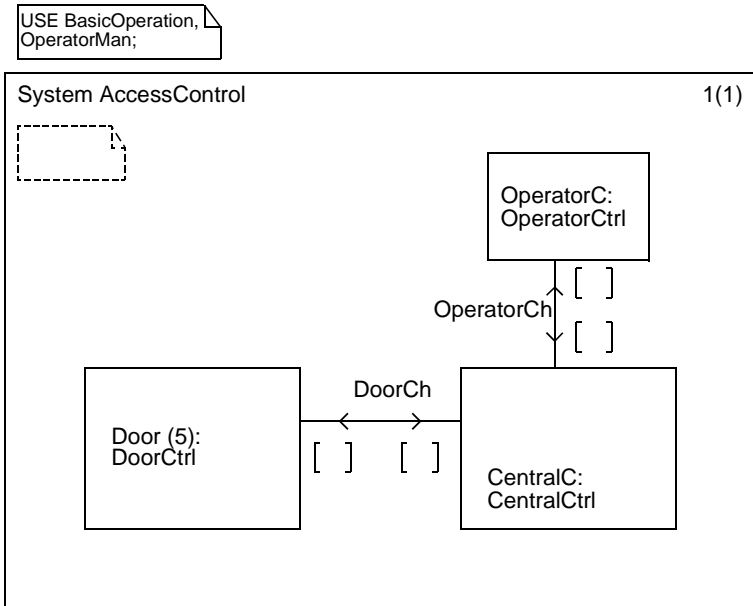


Figure 647: The architecture of the Access Control system defined by an (incomplete) SDL system diagram

In this diagram we can see the blocks *CentralC*, *Door* and *OperatorC* that are instances of the block types *CentralCtrl*, *DoorCtrl* and *OperatorCtrl*. *CentralC* contains the common information base about cards, codes etc. that are registered in the system. *Door* is the block responsible for the control of each door and *OperatorC* handles the operator communication. The diagram also shows how the blocks communicate using the channels *DoorCh* and *OperatorCh*. Note that there are five doors in the building in this case and that this is shown by defining *Door* to be a block instance set. The types *DoorCtrl*, *CentralCtrl* and *OperatorCtrl* are assumed to be defined in the packages *BasicOperation* and *OperatorMan* that are referenced in the *USE* clause in the top of the diagram.

Design Module Structure

The purpose of the design module structure is to show the actual components the application will be built from. The module structure should depict the actual source code modules etc. that the application will contain. A number of different aspects must be taken into account when defining the module structure:

- The implementation strategy for each module: Some modules may be designed in SDL with automatic C code generation. Other modules may be manually designed and implemented in a programming language and yet other modules may need a hardware implementation.
- Existing utility modules that can be reused in the new application
- Existing architectural frameworks that can be reused in the application
- Of-the-shelf utility modules that can be purchased and used in the application

Especially the reuse of existing architectural frameworks is very common and very beneficial. Most applications are not built from scratch, they are rather extensions/modifications of old applications and the design module structure is the place to show how this is done.

One notation that can be used in SOMT to describe the design module structure is object model instance diagrams, where the instances represent the different modules. Where relevant, the attribute field can be used to show what components of the logical architecture are contained in the modules. As an example consider a typical SDL application running on a small microprocessor where a proprietary real-time operating system is used. A possible module structure is shown in [Figure 648](#).

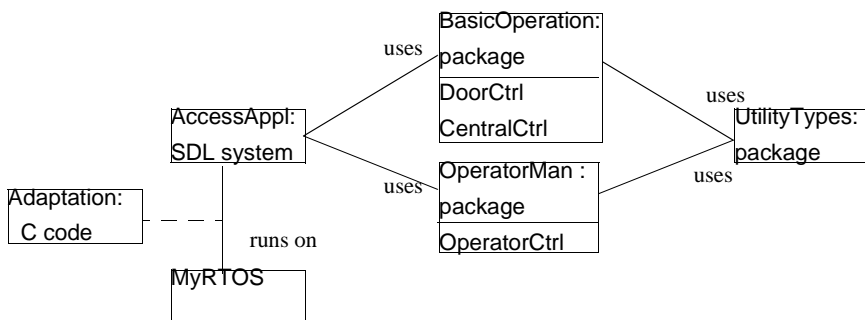


Figure 648: The module structure of the Access Control system using an in-house real time operating system

In this example the top-level of the application is described in the SDL system `AccessAppl` which is defined using the types defined in the packages `BasicOperation` and `OperatorMan`, both of which uses common types defined in the `UtilityTypes` package. The application will run on the already existing real time operating system `MyRTOS`. To make the C code generated from the SDL system run on `MyRTOS` a C code module `Adaptation` is used that defines the necessary interfaces.

In this example it is very likely that some of the modules will be developed within the project (the `AccessAppl` and the modules it uses) while others are already existing (like the real time operating system) and some can be taken from previous projects and be modified to fit the current project (like the `Adaptation` module).

The most important aspect of the module structure is that it forms the basis for dividing the work load on different development teams. This is in many cases the major reason to decompose the system into design modules. However, another reason may be the issue of reuse.

The design for reuse can in this context be viewed as an activity that defines the design module structure based on other premises than the architecture as discussed so far in this chapter. Consider for example the access control system decomposed as in [Figure 647 on page 3756](#), where the system is divided into three different parts according to essentially the physical distribution that is needed in the application. There may in this example exist concepts that can be used in more than one of the subsystems. Examples may include knowledge about some passive

data structure, like the concept of a card, but also entire functionalities like the concept of time seen as a clock functionality.

The identification of these type of components is also a task of the system design. It is particularly important if the different subsystems are to be designed by different design teams. It is also important to identify common components to avoid duplicate work and lower the complexity of the individual subsystems.

The concept that is used to describe the different modules is the *package*. A package is essentially a container of SDL types, this may range from system types and block types over process types down to signals and data types. When a package is used by an SDL system the types defined in the package can be referenced from within the SDL system. For example, in the access control system we may decide that we need a package `UtilityTypes` (as in [Figure 648](#)) that defines the common data types needed in the different subsystems.

Another issue that needs to be handled in the system design is to analyze the consequences of the requirements on different configurations of the system. Are there any optional parts or functionality? In many cases the optional parts are captured by different modules in the system structure, but sometimes, like if there would be an optional requirements on a synchronized clock in all parts of the access control system, it is distributed over the different blocks in the system. If this is the case a package is the most useful concept to use to encapsulate the optional feature.

The discussion so far has been about reuse within one development project focused on one specific application. There is however also the issue of reuse outside the local project. When using an object-oriented approach to the analysis and design the objects tend to be fairly general and applicable in more than one project. If the objects in a particular part of the system are defined as types in a package this will form a good foundation for reusing the objects in future projects. This implies that there may be a reason to use a package structure that is different from the block structure of the system. The package structure reflects the decomposition into packages as defined by the possibility for reuse while the block structure defines the current system structure.

Deployment Description

The purpose of the deployment description is to define the physical distribution of the application and also define the practical details on how to build the different parts.

The specific objectives for a deployment description might vary during different activities: during design we are more concerned on describing on how to verify or validate the design, i.e. how to simulate the design in different ways, while in later activities it is desired to specify the final application build process for the application structure. It is therefor possible to have several deployment descriptions for one system.

There is a textual format for defining a deployment description – for more information, see *“Build Scripts” on page 2572 in chapter 57, The Cadvanced/Cbasic SDL to C Compiler, in the User’s Manual.*

Example of a textual deployment description, i.e. build script:

```
set-kernel SCTAAPPLCLENV
set-env-header on
program UserPart
component system AccessControl / block LocalStation
make-template-file UserMake.tpm
generate-micro-c
program CentralPart
component system AccessControl / block CentralUnit
make-template-file CentralMake.tpm
generate-advanced-c
```

Static Interface Definitions Using SDL

SDL offers two major means to define the interfaces of a block:

- Signals (for asynchronous communication)
- Remote procedure (for synchronous communication)

When signals are used to define the interface to a block they define the communication items that can be sent to and from the block. A signal can represent a service to be carried and it contains all relevant data that is associated with the request. A useful way to structure the signals if one particular interface contains many signals is to define signal lists that group together related signals. Consider the CentralCtrl block above. This block has two interfaces, one to the Door blocks and one to the OperatorCtrl block. The interface to the Door blocks can in SDL be defined as in [Figure 649](#).

```
/* CentralCtrl door interface definition */  
signal  
  Validate(Card, Code), /* Check card and code authorization */  
  Accept,               /* Card and code accepted */  
  Reject;               /* Card and code rejected */  
signallist CCSERVICE = Validate;  
signallist CCSERVICEReply = Accept, Reject;
```

Figure 649: An interface definition using signals

When using signals to define the interface of a block we do not put any constraints on the execution strategies in the respective blocks, we only define the data that is transported. However, in some cases, especially when using a client-server based architecture, it is more convenient to define the interface using remote procedures instead of signals. As an example consider once more the CentralCtrl block. The major responsibility of this block is to store the cards with their associated code. Some possible operations on this data is to check whether a particular card is registered and what the code for a particular card is. A remote procedure definition of these operations is depicted in [Figure 650](#).

```
/* CentralCtrl interface definition */  
remote procedure CardRegistered; fpar Card; returns Boolean;  
remote procedure GetCode; fpar Card; returns Code;
```

Figure 650: An interface definition using remote procedures

In addition to the signals/remote procedures that are used to define the interfaces in SDL there is of course also a need to define the data types that are visible in the interface. This issue is to a large extent the same as the issue of mapping passive objects to SDL data types. This is treated in more detail in section [“Mapping a Passive Object” on page 3781](#).

Mapping Object Models to SDL Interface Definitions

When mapping object model concepts to SDL there are two aspects that need to be taken care of:

- The design of the interface
- The design of the object itself

In SOMT this implies that an analysis object is seen to have two different descriptions in the design model, one description of the interface and one description of the object itself. In the system design the focus is on the interface definition so we will save the mapping from object models to SDL object definitions until the next chapter ([“Object Design” on page 3771](#)). However, defining the relation between the object model concepts and the interfaces between the components of the system is a very relevant issue for this section.

Since the basic mechanism in SOMT to go from analysis to design is using the *Paste-As* mechanism (see [“Implinks and the Paste As Concept” on page 3666](#)) this is of course also used when defining the interfaces. As seen in the previous section interface definitions in SDL are defined using signals and/or remote procedure calls. Consequently this is what is produced when mapping a class to an *SDL Interface*.

As an example consider the `DisplayInterface` objects in [Figure 651](#) that has one operation each.

DisplayInterface	DisplayInterface2
Display	Display2{sync}

Figure 651: The DisplayInterface objects

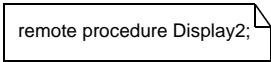
When mapping these objects to SDL interface definitions we get in the first case the signal interface definition in [Figure 652](#) (with the signal *Display*) and in the second case the remote procedure definition in [Figure 653](#) (with a definition of the remote procedure *Display2*).

In SDL the natural way is to express interfaces by asynchronous signals, therefore this mapping has been chosen to be the default. (It can also be explicitly denoted with the word *async* within brackets, after the operation name.) If synchronous interface is preferred, this can be denoted by the word *sync*. This is an extension of the original class diagram notation.



```
signal Display;  
signallist slDisplayInterface = Display;
```

Figure 652: The signal interface from the DisplayInterface object in [Figure 651](#)



```
remote procedure Display2;
```

Figure 653: The remote procedure interface given by the DisplayInterface2 object in [Figure 651](#)

It is of course also possible to have a mixed signal/remote procedure interface. In this case some of the operations are asynchronous and thus mapped to signals while other are synchronous and thus mapped to remote procedure definitions.

Design Use Case Model

The static interface definition alone is not enough to define how the blocks are supposed to cooperate to meet the requirements on the system. In the requirements and system analysis, use cases were used to describe the requirements on the system. This is continued in the system design to define the dynamic interface between the blocks in the system. Essentially the idea is to take each one of the use cases found in the system analysis and formalize this to a sufficient degree of detail that is consistent with the level of detail that is found in the static interface definitions. The degree of detail must be precise enough to make the design use cases act as detailed test specifications.

A benefit with the design use cases is the structured way in which they are constructed. It is easy to verify that all requirements as expressed by the requirements use cases and refined in the analysis use cases are handled by the design use cases and this gives a formal link between the requirements and the structure of the system that implements it.

In a development environment where the different blocks are developed by different teams they also form a necessary common definition of the responsibilities of their respective blocks and how their blocks are to together fulfill the requirements on the system.

From a practical perspective this puts some requirements on the notation used to describe the design use cases:

- It must be precise and formal enough to allow an specification of test cases on a detailed level of abstraction.
- It should be possible to automatically check the design use cases against the SDL design model.
- There must be a well-defined way to transform the design use cases to executable test programs that can be executed in the target environment against the application.

There are two levels of testing of interest for the design use case model:

- Module testing
- System testing

Module testing is intended to test one specific part of the system and should check that this particular part of the system fulfills its requirements. The system testing is intended to test the integration of the different parts and check that they together fulfil the requirements on the total system.

The design use case models should form the basis for both kinds of testing.

Another aspect of testing is when in the development project it is performed. One of the benefits of SDL is that it is possible to test already on the design model, essentially testing against a simulation of the SDL system. In addition there is of course also a need to test the implementation in the target environment, but if the logics of the application already has been tested during design then the focus of the target testing can be on target integration issues and the risk for logical errors in the design is reasonably small. The design level testing is further discussed in [“Design Testing” on page 3810](#).

In SOMT two different alternative notations are used:

- MSC
- TTCN

Usage of MSC

MSCs can be used in a way that is precise and formal enough and can automatically be checked against SDL design models. A benefit is that it is used also in requirements analysis and system analysis and is intuitive and easy to understand also for non-experts. Notice however that there is a difference between the analysis use cases and the design use cases. The fairly abstract messages exchanged between the instances in the analysis use cases must in the design use cases be refined to the level of the static interface definitions, this may include specifying parameter values that were left out and even replacing one message with a sequence of message exchanges. It may also often be necessary to have more than one design use case for each analysis use case, for example to handle a situation where the analysis use case has left out a parameter and there is a need to test more than one combination of parameters.

Consider again the access control system with a decomposition according to [Figure 647](#) where the system is divided into a DoorCtrl, a CentralCtrl and an OperatorCtrl block. If we take the *Enter building* use case as defined on requirements level by an MSC in [Figure 628 on page 3707](#) and refined to the analysis use case in [Figure 635 on page 3722 in chapter 72, System Analysis](#). When further refining this to a design use case we get an MSC as shown in [Figure 654](#) where some of the messages have been refined.

Note that an MSC describes both the requirements on the separate parts of the system and the requirements on the whole system. This implies that the same MSC can be used to define both module and system tests.

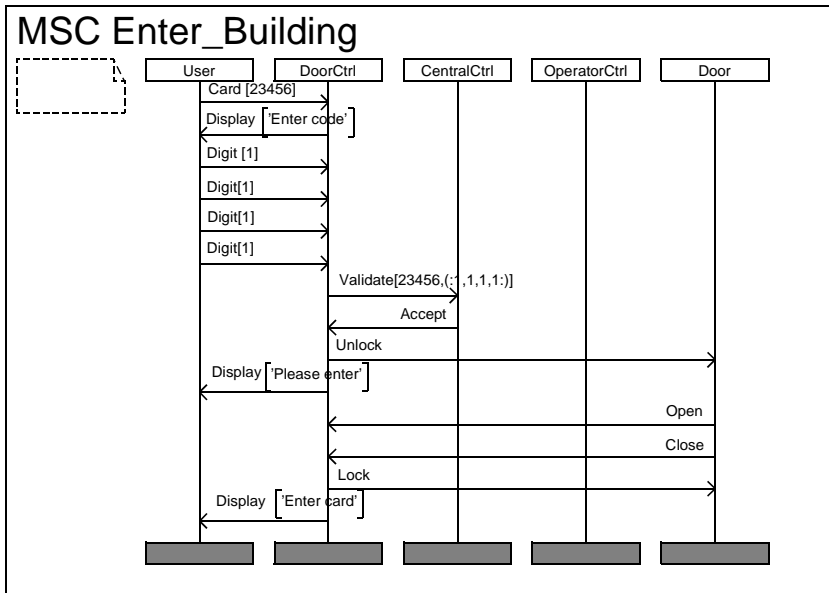


Figure 654: The Enter building use case distributed over an architecture

It is easy to see that the strategy outlined above is extendable to allow a decomposition of a system into not only one level of blocks, but into a hierarchy of blocks. For each new level of decomposition all use cases that involve the decomposed block are taken as input to the validation of the new decomposition. The block that was decomposed is replaced by the new blocks and new versions of the use cases are created.

Usage of TTCN

TTCN is another notation that is suitable for formalizing use cases on the design level. The benefit of TTCN is that it is a special purpose language for test description including:

- Facilities for describing constraints on complex data values
- Preambles and postambles to show how to compose test cases

Design Use Case Model

- Possibilities to handle alternative outcomes of a test case
- A special “verdict” construct to define the outcome of a test case

TTCN is also an established notation for test description so there is good tool support for executing TTCN test cases on target platforms.

The drawback is that it has not a particularly intuitive syntax, making it more difficult for non-experts to maintain, create and review TTCN test suites.

If TTCN is used to define both system and module testing each analysis use case will result in several TTCN test cases, one for each part of the system and then one for the entire system. As an example consider [Example 605](#), that shows a test case testing the requirements from the Enter_Building use case on a DoorCtrl, and [Example 606](#), that shows a test case testing the requirements from the Enter_Building use case on the entire AccessControl system.

Example 605: A TTCN test case testing a DoorCtrl

1	UsrPCO?Card	Card1
2	UsrPCO?Display	Enter_Code
3	UsrPCO!Digit	Digit1
4	UsrPCO!Digit	Digit1
5	UsrPCO!Digit	Digit1
6	UsrPCO!Digit	Digit1
7	CentralPCO!Validate	Validate_1
8	CentralPCO?Accept	AcceptOK
9	DoorPCO!Unlock	Unlock!
10	UsrPCO!Display	Please_Enter
11	DoorPCO?Open	Open1
12	DoorPCO?Close	Close1
13	DoorPCO!Lock	Lock1
14	UserPCO!Display	Enter_Card P

Example 606: A TTCN test case testing the AccessControl system

1	UsrPCO?Card	Card1
2	UsrPCO?Display	Enter_Code
3	UsrPCO!Digit	Digit1
4	UsrPCO!Digit	Digit1
5	UsrPCO!Digit	Digit1
6	UsrPCO!Digit	Digit1
7	DoorPCO!Unlock	Unlock!
8	UsrPCO!Display	Please_Enter
9	DoorPCO?Open	Open1
10	DoorPCO?Close	Close1
11	DoorPCO!Lock	Lock1
12	UserPCO!Display	Enter_Card P

The choice between MSC and TTCN as design use case notation is very much influenced by application and development organization aspects:

- It is more efficient if the same test definitions can be used both for design level testing and target testing, so the plans for how to perform the target level testing may have an implication for the choice of notation. If the target testing should be done using a TTCN environment then at least the system tests should be defined in TTCN also for the design level testing.
- On the other hand, if the target tests are performed using an in-house test script notation implying that the same notation can not be used both in design and target testing, then MSC has the advantage of being a simpler notation and is already known and used in the previous activities.

Textual Design Documentation

The SDL architecture definition and the design use cases form a specification of the static and dynamic aspect of the components from a functional viewpoint. In many cases there is a need to extend this with more information that is not suitable to express in SDL or as use cases. An example may be a system that requires a user interface with windows, menus etc. or a system with specific requirements on reliability or response times for some or all of the components.

To give the possibility to express this type of specifications and also to allow other types of design or project documentation in an environment that is mainly SOMT and SDL oriented, the SOMT method gives a possibility to include textual documents in the system design documentation.

Consistency Checks

This section gives a number of examples of consistency checks that can be made on the models produced in the system design.

- Check that there is a simple (preferably one-to-one) mapping between all the top-level subsystems in the architecture definition and some of the design modules as defined in the design module structure.
- Check that the actual modules (SDL packages etc.) used in the design are consistent with the design module structure.
- Check that the subsystems in the logical architecture in the analysis object model are mapped to the architecture definition in the design.
- Check that all use cases from the requirements analysis and system analysis are refined to design use cases.
- Check that the instances in the design use cases correspond to the blocks/processes in the architecture definition.
- Check that all objects in the analysis object model either has been mapped to some interface definition or really are internal to their module.
- Check that the different models conform to the rules for their respective notation (like SDL and MSC).

Summary

The system design is an activity in which the architecture of the system to be built is defined in SDL. Use cases from the analysis are refined to a granularity that will be sufficient for describing the behavior of the subsystems in the architecture. These use cases should be a source for module and system testing in later activities.

