4 TIMe at a glance

Introduction 2
TIMe from SISU 7
What's in TIMe for the manager 7
What's is TIMe for the designer 8
The Why, What and How of TIMe. 9
Introduction 9
TIMe Essentials 14
System Development Activities 27
Analysis. 28
Design. 48
Implementation 66
Instantiation. 66
Object and Property Models
- and the Languages for describing them 68
Object Modelling 68
Property Modelling 84
List of figures 92

TIMe at a glance

Introduction

The Integrated Method (TIMe) supports design oriented development, an approach to system development where systems are understood and maintained mainly in terms of abstract design descriptions. It even goes one step towards making the vision of property oriented development come true.

TIMe for what?

TIMe is designed for systems that are

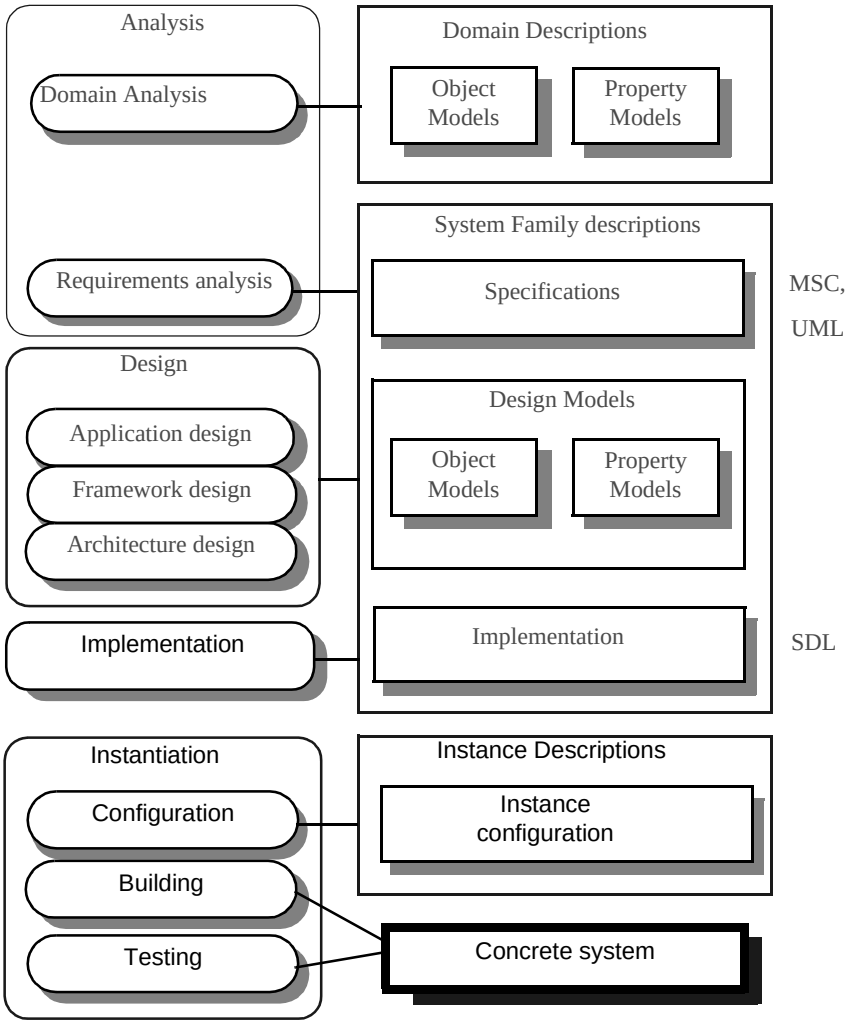
- reactive,
- concurrent,
- real-time,
- distributed,
- heterogeneous and
- complex.

TIMe is centered around a set of models and descriptions capable of expressing domain knowledge, system specifications in terms of external properties, system designs in terms of structure and behaviour, implementation mappings and system instantiation.

Like most other similar methods, TIMe distinguishes between Analysis, Design, Implementation and Instantiation (see Figure 4-1 (p.4-3)).

Figure 4-1: TIMe activities, descriptions and languages

Open figure



How TIMe is different

The distinction between Domain and System Design is not particular for TIMe. What is special, however, is that:

- design is split between
  - *application design*, where the functionality of the system is design,
  - *architecture design*, where the non-functional properties are taken care of, and
  - *framework design*, that defines types of systems with the same infrastructure (e.g. supporting distribution) where the application specific parts are singled out to be redefinable in specific systems.
- the complementary object models and property models are used both for domain and system analysis, and for design.

TIMe provides:

- a set of system development activities that covers most of the system development process, with emphasis on the activities leading to implementation,

- guidelines on object and property modeling in general, and particularly how to do it in UML [147] / SDL [102] - [104], [108] and MSC [105],[110] respectively, and
- tutorials in UML, SDL and MSC.

Object oriented

TIMe is truly object oriented in its approach. It defines its own underlying object and property models, and contains detailed guidelines on:

- how to make analysis object models using Unified Modeling Language (UML),
- how to make design object models using Specification and Description Language (SDL), and
- how to make interaction property models and Use Cases using Message Sequence Charts (MSC).

TIMe is characterised by:

Abstract models

- Emphasis on *abstract* models and descriptions: Abstract descriptions leave out implementation specific details and let the developers focus on functionality.

Property models

- Focus on (external) properties: Objects are the building material from which systems and components are constructed. Property descriptions are used at an early stage of development to express the properties *required* from a system or an object. At a later stage they are used to express the properties actually *provided* by a system or component.

Service orientation

- Users tend to think in terms of services and interfaces. Therefore TIMe recommends use of separate property models for services and interfaces. These models are used for high level service engineering, and for synthesising object designs that provide the services.

Roles

- Strong *object-property relationships*: Roles are used to describe properties, and are related to object designs by projection. Roles are used to link properties and objects. Projections are used for synthesis of new objects and for documenting existing objects.

Design for reuse

- Planned *variability* and *reuse*: TIMe seeks to make generic system families that may be adapted as easily and safely as possible to the needs of particular systems. Components for reuse across families come from general domain descriptions. TIMe describes a cost-effective way to define instantiation of particular systems by defining the general parts by reference to the family description, detailing only what is special for that particular occurrence, i.e. its *configuration*.

Synthesis

- Design synthesis: Property oriented design involves:
  - Decomposing required service and interface properties into object properties.
  - Synthesizing object designs from required object properties, by transformation and by composition, taking reuse into account.
  - Comparing properties: required against provided (validation).

Design with reuse

  - Searching for components with provided properties corresponding to some required properties.

- Composing properties corresponding to object composition.

Goal of this document

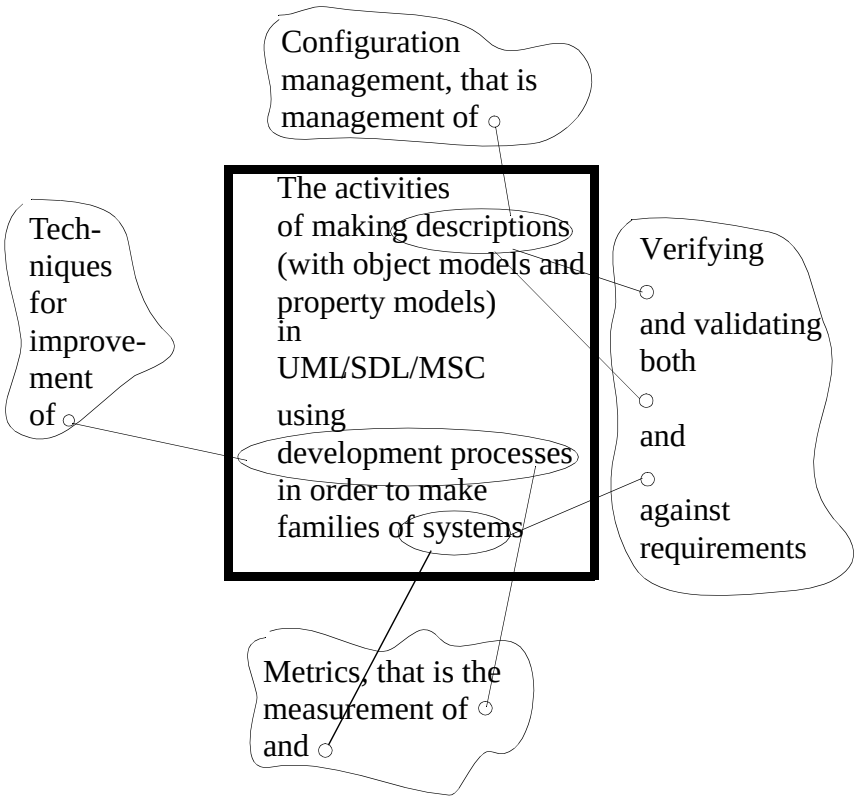
This document provides the TIMe Essentials (p.4-14), intended for readers that would like to know why they should use TIMe, get a feeling for what TIMe is, if it applies to their needs, how it differs from other similar methods, etc.

Focus

The focus in this document is the *core* of TIMe, that is system development *activities* with the combined use of UML, MSC and SDL for making *models*, based on a common approach to *object modeling* and *property modeling*, with emphasis on the early stages of system development. As indicated in Figure 4-2 (p.4-5), TIMe is more than this.

Figure 4-2: The core themes of TIMe covered in this introduction, and supplementing themes

Open figure



This introduction can be read as a stand-alone document, but when read in electronic form, and integrated with the full method book, it also works as an introduction, with hyperlinks to the whole method.

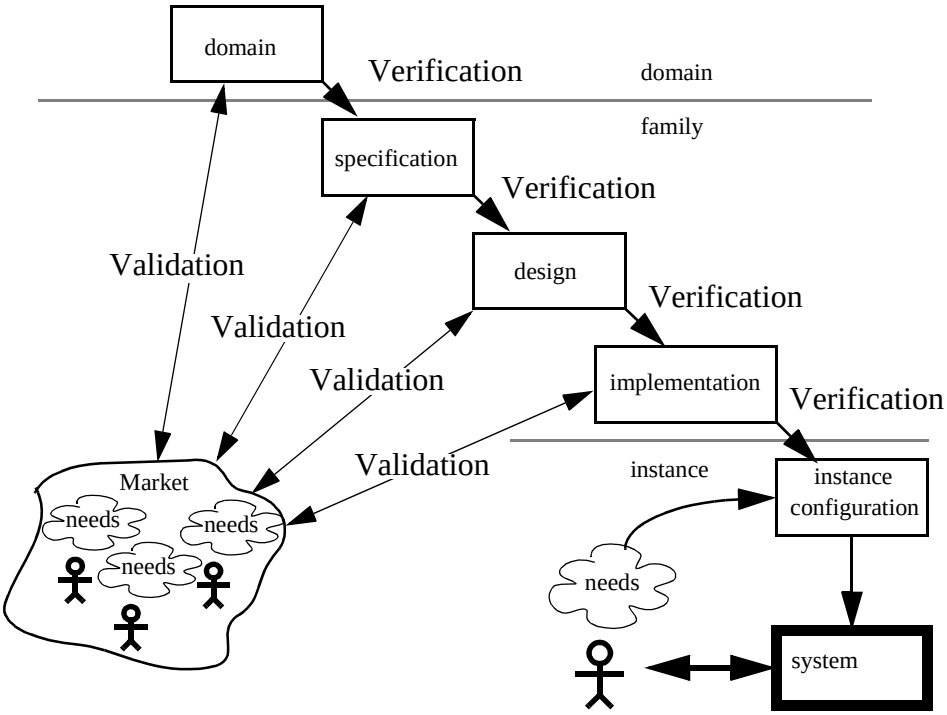
The section The Why, What and How of TIMe (p.4-9), together with the last part of TIMe Essentials (p.4-14) will tell you why you should use TIMe and what is special about it. Object and Property Models - and the Languages for describing them introduces UML, MSC and SDL for those that are not familiar with them. TIMe Essentials (p.4-14) provides an overview. The rest of the document is organized mainly according to the development activities of the method (see System Development Activities).

Supplementing elements of TIMe

As mentioned above, the full method book of TIMe contains important elements that for reasons of space are not covered in this introduction:

Figure 4-3: Verification and Validation

Open figure



- Verification and validation deals with “validation”, meaning to determine “whether we are making the right system” and “verification”, meaning to determine “whether we are making the system right”. TIMe presents several different approaches to verification and validation that correspond to different maturity levels of the companies (or projects):
  - test orientation: performed on the implementation of the system.
  - inspection orientation: involves human readers who control the quality of the descriptions.
  - animation orientation: executions of the system based on descriptions on higher abstraction levels than implementation.
  - formal analysis orientation: used in order to prove statements about the system, or to disclose hidden aspects of a system.
  - synthesis orientation: the implementation can be synthesised from a description of the requirements.

  TIMe presents techniques on all these levels. TIMe considers constructive rules to be superior to corrective measures.

- Process improvement deals with the introduction of TIMe into a company, and also covers process monitoring and improvement in general.

  We talk about achieving improved productivity *and* quality by setting goals and following one of several improvement methods, like “Mean & Lean”, the Capability Maturity Model (CMM) or the Risk Management Approach.

  It also discusses Risk Assessment and Control, as well as Change Cost Analysis and measuring the effect of introducing new tools and methods.

- Software configuration management covers how to control a product (in terms of descriptions) as it evolves. It describes levels of control and management, and describes means to cope with the complexity of product management.

  It presents our view on Configuration Management, which should help in defining plans for projects and companies.

  We identify 3 levels of control and management that can be useful: to achieve Configuration Management we need a platform for Configuration Control. To achieve Configuration Control we need a platform for Version Control.

  We give an indication of what can be obtained by state-of-the-art tools at each level.

- Metrics is about measurement in software development, a field that is known as metrics or software metrics. TIMe gives the answers to the questions

  - Why are we collecting measurement data?
  - How shall we collect measurement data?
  - How will we analyze the measurement data?

  A method that focuses on this, GQM - The Goal Question Metrics, is presented. It also discusses how to define useful metrics, and presents a few individual metrics.

TIMe from SISU

TIMe is a development of the Norwegian SISU I methodology described in *Engineering Real Time Systems* (Bræk and Haugen 1993, [24]). TIMe has been continually developed since its inception in the SISU project (1988 - 1996), see http://www.sintef.no/sisu - and has its name from the fact that it consists of an integration of method elements from different parts of the project. For people with no relation to the project, TIMe could as well have been an acronym for The Interesting Method, The Important Method, etc.

What’s in TIMe for the manager

TIMe saves time first time

Experience from the SISU project has show that TIMe can give you:

- 50% reduction in errors in delivered systems
- reduced development costs equalling or surpassing the cost of introduction on the first project
- 20% or more reduction of development costs on subsequent projects

- better control over the development process
- more flexible staffing, with less dependency on individuals
- smoother cooperation between professionals

when TIMe is carefully introduced into an development organisation, compared to a non-TIMe development paradigm. Some of these claims are proven by metrics programs in the SISU project, while others are based on interviews with managers.

What's is TIMe for the designer

TIMe is fun

Systems and software designers using TIMe typically experience

- more focus on designing functionality
- more precise communication with peers on design issues
- the pleasure of simulating (executing) designs at an early stage
- modern, state-of-the-art development tools
- less dependence on detailed development environment know-how, more focus on domain knowledge, making for easier shifts to new projects
- easier maintenance, simpler error correction
- the initial burden of learning a new development paradigm being outweighed by a better working environment and more job possibilities

compare to a pre-TIMe setting.

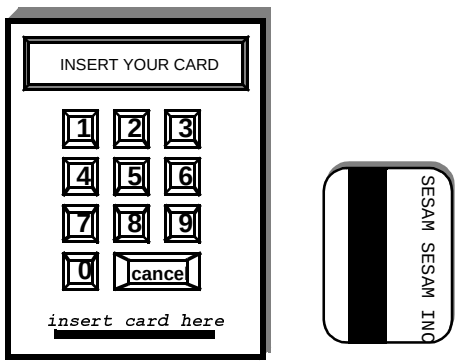
The Why, What and How of TIMe

Sesam Sesam

decided to go Object Oriented (OO), like rest of the world. Some years ago this would have been a tough decision, but now it looked as if this was the only right thing to do. The languages and tools were mature and there was plenty of help to get from books, courses and consultants - the problem was rather that there was too much help to get.

Figure 4-4: Sesam Sesam Inc

Open figure



For many years, the company had great success with their door locks and system keys. Their selling point was the highly flexible way that keys and locks could be coded to give user groups different access rights in a building complex.

But even their system had two main drawbacks: Lost keys and Code limitations. Whenever a key was lost, they had to change the locks to prevent unauthorised persons to gain access. Although the system was very flexible, it was based on fully mechanical locks and keys with inherent limitations in the coding.

To overcome these problems and to stay in front of competition the **Sesam Sesam** people were continuously looking for improvement opportunities. They saw that electronics and computers were rapidly becoming attractive alternatives as the prices went down and the reliability up, so they decided to go for plastic cards and panels with keyboards and displays at the access points.

The reason for going OO was that they would try to come in a situation where each delivery was composed from general components - up till now they had experience each delivery as almost a separate implementation. However, they also knew that their systems would be rather complex, involving real-time constraints and consist of large parts that were reactive of nature, so the choice of languages, tools and methods was not obvious.

They bought a tool, consulted the accompanying method book, and got the advice (in a condensed version): “Just find the objects (they are there to pick) and you will have the structure of your system”.

Introduction

For the development of complex telecom, real-time or reactive systems in general, a promising combination is to use:

- Object Orientation as a common approach to analysis, design and implementation, with concurrent processes as objects;
- Interaction Scenarios for the specification of communication between users and systems (use cases) and between objects of systems;

- State/Transition based specification of behaviour of individual objects.

Object orientation helps to master complexity by structuring in terms of objects and by factoring out common properties in general classes. Objects do not live on their own but communicate with other objects. Interaction Scenarios help to describe and understand even the most complex interaction cases. Describing the behaviour of each object in terms of states and transitions that are triggered by incoming signals from other objects has proven to be of great value for this kind of system.

The combined use of UML, MSC and SDL

TIMe supports this combination by the integrated use of

- UML for object model analysis,
- MSC for interaction scenarios, and
- SDL for specification and design of behaviour.

UML and SDL both support object orientation, there are tools integrating them, and the same tools also support MSC. UML is an OMG standard, while SDL and MSC are standards from ITU.

Why not just UML?

UML is accepted by the Object Management Group (OMG) as a Visual Modeling Language, and has received much attention from the software engineering community. The establishment of the UML Revision Task Force gives the potential methods user confidence that this will become the new industry language for systems design. However, UML is not mature enough to be adopted in its present form as a design language in an industrial context:

- The language is not yet stable, with considerable changes between 1.0 [21], 1.1 [146] and 1.2 [147].
- Support for real time concepts is only partial
- The language is not formally defined, with a self-referential meta-model and a semantics written in prose.
- The interchange format is not yet stable.
- Several textbooks exist [22], [51], [60], [112], [166], but many include features that do not adhere to the approved standard [146].
- No tools fully support UML, although many promise they will [149], [155], [156], [157].

For these reasons UML is not yet the ultimate, all-compassing language that its founders aim it to be. TIMe recognizes UML as a substantial improvement over predecessors like OMT [165], and currently recommends that parts of UML be used, along with industrial description languages like MSC and SDL, especially for illustrative sketches in early phases. If UML turns out to be what its founders aim at, while MSC and SDL do not evolve, TIMe may in the future become a UML methodology.

Presently we believe the combination of UML along with SDL and MSC following the formal rules to be defined by the ITU in the forthcoming Z.109 standard “SDL with UML” is the most promising. This is the strategy taken by the major SDL tool vendors [153], [154].

What about OMT?

OMT [165] is in widespread use, and many tools are available. TIMe sees OMT as an informal notation (not a language), and recommends that OMT be used in the same way as UML, so that the transition from OMT to UML has little risk involved. Version 3.1 of TIMe included an extension to OMT called OMT+-, which has now been discontinued.

Why UML and not just SDL?

Some companies have an established use of MSC/SDL, but for early analysis they normally use informal drawings. In order to become a little more precise, it could be argued that SDL can be used for this purpose. It has benefits in that it will ease the shift to design in SDL, but we advocate the use of UML for the following reasons:

- UML models do not require the same degree of formalization as SDL models do,
- UML supports relations (associations) between objects,
- UML supports fragments of object models, e.g. specifying relations in one fragment and attributes/operations in another.

However, TIMe also advocates that SDL is used for object modelling in case this is most appropriate. As an example, if it is important during analysis to specify some main states (modes) the system may be in, then this may directly be specified in SDL, as opposed to Statecharts in UML.

Why MSC and not Sequence Diagrams or Event Traces?

Some companies have chosen to go Object Oriented by means of UML or OMT. Tools for UML support Sequence Diagrams (and OMT tools supported Event Traces) for the formulation of properties of interactions, and to some degree these are integrated with the object modelling. The reasons for choosing MSC are still:

- MSC is more precise and richer in expression than Sequence Diagrams or Event Traces. HMSC, MSC references, conditions and in-line expressions are some of the distinguishing features of MSC.
- The instances in a Sequence Diagrams or in an Event Trace are objects from the object model, but often they should rather just be roles played by objects. Instances in MSC diagrams can represent both objects and roles.

Why SDL and not State-charts?

In some literature on state machine based specification of behaviour, Statecharts [76] as used in UML is the preferred notation. The reason for this is that the notion of nested states is appealing and that it produces compact specifications[1]. Statecharts alone is, however, not a complete language. It does not define communicating objects with data having the behaviour specified, so other notations and/or tools often add this. The main reason for using SDL is exactly that:

- SDL is a complete language that defines communicating objects (processes) with data attributes, operations *and* behaviour in terms of states and transitions;
- it also defines a structure of subsystems (blocks), and
- because it is a complete language, tools can (and do) support code generation from SDL specifications, and the integration with MSC allows for some degree of formal verification and validation.

1. There are currently initiatives in the ITU standardisation work to introduce nested states in SDL. This work is near its conclusion, and will be part of the year 2000 revision of SDL.

In addition, inheritance from the object model can be directly mapped onto inheritance for SDL process types, including inheritance of attributes, operations *and* behaviour (that is inheritance of states and transitions). This means that, if desired, it is possible to inherit "functionality" and not just "interfaces".

Another line of reasoning is that an interchange format for SDL descriptions between different tools is standardised [109] - this eases the transition from one tool vendor to another.

Are MSC and SDL perfect?

The above discussions are not meant to promote MSC and SDL as "the perfect languages". They are not. There are thing we miss in MSC, such as guard conditions, transitions names and the possibility to express constrains. There are things missing from SDL, not only the obvious lack of relations (which we recommend be expressed in UML), but also a number of niceties such as substates (i.e. the compact description they give, which SDL Procedures lack), for/while loops, expressions of algorithms and a dozen other issues.[1]

Neither MSC nor SDL are capable of formally defining execution time constraints, or expressing exact real-time behaviour in terms of process interleaving. Hence TIMe does not address mission-critical, "hard real-time" systems. Certain vendor-dependent solutions to this are provided by tool vendors [153].

We nevertheless recommend that MSC and SDL be used for the types of systems target by TIMe for detailed design and systems generation in an industrial context. MSC and SDL have proved themselves in many real-life projects, and are mature, albeit not perfect. UML is still promiseware.

Why a separate method on the combined use?

In conclusion, the combined use of UML, SDL and MSC seems a good idea. But there are still some issues to consider when using two slightly different object oriented approaches as represented by UML and SDL:

- uncritical use of relations (associations) will lead to problems when turning to design in SDL;
- aggregation was a special association in OMT, while UML and SDL support "real aggregation" (called composition in UML);
- UML (and OMT) supports multiple inheritance (the semantics of which will first become clear during design), while SDL supports single inheritance only - careful use of inheritance is therefore an issue.

In addition comes the object orientation you may have to use when considering distribution, e.g. CORBA, and this is yet another approach. TIMe has the answer on how to isolate the application specific aspects from the distribution aspects.

Why use TIMe?

There are already a number of methods supporting the combined use of UML, MSC and SDL, two of these are supported by tool vendors [174], [180]. Still there are some valid reasons for using TIMe:

1. Work is currently being carried out in ITU SG10/Q6 and Q9 to enhance the "year 2000" versions of MSC and SDL with such features.

- Most methods have a bias towards object modeling - “just find the objects and you are done”. TIMe comes with a system reference model and emphasises *property modeling* as equally important as object modeling. Property modeling includes use case modeling as a special case.
- TIMe has an answer to *where the design objects come from* and does not just provide technical guidelines for how to go from UML to SDL.
- As a mechanism not supported by other methods, TIMe tells you how to make *frameworks* in SDL. Frameworks produce the most effective reuse.
- TIMe represents many years of experience with system development and object orientation.
- TIMe bridges the gap between the user’s world of needs and the designer’s world of objects.

TIMe can be used as a supplement to other methods. Tool vendor specific methods will always be useful, as their elements most probably will be supported by the tools.

How to use TIMe?

TIMe is available both as printed material and as an “electronic book”. The electronic version allows you to follow links in order to read what you want, e.g. at a specific stage in the development process. The electronic version allows for company specific extensions, with links into and out of those parts of TIMe that are used.

TIMe Essentials

This section gives a description of the *essential elements* of TIMe and what makes it *different* from other methods.

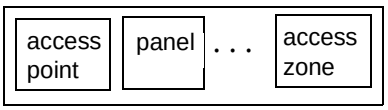
Systems and system descriptions

TIMe is a system development method. A *system* is a part of the world that a person or group of persons during some time interval and for some purpose choose to regard as a whole, consisting of interrelated components, each component characterised by properties that are selected as being relevant to the purpose. A system is not a description on a piece of paper, but something actually existing as a phenomenon in the real world. This puts the system apart from the description of the system. The system actually exhibits behaviour, while its description is a dead pile of paper.

Systems made by means of TIMe (and by means of many other methods) are produced by making *descriptions* in a variety of languages and notations. These descriptions prescribe how systems should be generated by having computers and similar equipment (*platforms*) execute these descriptions.

Systems consist of objects. In order to describe them, classes of objects are defined and described. In short, methods consist of approaches, guidelines and techniques for identifying and describing classes of objects.

With the advice “Just find the objects (they are there to pick) and you will have the structure of your system”, the development group at **Sesam Sesam** imagined the picture below. This is of course oversimplified, but in fact most methods they consulted advocated no more structuring of systems than this. This had to do with the fact that most object oriented languages support only a flat structure of objects with relations - aggregation is just a special relation between objects.



The process of finding the objects (or rather classes) was driven by the developers, but management had learned that OO was the best way to model the real world.

The people that were involved with this real world were the market people and the people responsible for customer solutions. They were therefore brought into the process and asked to contribute to the object modelling.

It turned out, however, that these people were so heavily stuck in the(ir) real world that they could only think in terms of required (or desired) properties of the system as such (in terms of functions, list of functions, features, requirements, etc.). They had the picture of a system as illustrated above, that is a list of services.

change PIN code
.
.
block access point
.
accept/reject users

Some of these services were defined on the basis of use cases. With these two very different perspectives on a system, it was no surprise that finding the objects turned out to be finding the “design/implementation” objects, and that properties were not taken into account. Finding objects by functional decomposition was, correctly, regarded as a bad thing, and was not considered at all. So where had all the properties gone in this new object oriented way?

Properties and objects

TIMe has the two dimensions *properties* and *objects* as integral parts of the method.

Systems consist of objects. Objects and systems have properties

A system consists of a set of objects. Objects are described by:

- *object models*, that model how a system or a set of related classes are composed from objects, connections and relationships.

Systems and objects have properties (both provided and required). Properties are described by:

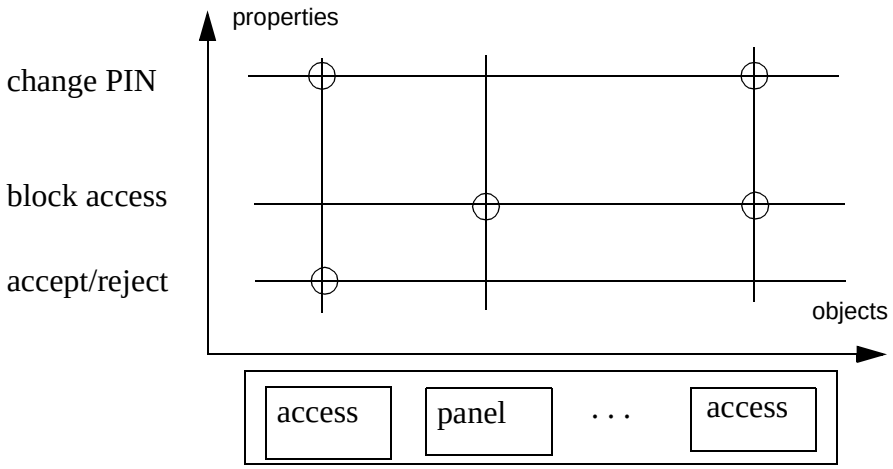
- *property models*, that model the properties of a system or object without prescribing a particular content or implementation.

Object models are constructive in the sense that they describe how an object is composed from parts, and is the perspective of designers. Property models are not constructive, but are used to characterise an object from the outside: behaviour properties, performance properties, maintenance properties, etc. This is the perspective preferred by users and sales persons. It is also the main perspective in specifications.

TIMe provides some of the answers to the challenge of system development: to identify objects and give them properties so that they contribute to the properties required of the whole system, see Figure 4-5 (p.4-15).

Figure 4-5: Matching objects and properties

Open figure

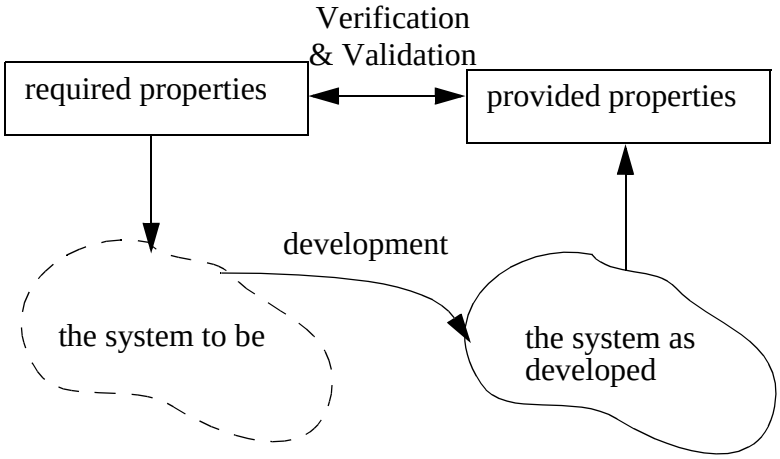


Properties

A central idea in TIMe is that every object (and system) is characterised by *provided properties* that can be matched against *required properties* (see Figure 4-6 (p.4-16)).

Figure 4-6: Required and provided properties

Open figure



Of special interest are interaction properties, where a property involves the interaction between the system and one or more users of the system or other systems in the environment, or between objects in the system. The "accept/reject user" property in Figure 4-5 (p.4-15) is an example of this: it involves the user entering the card and code to the system, and the system answering back with either OK or not OK to enter. In Figure 4-7 (p.4-16), part of this is specified in MSC.

Figure 4-5 (p.4-15) only indicates that access point objects are involved, but during the design we shall see that both panels and a central unit will be involved.

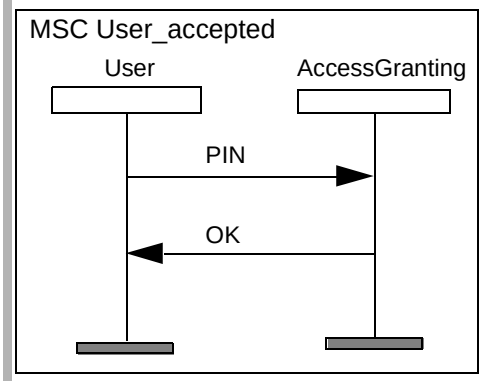


Figure 4-7: Simple interaction property model

Roles

TIMe makes it possible to express property models without referring to specific objects. Sometimes we need to specify properties without knowing the objects they shall be associated with, and we may want several different objects to share the same properties (e.g. a common interface). TIMe also makes it possible to compose the properties of an object from parts described in different property models.

The notion of *role* makes this possible. Roles are used to represent objects in property models, and we may compose the properties of an object from roles described in different property models.

One of the instances in the MSC in Figure 4-7 (p.4-16) is "AccessGranting". This is not an object of the object model, but a functional role. Behind this name can be hidden any structure of interacting objects. At some point in the development it is necessary to associate the functional role with an object of the object model. We call this *synthesis*.

TIMe uses three main categories of roles:

- service roles, which are the observable behaviour of an object in a given service;
- interface roles, which are the observable behaviour at given interfaces;
- association roles, which are the conceptual constraints on objects that participate in associations (relationships).

After having clarified that both object and properties are supported, the group at **Sesam Sesam** went for the TIMe method and gave it a try. They were successful in working together towards a common object model. The market people saw that their functions (properties) would be combined with objects, and that object modelling was not the only activity.

- "OK, we started by making an object model of the current system as it appears to its users in the real world - now what do we have?"
- "We have the main user panel, and we have the gates where users get access to the system. We also have the mandatory objects that control the equipment, without those there would not be any system."
- "..."

This process was driven by a number of use cases, so it was not surprising that the group ended up with an object model where the dominant object was the main user interface object of the system. Most of the properties became associated with these objects.

Interface and application given aspects

TIMe makes a distinction between the (user) *interface* given aspects of objects, and *application* given aspects of objects, see Figure 4-8 (p.4-17).

In some cases these are two different sets of objects, in other cases they are just two different aspects of objects - this is indicated by the jagged line in Figure 4-8 (p.4-17). The important thing is to make the distinction and be able to maintain it. The interface may change e.g. with new technology, while the application objects providing the service properties of the system will typically have a longer life.

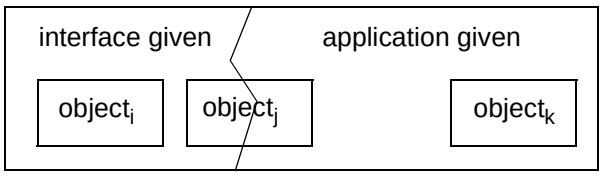


Figure 4-8: Interface and application given aspects

We shall later see that the application given aspects can be decomposed into system given and domain given aspects.

After having learned the distinction between interface specific and application specific objects, the question was how to get at some reasonable application specific objects. The people at **Sesam Sesam** were experts on their kind of equipment and other technical systems that were to be used, so they identified objects that modelled this equipment and other (technical) systems and devices in the real world of the system.

The object model turned out to have, not a black box, but a white hole at the place where the system object model was supposed to be. The problem was: what kind objects should the core of the system consist of; it was obvious that there would be objects that handled the user interface and the interface to other systems, but apart from that??

- Says the TIMe consultant: "What about a domain analysis"
- "What was the word again? And what does it mean?" says the project leader.
- "OK" - says the TIMe consultant - "what is the system all about, irrespective of how it is realised, what is the basic problem(s) that the intended system is supposed to solve; which kinds of entities and/or events in the so-called real world or in your imagination are handled by the system?"

Systems belong to domains and are used in environments

TIMe makes a distinction between a *domain*, the *systems* within the domain, and the *environments* in which the systems are used. While systems belong to a domain, in that they handle the same types of phenomena, they exist and are used in an actual environment, see Figure 4-9 (p.4-19). Accounting systems are different from access control systems in that they belong to different domains. The example system in this introduction to TIMe belongs to the access control domain, and that includes phenomena and concepts like access points (where users get access or not), access zones (to which users would like to get access), PIN codes, etc.

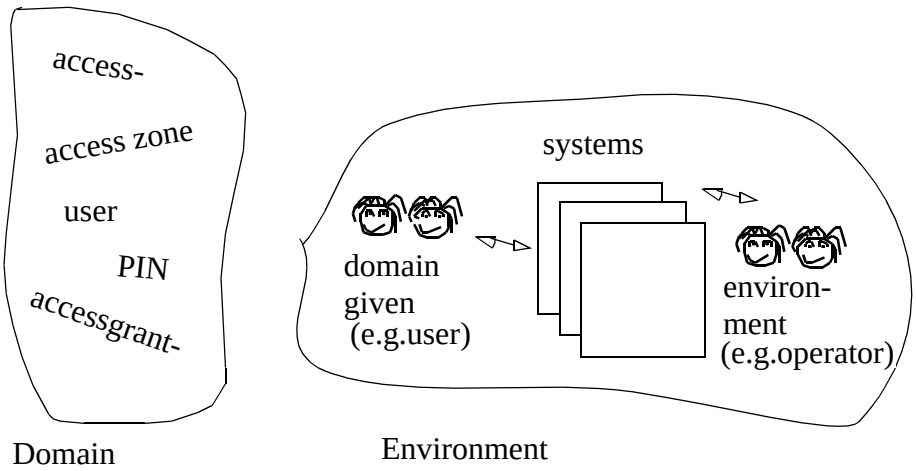
The domain models a part of the real world having similar needs and terminology where a system instance may be a solution to some need. The domain is not specific to a particular system or system family, but rather to a market segment. It covers common phenomena, concepts and processes that need to be supported irrespective of particular system solutions.

Required properties derived from an actual use situation may come in addition to the properties stemming from a domain. Properties required by the actual environment and by its realization are very specific, while domain-given properties are often more general and express idealized needs.

After this distinction it turned out that the project team at **Sesam Sesam** were not only experts on the technical parts, but they also knew what the system was supposed to do. By taking one step backwards and identifying entities and events in the domain and representing these by object classes, they arrived at some domain specific classes that may be candidates for objects in the system.

Figure 4-9: Domain, environment, and systems

Open figure



The access control domain has to do with controlling access to *access zones*, based on e.g. card *codes* and optional *PIN* codes. *Users* present their cards and PIN code at a number of *access points*. Some access points may be *blocked* even if a valid code is entered, while other access points may *log* what goes on at the site. Users may also *change* their PIN code.

The environment of a specific access control system may in addition to the users have *operators* that have other requirements to the system, e.g. getting the status of access points.

General properties like *access granting* come from an analysis the domain, while properties that have to do with the specific use of the system (e.g. how to read cards and control doors) and operator requirements come from an analysis of the environment.

Domain, interface and system given aspects

Domain objects and their properties are not enough to provide the required properties of the whole system. Many general properties can be provided by the domain given objects, but some properties will often be required in addition. For instance properties related to the operation and maintenance of a specific system.

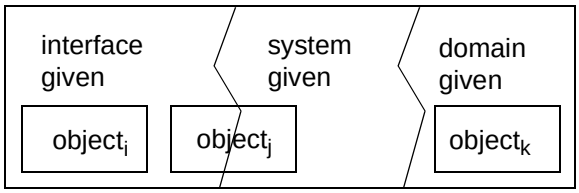
This is reflected in the *system reference model* of TIMe: In addition to the domain and interface given (aspects of) objects, the application given objects may have some aspects that are special for this *specific system*, in addition to the general properties of domain given objects.

The object model of a system is therefore divided into three aspects:

- the *domain* given aspects
- the *system* given aspects

- the *interface* given aspects.

These aspects may be whole objects or just aspects of objects.



Domain given aspects come from an analysis of the domain, *interface given aspects* have to do with user interface, interface to other systems or to controlled equipment, while *system given aspects* are those aspects that arise because there is a system, and that are specific for the system.

Domain given aspects have a larger potential for being reused in other systems in the same domain than the system given aspects, and interface aspects may have to be modified when the interface technology changes.

The interface given aspects of the access control system are illustrated in Table 4-1 (p.4-20).

Table 4-1: The three aspects of the access control system

interface given aspects	system given aspects	domain given aspects
• panel for entering card code, PIN and for displaying messages to the user • door control, unlocking the door	• operator requirements • validation shall be done by a central unit • backup requirements	• access points • access zones • users • access granting • ...

Abstract and concrete descriptions

Descriptions suitable for execution by existing platforms contain a lot of detailed, *concrete* description elements (implementation details, platform specific details, etc.). Descriptions suitable for system developers in their strive to match required properties expressed by users, owners o.a. are preferably more *abstract* in the sense that they describe systems in terms that reflect established concepts within a given domain.

TIMe achieves abstraction by supporting UML and MSC for analysis models and SDL for design models. UML is a notation that enables informal, abstract object models, MSC describes use cases and interaction between objects, and SDL supports abstract

descriptions that (by including concrete description elements) automatically may be transformed to concrete implementations. The use of abstract descriptions is one of the main ingredients in property oriented development.

In the access control system example, the *system* is the collection of panels, doors and program executions running on some processor(s), while *system descriptions* will include (abstract) overview descriptions of the total system	(including pure hardware components), interface description of the hardware components, SDL descriptions of some of the program executions, and (concrete) Java or C++ code for the SDL run time system and for hardware drivers.

Abstract descriptions are organised in two main parts:

Application
- an *application* part that describes what the user environment wants the system to do;

Infrastructure
- an *infrastructure* part that describes additional behaviour and supporting functionality that needs consideration, e.g. in order to fully simulate its behaviour. This may e.g. include support for distribution, exception handling etc.

The reason for this distinction is that systems within the same domain and in the same family (see below) often will have the same infrastructure part, but different application parts. Reuse of infrastructure is eased by keeping them separate, and application evolution is simplified.

Concrete models describe the *implementation architecture*. This is a high level description of the physical implementation. The purpose is to give a unified overview over the implementation and to document the major implementation design decisions.

Each object has attributes and behaviour, is related to other objects, and is structured for two different reasons:

Context
1. so that it *models* the corresponding domain entity and *represents* itself adequately to the objects in the *context* of the object (for the system object this means the objects in the environment);

Content
2. so that the object is completely defined with respect to its realization on the executing platform. We will talk about the *contents* of the objects in contrast to its context, see Figure 4-10 (p.4-21).

Analysis will produce *specifications* of objects, while design and implementation activities will produce *designs* of objects. In specifications the object context and external properties are defined. Some limited parts of the content may also be specified, see Figure 4-10 (p.4-21). In the design the remaining content is defined. The specification of an object includes what is needed to use the object - and that may be more than just an interface specification.

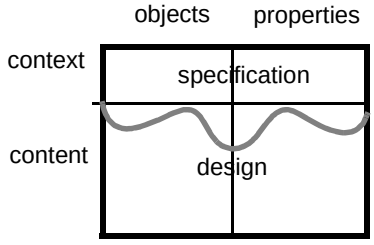


Figure 4-10: Context/

When deciding upon what belongs to the domain and what is more system specific, the main distinction is between the domain and single systems within the domain. During development we often think in terms of making one specific system. We talk about the "system" and the "domain", and about e.g. "system specification" and "system design".

Families

It is, however, fruitful to think in terms of families of systems and really make "system family specifications" and "system family designs". The idea is to focus development and maintenance effort mainly on the families, in order to reduce the cost and time needed to produce each particular instance, and to reduce the cost and time needed to maintain and evolve the product base.

A system family is a generalised system or set of component types (classes) that can be configured and instantiated to fit into a suitable range of user environments. They represent the product base from which a company can make a business out of producing and selling instances.

TIMe provides guidelines on how to make system families in addition to single systems. Where practical, system types and classes will be defined from which complete system instances may be generated.

Frameworks

The notion of framework is one important mechanism for defining families of systems. A family comprises more than just a type of system from which several system instances can be generated. In addition a family includes e.g. the necessary documentation, manuals, etc. that make up a complete product.

At the heart of a system family lies the parts that are generated from an SDL design. This may either be a complete SDL system, a set of SDL systems, or a set of general block types and process types that can be (re)used for making systems.

Application and infrastructure

The SDL descriptions will be organised according to the distinction between application and infrastructure. It is normally the case that different systems within a family will have the same infrastructure but slightly different application parts, and when making different systems it is desirable not to change or even consider the infrastructure part (besides what it offers). A framework defines the composition of the infrastructure parts and application parts in such a way that different systems can be made by only changing the application parts.

> Says the people at **Sesam Sesam**: "We have seen frameworks work for window systems and implemented in object oriented programming languages, but we guess frameworks are not for us, now that we have chosen SDL...

The notion of framework is not special for TIMe. Within the field of object orientation, a framework is well-known (*"In object oriented systems, a set of classes that embodies an abstract design for solutions to a number of related problems."* - Free On-line Dictionary of Computing), and there are good examples of frameworks, e.g. window systems. The use of frameworks supports the (re)use of whole designs and not only single classes.

What is special for TIMe, however, is that this idea is adapted to SDL. A framework can be defined as an SDL system type, and the different systems as instances of subtypes of this system type. TIMe provides detailed guidelines for how to do this.

Languages and notations of TIMe

The main languages and notations of TIMe are UML, MSC and SDL. For readers not familiar with these, please consult Object and Property Models - and the Languages for describing them. The following is a very short introduction.

UML for analysis and design object modelling

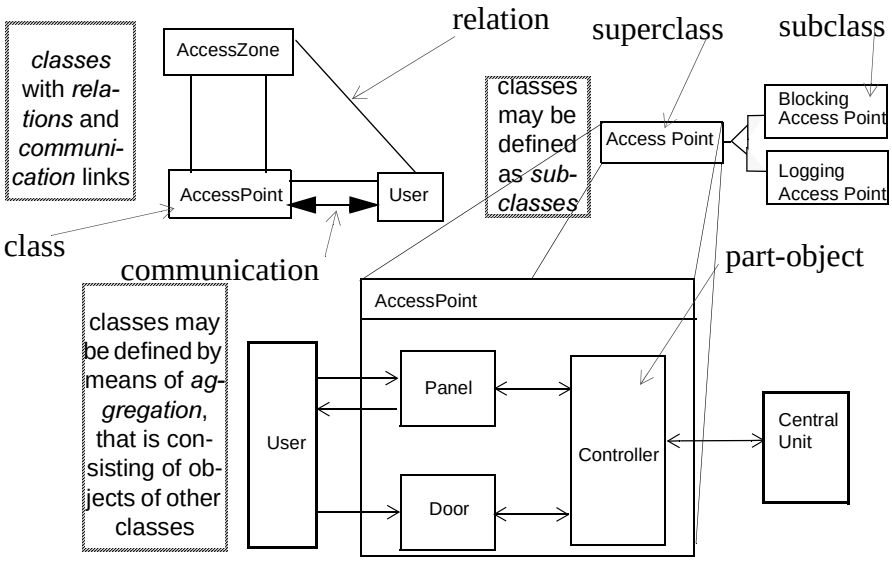
TIMe uses a UML for object modeling in terms of

- *classes* (with attributes and operations) representing application specific concepts, with *relations* (associations) and *communication links*;
- *inheritance* between classes in order to express *specialization* of application concepts;
- *aggregation*, that is objects defined by means of *part objects*.

TIMe defines its own approach to object orientation, and UML is used as a notation for expressing this. UML matches the TIMe object orientation better than OMT.

Figure 4-11: UML for object modelling

Open figure



MSC for specifying interaction scenarios

TIMe uses MSC as its basic notation for property modeling. MSC highlights *interaction* between *instances* based on *messages*. Instances may represent objects from some object model or just roles played by some objects. A message is asynchronous, the *output* must come before the corresponding *input*. The *events* on the timeline of an instance are strictly ordered, and the distance between events is not significant.

An MSC document consists of a set of MSCs. Different MSCs within the same MSC document are related by *conditions*. A condition is a label which signifies a global or local state. Conditions can be used to mark situations where there are different alternative continuations, and they may describe looping.

Instances may be *decomposed*, in order to see the details of this in terms of further MSC diagrams.

SDL for design and for specifying behaviour

SDL is the main language for design, and the only language for specifying behaviour.

An SDL *system* consists of a number of *blocks*, connected by *channels*. Each block may either consist of a *substructure* of blocks, or of a set of *processes* connected by *signal routes*.

Processes execute concurrently, communicate by sending *signals* (non-synchronized), and have their behaviour described in Finite State Machines extended with variables, procedures and transitions.

SDL may define types and subtypes of systems, blocks and processes.

Figure 4-12: MSC for interaction properties

Open figure

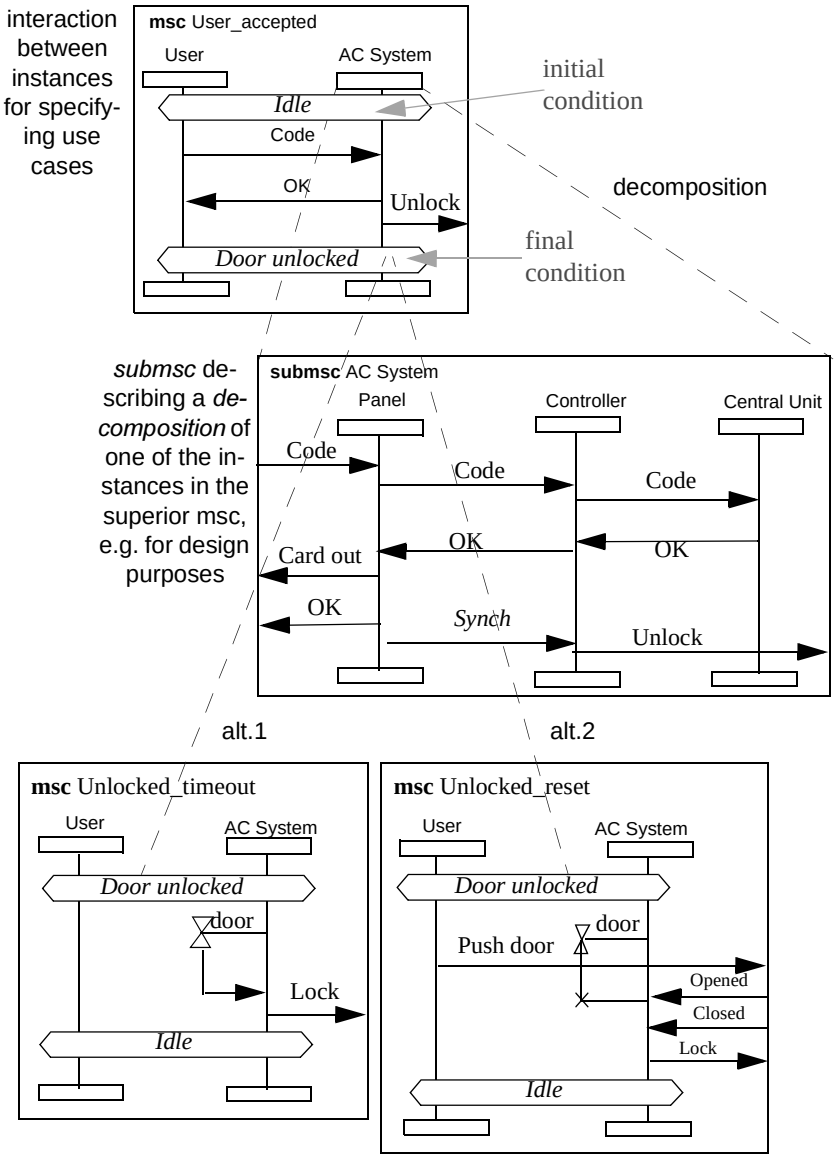
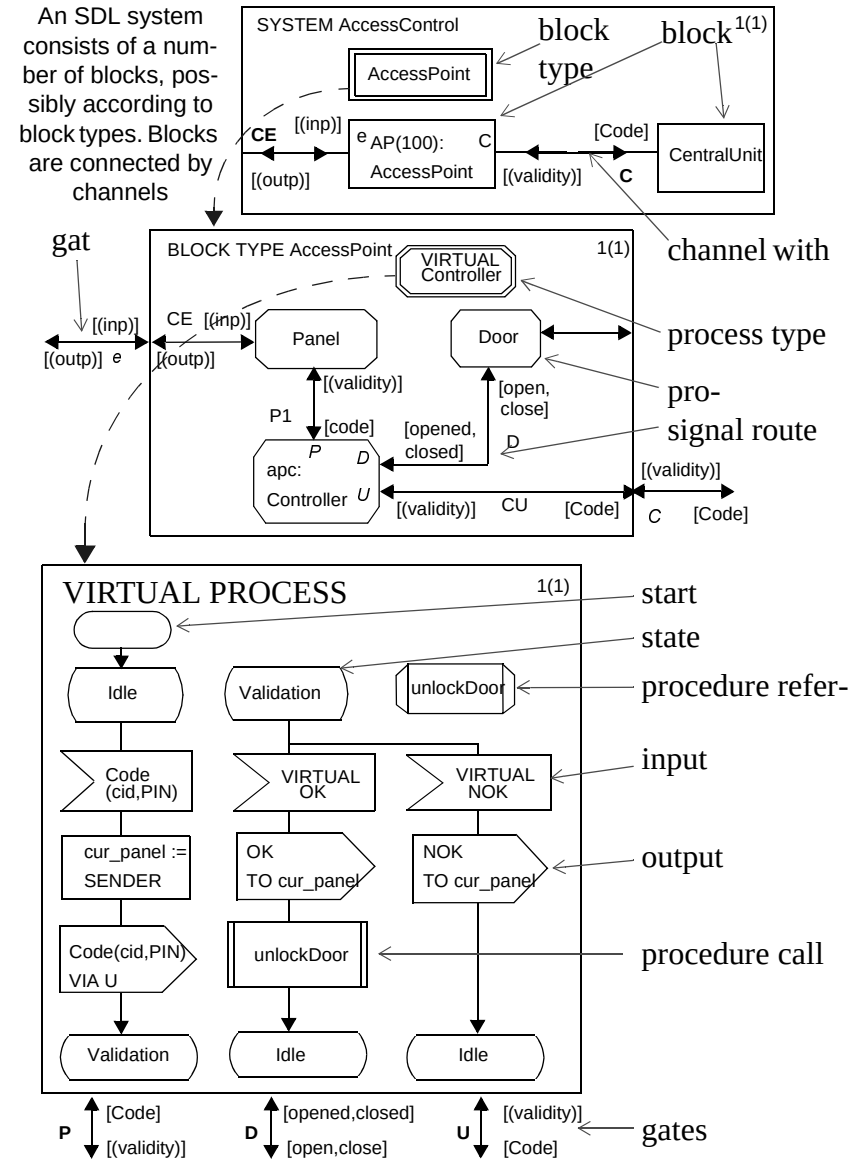


Figure 4-13: SDL for design and specification of behaviour

Open figure



System Development Activities

The following overview of TIMe is structured according to system development activities, and the corresponding kinds of models and descriptions are introduced along with the activities. In this overview emphasis is put on the activities leading to implementation.

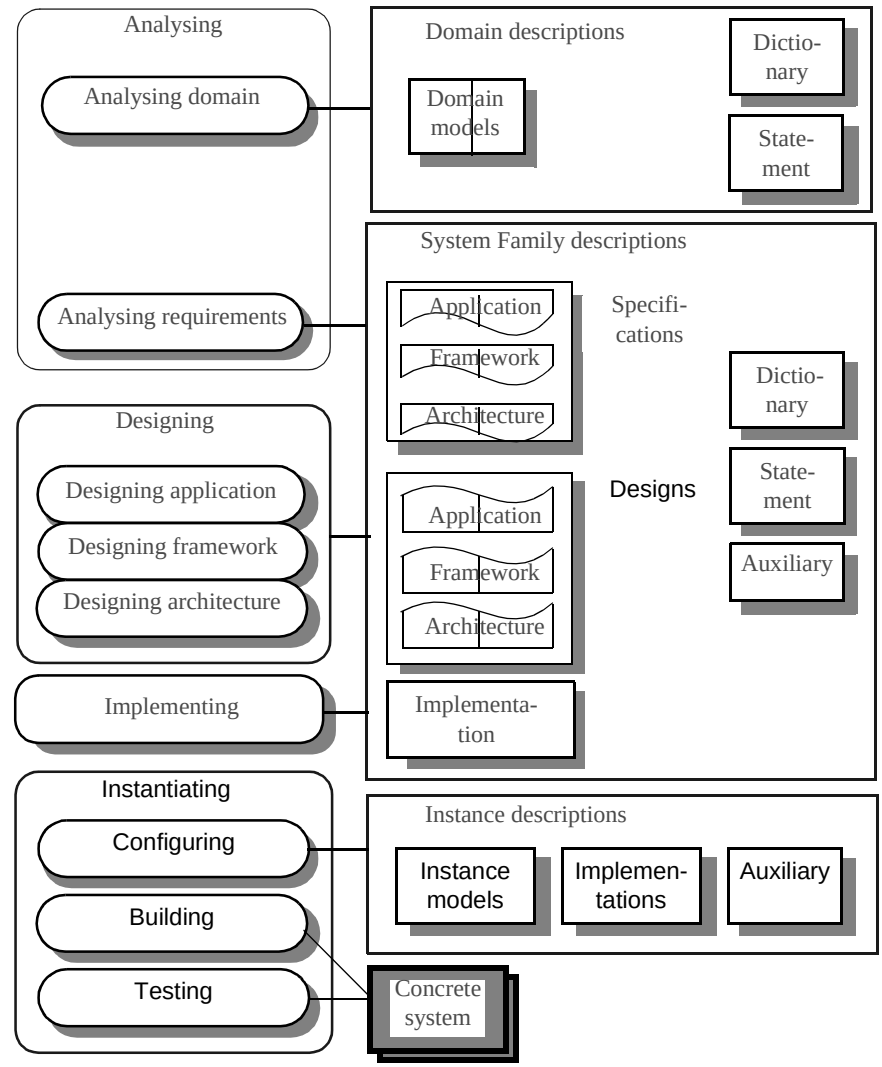
This chapter covers the following activities:

- Analysis (p.4-28)
- Domain analysis (p.4-29)
  - Domain Statement: what is it all about (p.4-31)
  - Domain object model: modeling the established domain concepts (p.4-32)
  - Dictionary: not just a data dictionary (p.4-34)
- Requirements analysis (p.4-37)
  - Application specification (p.4-40)
  - Architecture specification (p.4-45)
  - Framework/Infrastructure specification (p.4-46)
- Design (p.4-48)
  - Application Design: where the real functionality is designed (p.4-49)
  - Architecture Design: choice of implementation platform (p.4-58)
  - Framework Design: from Infrastructure to Framework (p.4-59)
- Implementation (p.4-66)
- Instantiation (p.4-66)

Guidelines on Object and Property Modeling are provided in a separate chapter (Object and Property Models - and the Languages for describing them). These are modeling techniques that are part of many activities and therefore most conveniently covered in one place. That chapter includes guidelines on the matching of properties and objects, and on the transition from UML to SDL object models.

Figure 4-14: The main activities in TIMe

Open figure

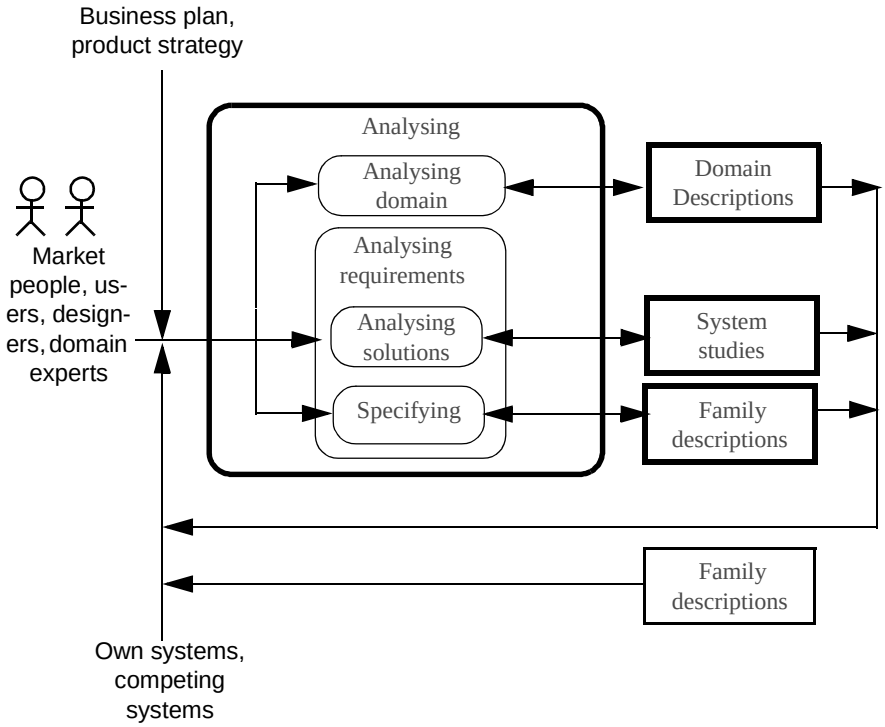


Analysis

The objectives of analysis are to understand the domain and what users and other stake holders want to achieve, i.e. their needs, to find improvements to existing systems, or to plan new product families that will give valuable improvement and thus create business in the future.

Figure 4-15: Analysing

Open figure



Product Planning

Product planning is another word for analysis. Product planning is a strategic process at the company level. Its main goal is to consider needs existing in the market and plan new products or enhancement to existing products. Few tasks are more critical to the success of a company than its product planning. Product development is a process which produces the new products or product enhancements that are planned.

Why domain descriptions?

At the product planning level, domain descriptions are used to collect and organize domain knowledge in a way that will enable product development to work more efficiently.

At the product level, product families will enable faster and more cost effective configuration and production of system instances, while common components will be used to develop product families more efficiently.

Product planning consists of two main activities: Domain analysis and requirements analysis. The task of the latter is to plan what parts of a domain to support by a new system family and to specify its required properties.

Domain analysis

The first analysis will be an analysis of the domain. This includes identifying which phenomena and concepts (like access zones, access points) are part of the domain, with focus on concepts. The result is represented by two *domain models*:

Domain object model in UML

- A *domain object model*, that is a collection of classes with attributes, relations and communication connections that describe the general concepts of the domain, without going into details needed for design and implementation.
  - UML is the main notation used for this kind of modeling, but if it for some reason should be important to describe some general states and transitions, then SDL is used.

Domain property model in MSC

- A *domain property model*, that is a description of properties of domain object classes, and of roles.
  - Domain objects do not have to be just “data (passive) objects”, so properties may involve interaction properties - they are described using MSC. Other kinds of properties are described using natural text, e.g. organised in lists of required properties. If interaction properties are not obviously associated with objects from the domain object model, we will say that they are associated with roles.

Two additional *domain descriptions* are recommended:

Dictionary

- A *dictionary*, that is a list of terms with an explanation of their meaning, including each of the elements of the domain object model. A dictionary is not just a data dictionary, it also includes definitions of concepts that exhibit behaviour.
  - Dictionaries are described by structured natural text.

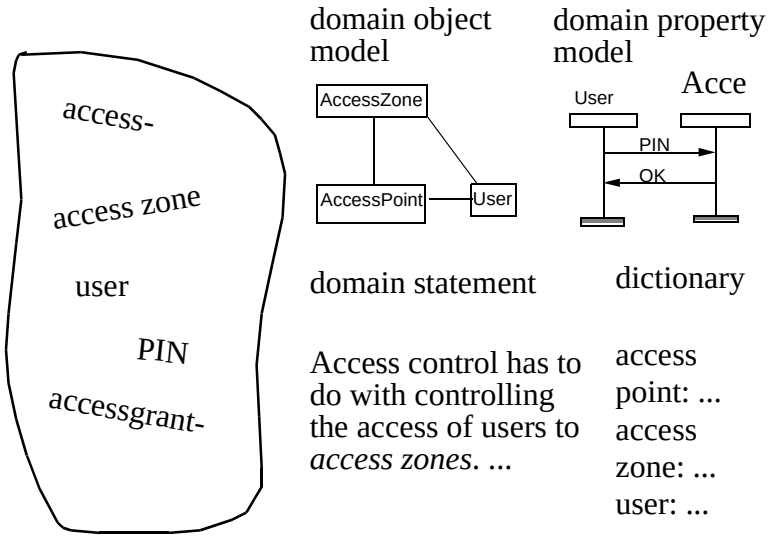
Domain statement

- A *domain statement*, that is a concise description of the domain with focus on stake holders and their needs, the essential concepts, functions and work processes, rules and principles.
  - It is normally sufficient to express the domain statement informally using natural language and drawings, but one should try to be as clear and precise as possible.

These models and descriptions represent the understanding of the domain common to users, owners and developers of systems in the domain, see Figure 4-16 (p.4-31).

Figure 4-16: Domain Analysis Models and Descriptions for the Access Control Domain

Open figure



Domain Statement: what is it all about

The domain statement leads to the very first understanding of what the domain is all about. It helps to clarify needs and to understand the real purpose of systems in the domain. It also serves as an introduction to the other domain descriptions.

The domain statement can often be based on existing prose descriptions. There may be descriptions of earlier systems, there may be textbooks on the subject and there may be informal statements about the system.

Domain Statement

By considering similar systems on the market, by analysing the needs and by consulting domain and market experts, the short Domain Statement V1 (p.4-32) is written. It seeks to describe what is special for this domain in contrast to other domains, and is used to guide what to include and not in systems.

Figure 4-17: Domain Statement V1

Open figure

Area of concern

Access control has to do with controlling the access of users to *access zones*. Only a user with known identity and correct access right shall be allowed to enter into an *access zone*. Other users shall be denied access.

Stakeholders

Users of the system, those responsible for the security of the access zones.

Services

The user will enter an access zone through an access point.

The authentication of a user shall be established by some means for secret personal identification (code). The authorisation is based upon the user identity and access rights associated with the user.

A supervisor will have the ability to insert new users in the system.

Users shall be able to change their secret code.

Helpers

We assume some central means to establish access rights automatically.

Domain object model: modeling the established domain concepts

Classes and objects

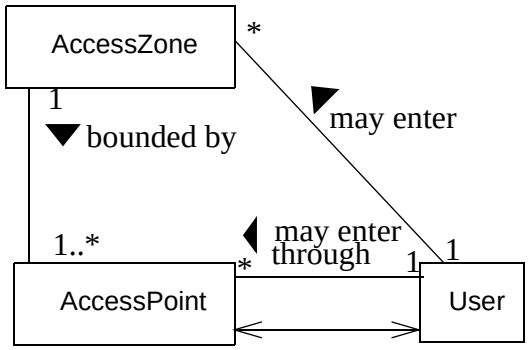
A domain object model describes the domain from an object oriented perspective. It defines *classes* which represent concepts in the domain, and *objects* which represent phenomena in the domain. It defines the *attributes*, the *operations* and the *behaviour* of objects as well as *associations* and communication *connections* between objects.

- The **Sesam Sesam** project leader tries: “Aha, so domain modelling resembles the way database applications are made: we make a data object model for the domain, that is the objects that the systems in the domain must *know* about, and then we make different database applications with this data object model as a basis.”
- “That´s right, we know in fact that we have to keep a database of what users have ordered, so that they can be billed, so that statistics can be produced, etc.” says a project member.
- “OK, this seems to be a fairly simple distinction: “passive” data objects for the domain and then more “active” “controlling” application objects. As far as I remember this is also the main distinction in the Use Case approach of ObjectOry” says another.
- Says the TIMe consultant: “This will work as a starting point, and for some systems this will do, but for most domains we do not have to restrict ourselves to regard the domain specific objects as data objects only: if there are general properties that have to be fulfilled by “active” objects (that is objects with behaviour, with a life-cycle and often concurrent with other active objects), then these objects obviously are domain objects, and their classes will be used for many systems in the domain. In TIMe the dimensions domain-system and active-passive are two different dimensions - it is not so that domain object are always passive and system objects always active!”

With this definition of domain object model, it is rather straight forward to identify the domain specific objects. In our example it turned out that some of these were really “active objects” (e.g. User and AccessPoint in Figure 4-47 (p.4-70)). Note that this object model comes about when considering only classes, relations and connections. If only considering e.g Use Cases, AccessPoint may not have turned up, but rather a role like AccessGranting.

Figure 4-18: The access control domain

Open figure



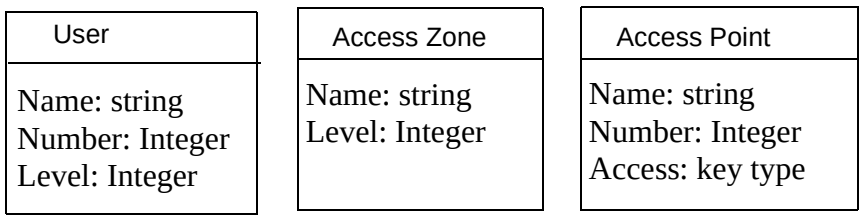
In this part of the domain object model it is described that a User may enter more than one AccessZone, and may therefore use more than one AccessPoint. There may be more than one point at which a given AccessZone can be entered and exited.

The User and AccessPoint objects will be active objects, while AccessZone objects are passive. This is indicated by the communication *connection* between AccessPoint and User.

The corresponding property model reflects that User and AccessPoint interact. Note that User is the class of real Users, and not the class of User objects eventually representing users within the system.

Figure 4-19: Attribute specification

Open figure



Dictionary: not just a data dictionary

The objective of the dictionary is to define terminology and thereby enable precision in communication between people involved. Terminology names the domain specific concepts and defines their meaning.

An important set of concepts in the dictionary is the set of concepts that are covered by the corresponding domain object model. There may also be phenomena, like e.g. access granting, that will not be covered directly by a class in the object model, but by property models involving more than one class of objects.

It is important that not only “data” concepts are included in the dictionary, but that typical “event”- or “action” concepts also are included - hence the name Dictionary and not Data Dictionary.

Figure 4-20: Domain specific Dictionary

Open figure

Access point	A point of access into an access zone.
Access zone	A physical or logical zone guarded by a set of access points.
Authentication	To establish the identity of a user.
Authorisation	To establish the right of a user to enter an access zone.
Authorizer	The entitity which determines authentication and authorisation.
PIN	A personal identification means.
User	A person with known identity with authorisation to enter specific access zones.
User name	A user name.
Access Granting	The role of granting (or not granting) a user access.

Domain property model: modeling the needs

A Domain Property Model is used to describe the problem domain from the Property perspective. It includes functional and non-functional properties.

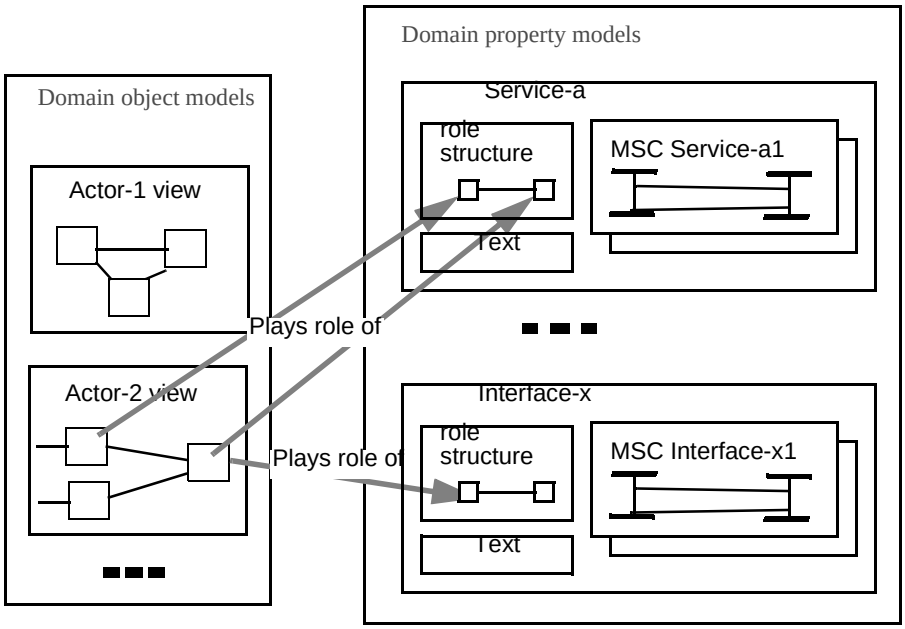
Functional properties are considered as projections of object behaviour, and are described using text, role structures and MSC.

Important properties for the systems that TIMe is intended for are properties of interaction between parts of the systems and parts of the environment. Some methods recommend pure role modeling for this purpose: that is all instances involved in interaction scenarios are roles played by some objects that will be found during design. Other methods (like UML) use the object model as the basis for interaction scenarios, and therefore only have objects as instances in interaction scenarios, never roles.

TIMe supports a mixture: if interaction properties are obviously associated with objects already identified in the object model, then the property models describe the properties of these. On the other hand, if the object model has not even been identified, it is still possible to make interaction scenarios only involving roles. During design, roles will be assigned to objects. The relationships between objects and properties are illustrated in Figure 4-21 (p.4-36).

Figure 4-21: Domain Models

Open figure



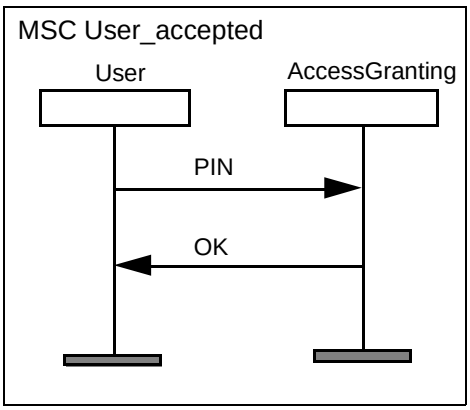
Role structures

Text is used to give a textual explanation of a service or interface. *Role structures* are UML instance diagrams that represent the roles of the service or the interface. The objects in role structure diagrams can be considered as anonymous objects. They will be related to object model objects by role association links, and to the instances in the service MSCs through the same name.

When the system is designed, the domain property models will also be valid property models of the corresponding (domain given) system objects. Properties belonging to the domain will be candidates for properties of several systems in the domain.

Figure 4-22: MSC User_accepted

Open figure



We know that there will be AccessPoints and that the Users will interact with these in order to enter an AccessZone, but it is not obvious if Access-Points are the objects that will grant access.

If it is important to express this uncertainty, then we define Access-Granting as a role - in other parts of the development we will assign this role to one or more objects, and probably AccessPoint will be one of them.

MSC does not take any stand as to what the instances are - an instance just represents one sequence of events (sending and receiving messages).

During domain analysis, **Sesam Sesam** used the set of rules/guidelines being part of TIMe. The following is a list of *some* of these:

- As a start, consider how things are done today and describe the existing domain. Then consider how it may be improved and develop a new domain description.
- Focus on abstract objects that are essentially needed and avoid system specific solutions. This does not exclude elements that eventually will be part of systems. The essential thing is that the Problem Domain generalises over system specific solutions. Classes of objects coming from an analysis of the Problem Domain are candidates for reuse across systems, but reuse requires at least one use.
- For each stake holder, describe their needs for services and interfaces.
- Represent every actor as a type with context in the object model and describe its services in property models.
- When systems are defined classify the entities into interface, system, and domain specific parts.

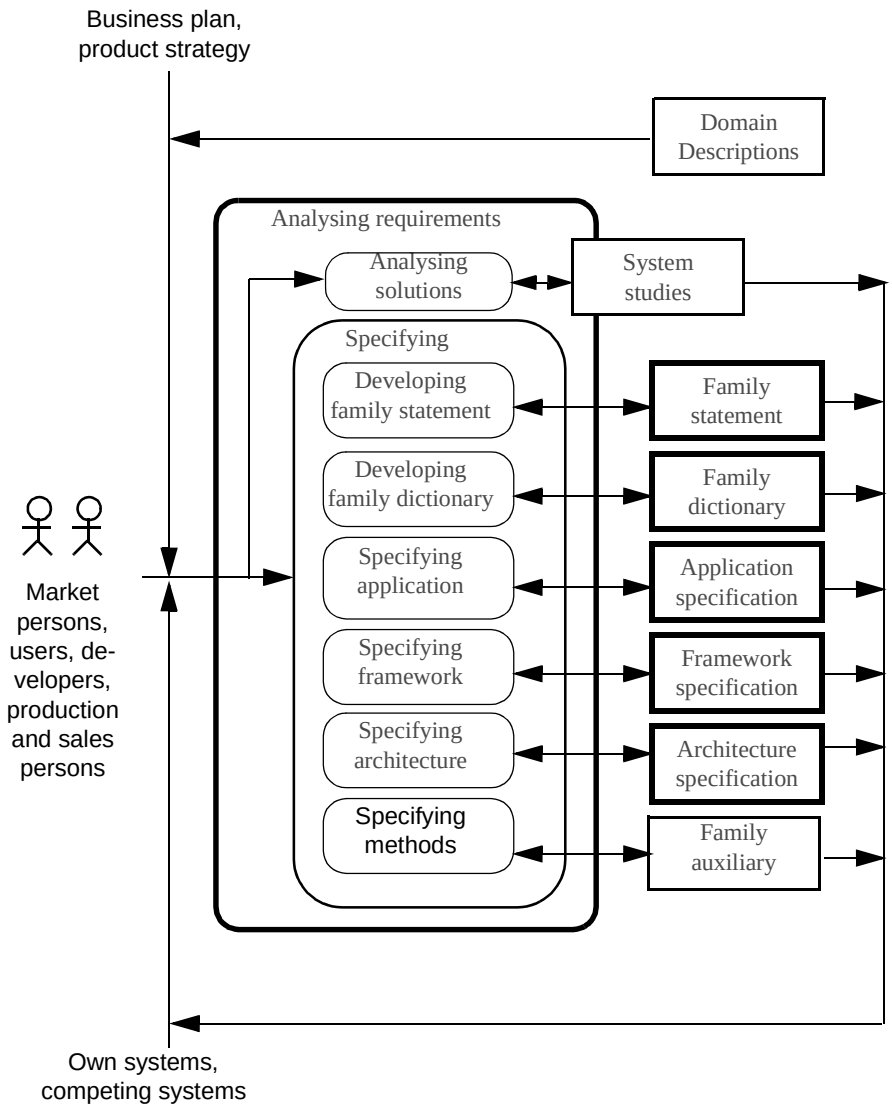
Requirements analysis

This activity produces requirement *specifications* in terms of *context* specifications in UML or SDL (depending on the desired degree of formality and on the starting point) eventually supplemented by *content* specification where this is known and needed in order to fully specify requirements. Corresponding *property* specifications are produced.

The activity will specify the properties of the systems down to a level where the system can be evaluated and compared to other possible solutions. It studies different system alternatives, and it makes requirements on how systems shall be instantiated and how they may evolve.

Figure 4-23: Analysing requirements

Open figure



In addition it updates the Dictionary (p.4-30) and Domain statement (p.4-30) from the domain analysis with elements that have to do with the introduction of a specific system (or family of systems) in this domain.

Central to this activity is the notion of specification, defined thus:

- A specification covers those aspects of a model that are relevant for its external representation and use. The context part is often sufficient as a specification, but if parts of the content are important it may be included in the specification. Specifications are associated with the abstractions they belong to.

Requirements specification

A requirements specification is a document which is normally produced early in a development project and used as a contract for the design work. It will contain specifications and other items of relevance at that stage. After delivery we are interested in the provided properties (i.e. specifications) of the system, and are not interested in the historic document. TIMe unites these two aspects in the single term *specification*.

Specifications vs. design

Specifications contain the specification parts of Application, Framework and Architecture models (see Figure "Context/content").

Systems are part of an actual use environment

Requirements are requirements to systems. Systems within the same domain will have in common that they handle the same kinds of phenomena. All systems within the access control domain will handle access zones, access points, and users that want to get access to access zones.

A specific system may in addition have properties that are needed because the system will have other categories of users, e.g. operators that have other requirements to the system, or an owner that e.g. wants statistics on the traffic. A specific system may also have interfaces to other systems in the environment.

Guidelines for requirements analysis

- Make a context diagram with the system as focus and the system environment detailed. Only show parts of the environment that are related to the system.
- Sketch or outline the system structure using UML.
- Identify the parts that are subject to requirements.
- Use open aggregation to illustrate how entities in the environment relates and are connected with parts of the system.
- Define the interface behaviour of each role in the system and in the environment.

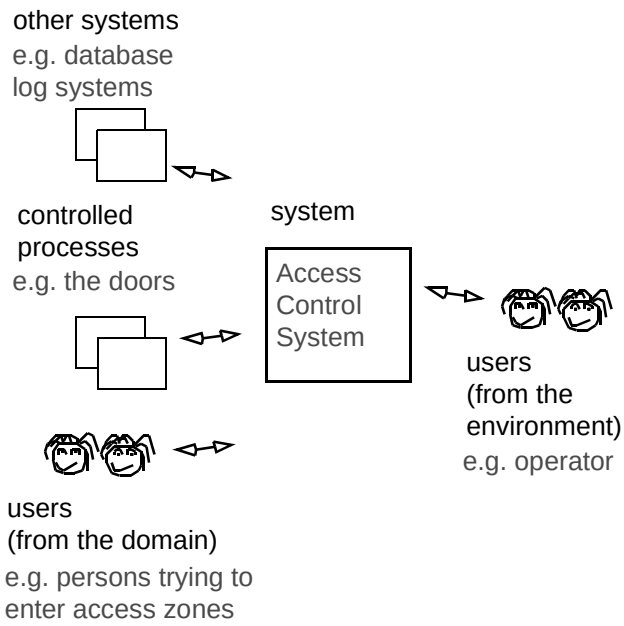


Figure 4-24: System and its environment

When an access control system is considered in its use environment, then there will both be users from the domain and users of the actual system (operator). The “domain users” will have their requirements to the system (and interface at the access points), while the operators will have other requirements and have access to quite other parts and quite different properties of the system.

It is also decided that doors are outside the system, and as such will be processes that shall be controlled by the system.

Application specification

When analysing and designing a system within a given domain, the domain models will be of less use if the method does provide guidelines on how they contribute to the system design. In addition to the properties identified as part of domain analysis, there will be required properties that are specific for this system in its use environment. It is an experience that interface properties should be treated separately.

TIMe therefore has a system reference model, where these three aspects are treated as separate issues and contribute differently to the system design, see Figure 4-25 (p.4-41).

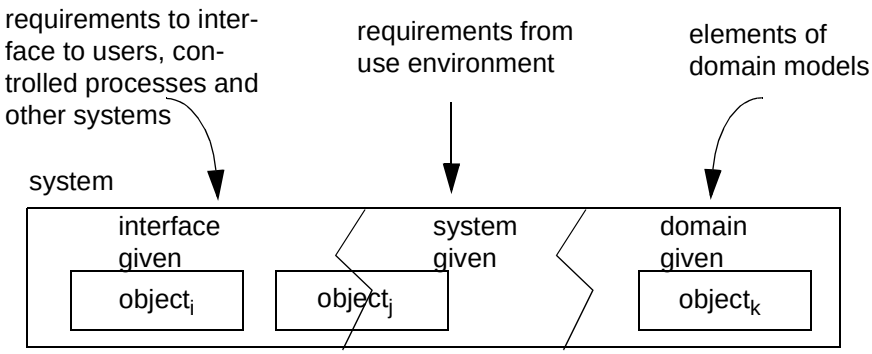


Figure 4-25: Contributions to the different aspects of a system

The domain models contribute to the domain given aspects of the system. These aspects will be more stable than the interface and system given aspects, and the domain given classes used for design will have a greater potential than the other classes for being (re)used in other systems in the same domain. That's the motivation for this distinction.

It is recommended to use:

- UML with the system represented by a central class and connections represented by special relations, or
- SDL with the system represented by a block type with gates (see Figure 4-26 (p.4-41)).

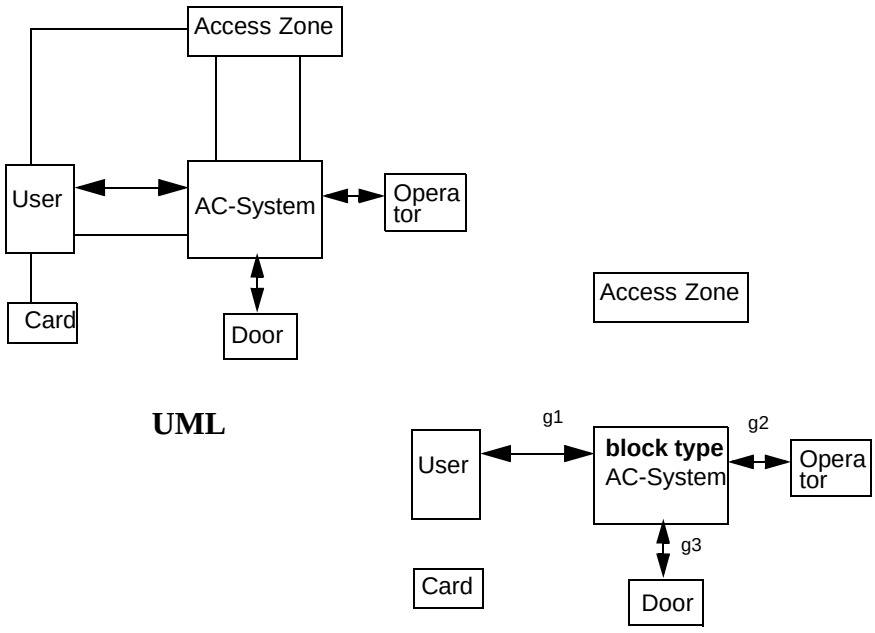


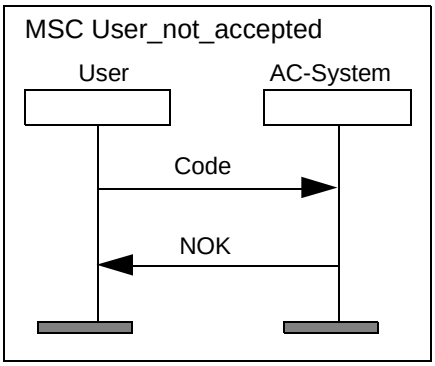
Figure 4-26: Context models

The choice depends on how close to UML one desires to be or how formal the context specification shall be. If SDL is chosen, then really only the connections can be shown, while UML can also show the relations (connections are special relations).

Such context models are matched with corresponding Use Cases (also called Interaction Scenarios) in MSC, where each connection corresponds to one or more MSC diagrams (see Figure 4-27 (p.4-42) and Figure 4-28 (p.4-42)).

Figure 4-27: Property model from domain: MSC User_not_accepted by system

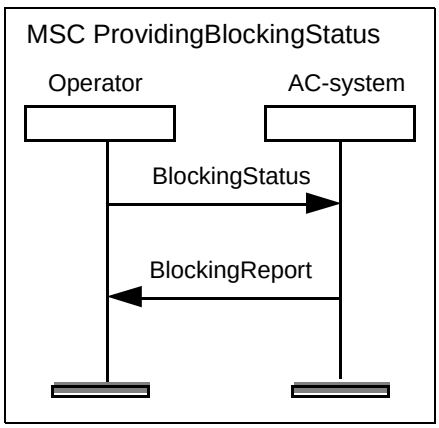
Open figure



One of the Use Cases between the domain given User and the system is the one where a user is not accepted, because the code is not OK. During system analysis it is decided that this shall be performed by the AC-system.

Figure 4-28: System specific property: Blocking Status provided by system and initiated by Operator

Open figure

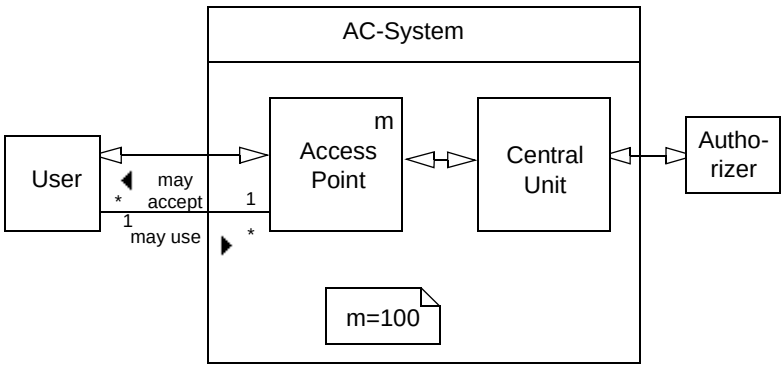


From the actual use environment we know that the systems will have operators, and that they will ask for the blocking status of access points. This was not part of the domain model.

Requirements analysis produces a (requirements) specification for the system to be designed. It may be so that the system has an *inherent* structure, and that this has to specified in order to get the specification right. In that case, the specification includes a structuring of the system by means of “real aggregation”, and the environment communicates with the parts of the system (see Figure 4-29 (p.4-43)).

Figure 4-29: System Context/Design Outline

Open figure



From the Domain Object Model we get the Access-Point object. The system will provide its services at a number of Access Points, but validation shall be centralised, i.e. there will be a central unit that takes care of the validation, while the access points are merely User servers.

This is reflected in a system context diagram where the system object contains a set of AccessPoint objects and one CentralUnit object.

When specifications include a structuring like this, the corresponding property models must be changed correspondingly, so that User does not only interact with AC-system but with AccessPoint, and Operator not with AC-System but with CentralUnit.

Application Specification is a crucial part of the method. The following lists the recommended activities and guidelines for this part, some of which have been illustrated above:

- Decide on what parts of the domain that shall be inside the system and what parts shall be in the environment, and what shall not be considered at all.
- Represent the system type as one entity, and show its interconnections to entity sets in the environment. Specify constraints and variability of the entity sets.
- Make a (passive) object model representing the entities in the environment that the system family shall know.
- Describe the domain specific services in terms of service lists, role diagrams and Use Cases (in MSC). For each of the active object types in the environment, make a context diagram and describe its active environment in terms of association roles. Make a function list and specify the corresponding service behaviour using roles and MSC. If possible or relevant, describe association role behaviours as completely as possible.
- Consider the system specific parts. Identify any system specific services and system specific objects (active or passive) that are needed.
- Add system specific entities to the active and passive environment.

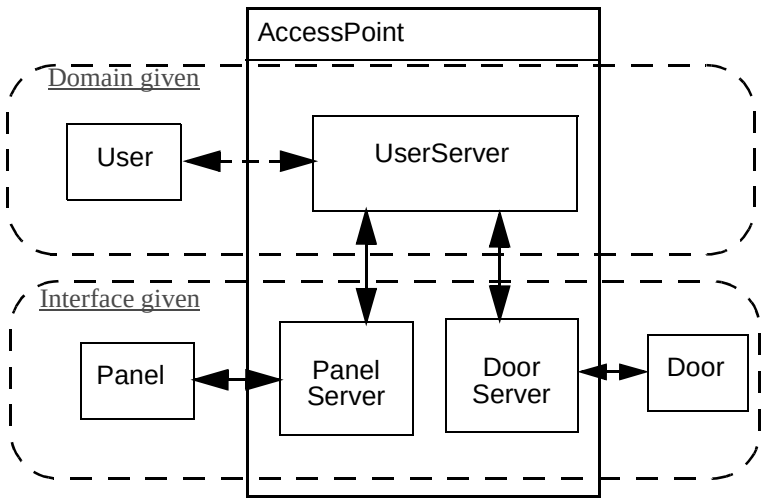
System analysis may also consider the interface specific properties and specify corresponding context/content models. When the system specification has included parts of the system (as with AccessPoint and CentralUnit in Figure 4-29 (p.4-43)), then the interface specification may take that into account. In Figure 4-30 (p.4-45) it has been decided that the interface of AccessPoint shall be to a panel and to a door, and the corresponding objects of AccessPoint have been identified.

Considering interface aspects in system analysis:

It is decided that the code shall be entered through a panel and that the resulting response (OK or NotOK) shall be presented at the same panel. Correspondingly, the actual controlling of the door is singled out as an interface specific part of AccessPoint.

Figure 4-30: Introducing PanelServer and DoorServer as part of AccessPoint

Open figure



Architecture specification

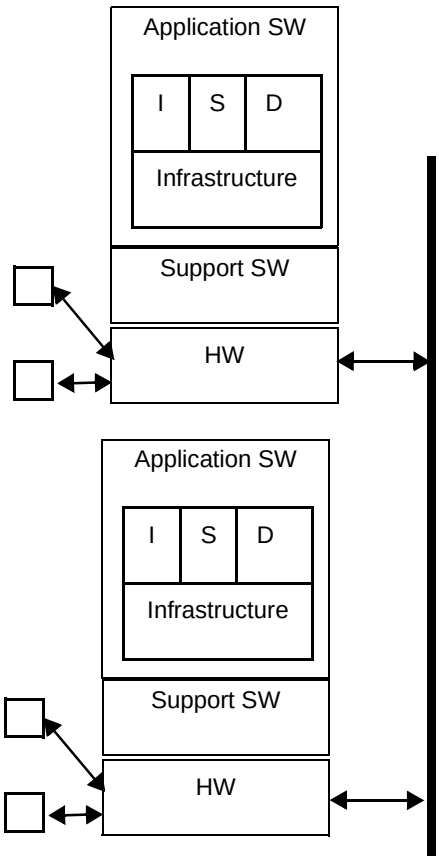
In addition to considering the application specific properties of a system, system analysis may also take requirements on *Platform* into consideration and specify these. Platform has to do with non-functional requirements to the implementation, e.g. the choice of technology, implementation principles, etc.

While the application specification is an abstract description which does not take physical aspects into account, the Implementation is considered as a concrete description. A central idea in the methodology is to describe abstract systems in a way that can be understood and validated without knowing how they are implemented.

The concrete description is composed from real hardware and executable software. The concrete system will have an Application part where we find the implementation of the abstract system, and a support part containing additional functionality needed to execute the application. It will often be distributed and have additional support for internal communication, see Figure 4-31 (p.4-46).

Figure 4-31: Concrete system reference model

Open figure



For the access control system, Architecture Specification amounts to specifying e.g. that plastic cards shall be the means for identification, and that implementation code for the software parts shall be generated from SDL designs and based on an existing runtime system.

Framework/Infrastructure specification

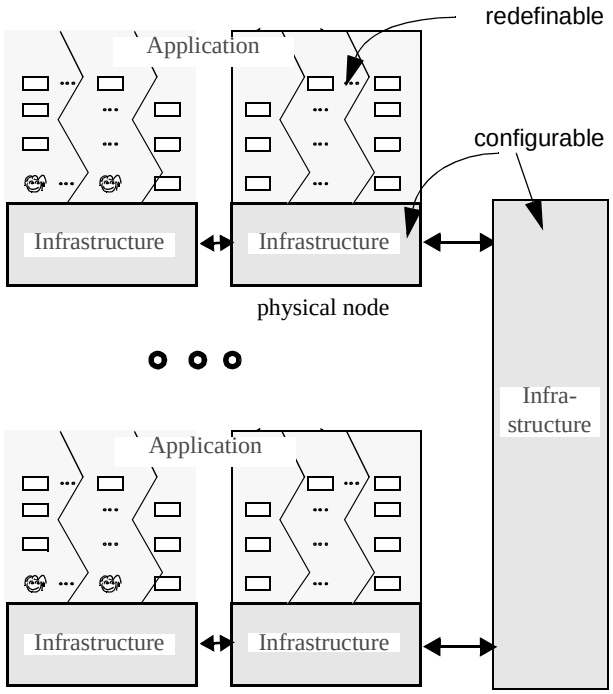
Consideration of issues like distribution, systems management, etc., that is behaviour that has to be part of the system but does not contribute to the services it provides, produces the *Infrastructure* specification.

The application Framework is an abstract system which takes into account concrete system issues such as distribution and error handling. It consists of a distributed Application part, and an Infrastructure part.

TIMe recommends developing a refined and restructured, complete functional specification reflecting the concrete system and the implementation dependent requirements, and turning this into a Framework specification, as illustrated in Figure 4-32 (p.4-47).

Figure 4-32: Application framework reference model

Open figure



If the Infrastructure specification can be constructed so that it forms a *Framework* for systems with the same Infrastructure, but with varying Application part, then this is done. The application specification is changed accordingly. The idea is that if a system can be made as an instance of a Framework, with much of the general properties of the Framework isolated in the Infrastructure, then the Framework will have a potential for being reused as a design.

In Object and Property Models - and the Languages for describing them we illustrate the access control system where the Infrastructure and Platform issues have been considered.

For the access control system the fact that validation shall be performed by central computer is an infrastructure issue, like the possibility of distribution of validation to the access points, with additional protocols as an implication.

Table 4-2: Application, framework and architecture aspects for the access control system

application specification	framework/ infrastructure specification	architecture specification
• system object, possibly containing accesspoint and centralunit, with context specification including signals like Code, OK, NotOK, and MSCs as a specification of use cases.	• Validation shall be performed by central computer. • Possibly distribution of validation to the accesspoints, with additional protocols as an implication	• Plastic cards as the means for identification. • Code for the software parts shall be generated from SDL designs and based on existing runtime system for the central computer and tailored run time systems for the code in the access points.

In an initial development the infrastructure aspect may not be obvious. Frameworks will often come as a result of a (successful) initial development, which is to be used as a basis for a new system. If e.g. distribution has been considered and isolated in an infrastructure part, the next system with the same infrastructure but with a different application part can reuse this framework.

Design

Design object models in SDL

This activity produces design object models primarily in SDL. Some parts of the design have to do with the required properties (*Application* design), another part of the design has to do with *Architecture* specific issues (including non-functional properties in contrast to the functional properties of application design), and a third part combines these two into a *Framework* for instantiation of specific systems with the same infrastructure.

Design is a creative process. One thing is that the system design model will be in SDL, while analysis models may be in UML. Another thing is that design may require a restructuring, and will certainly add details and precision.

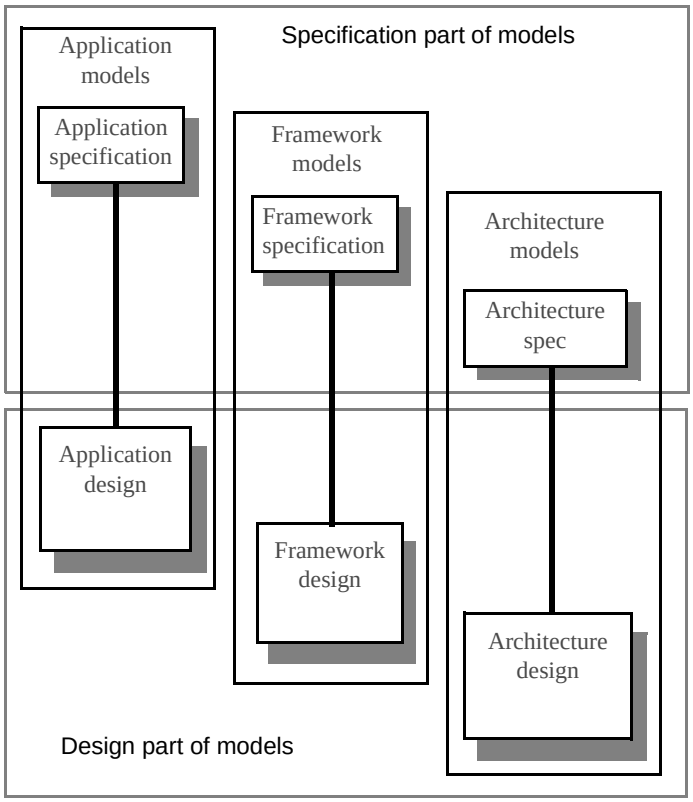
TIMe contains guidelines on how to transform UML models into SDL models, and these are more or less automatic. It is a point, however, that they are not quite automatic - if they were there would be no need for the UML models (or for the SDL models). The

interesting transformation is to take the system requirements, identify the system objects and assign attributes and behaviour to these object so that required properties are fulfilled.

Specifications contain the specification parts of Application, Framework and Architecture models. These are related to the design parts, as indicated in Figure 4-33 (p.4-49).

Figure 4-33: Specification and design related

Open figure



The main design language is SDL, but in cases where the system will be a combination of SDL components and components created e.g. from an UML model, or using an interface construction tool, the main design may be in UML.

There is some help to get in this main part of design activity. As mentioned above, the system analysis produces specifications on three levels, and the system design follows these specification levels:

- Application Design: where the real functionality is designed (p.4-49)
- Framework Design: from Infrastructure to Framework (p.4-59)
- Architecture Design: choice of implementation platform (p.4-58).

Application Design: where the real functionality is designed

Application Design produces context and content designs (in terms of structure and behaviour) for the system type and/or for types being used in the system:

- *Application context,* that is a context model for the type, i.e. a diagram with the type as a single entity. It specifies the environment, the interfaces and the knowledge of the type as well as external types which are used as components. It also specifies the context properties, i.e. services, and describes all objects in the environment.
- *Application structure* applies to types that consist of object aggregates, defining the content as a structure of components.
- *Application behaviour* applies to types that have a behaviour of their own, e.g. SDL processes.

The first purpose of an application design model is to describe the system behaviour at an abstraction level, where it can be understood and analyzed independently of a particular implementation. This is done in terms of both an object and a property model.

The second purpose is to be a firm foundation for designing an optimum implementation satisfying both the functional and non-functional requirements.

Application design starts from the application context and the required properties. New objects may be introduced during design, and these are also subject to the context/content distinction.

The application content may introduce new component types. In general the component types and application types are designed in the same way:

- context design;
- content design: this is either behaviour design or content design.

From domain objects to design objects

As mentioned above, some domain objects are candidates for design objects. In Figure 4-34 (p.4-50) it is indicated that AccessPoint may become a block type in the SDL design.

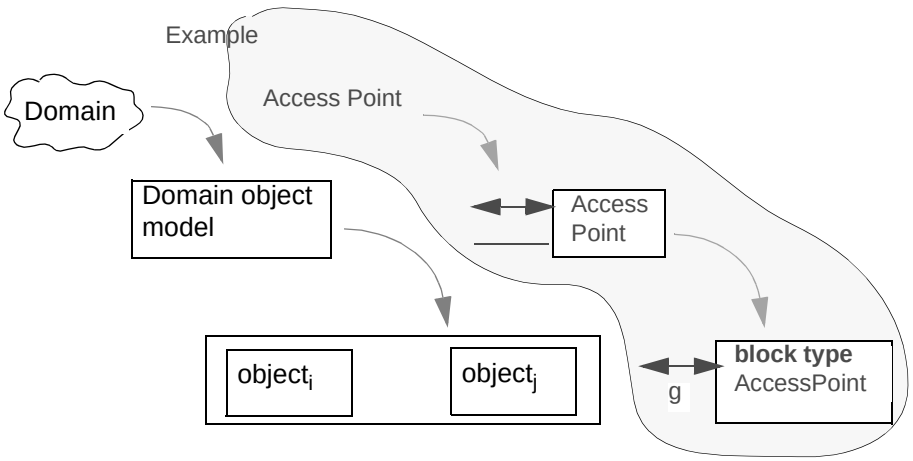


Figure 4-34: From domain objects to design objects

Another source of design objects comes from mirroring the entities in the environment of the system. Considering the system specific aspects or properties will either add new classes to the set of classes from the domain object model, it will add system specific properties to domain object model classes, or it will make new subclasses to the domain model classes. This may give the application specific objects two aspects: *domain* given

and *system* given. The pure domain model classes can be used in all systems in the domain, while the system specific (sub)classes can only be used in systems with properties specific for this family of systems.

- "Is this not making things complicated? We want all classes we make to be general enough to be used in other systems anyway, so why not just get started and make some classes!"
- Says TIMe: "It is a common misunderstanding that all classes are equally (or a priori) reusable, while the fact is that many classes defined for the purpose of a system are defined within that context and will only work in that context. It takes a lot to make a class generally usable. The TIMe recommendation is therefore that objects (and their classes) are mainly used for the purpose of structuring systems, and that classes in the first place should be defined with this purpose in mind. So, when making a domain object model, include only the obvious general objects and the obvious general properties - it is no mistake if the domain model starts out being small".

Subsystems or not, and when

In some cases it is obvious that the system shall be decomposed into subsystems, or that objects in the system have a content structure. In that case this is directly supported by the SDL *block* concept. An SDL system simply consists of a number of blocks connected by communication paths, so-called *channels*, and the blocks may in turn either contain a new substructure of blocks or sets of *processes*.

TIMe establishes rules for good subsystem design that are readily supported by SDL. Subsystems may either come as reflecting an inherent structure of the system, as e.g. the division into central unit and a number of access points, or they may come from a pure functional decomposition.

TIMe advocates to start the subsystem decomposition from an inherent structure and then introduce new subsystems if it turns out that required properties cannot be obtained by assigning behaviour to already identified subsystems.

Inspiration from the environment

"Finding" the content objects may in some cases appear as "magic" and may require some experience from good design for similar systems. However, once the environment is well defined, the task is simpler. With a slight adaptation of an old saying: *Tell me who is in your environment and I will tell you who you are (i.e. what your content is).*

In addition to guidelines like this, the complete TIMe contains rules for good design in SDL (e.g. when to use concurrent processes, purpose of block substructuring, redesigning by generalisation, etc.).

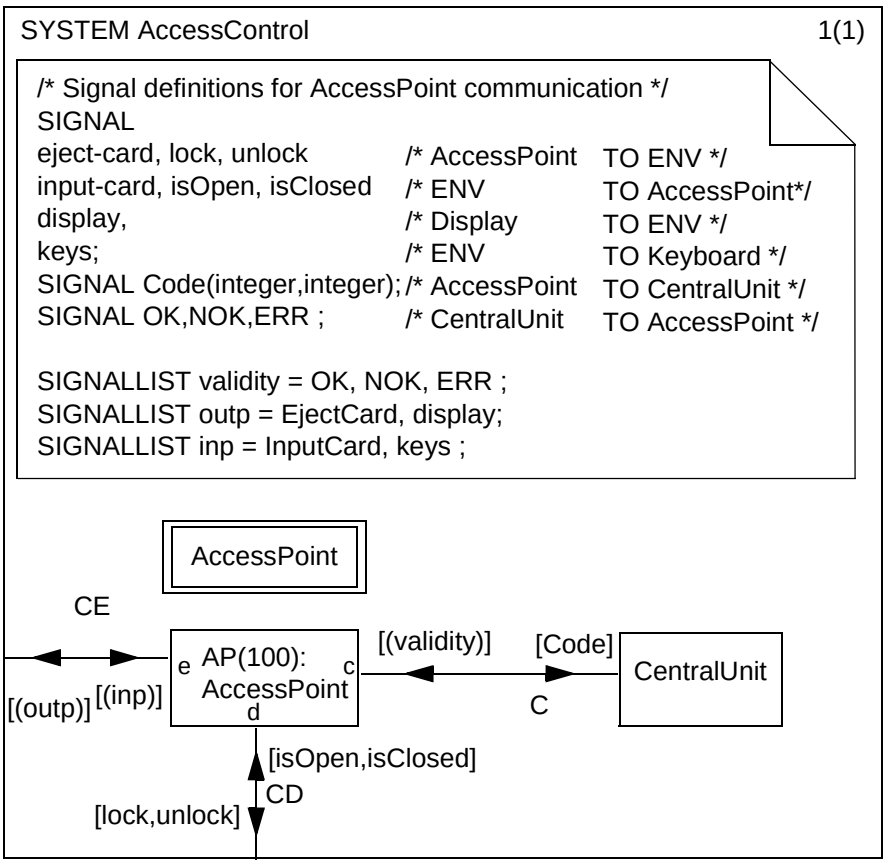
Try this sequence of activities to ensure that all roles supposed to be played by the system are provided by some objects in the system:

- Mirror the environment behaviour: *Identify the objects in the environment, and describe the corresponding types with association roles. For each association role directly interacting with an environment object through a static one-to-one connection, assign an actor object in the system.*
- *Define the corresponding object types and their association roles.*
- *If possible, assign the association roles remaining to be bound to objects already defined, otherwise introduce new objects.*
- *Introduce switched communication where n-to-m communication is needed.*
- *Continue until all roles have been bound to actors.* This may be an iterative process by which new actor objects are found.
- During these activities, make MSCs detailing the internal interactions (between the newly design objects) and check that the structure will give effective behaviour definitions.

From the system analysis we get the specification that the system shall be structured into a set of AccessPoints and a CentralUnit. We also know that AccessPoint shall both handle the panel, the door and communication with the CentralUnit (that is three processes), so we decide to have AccessPoint as blocks, because blocks may contain processes.

Figure 4-35: Application design in SDL

Open figure



Object (behaviour) design

If system content decomposition in terms of subsystems is not obvious, TIMe advocates to design the object types first. That is identify the *attributes* and *behaviour* that each object shall have in order to fulfill the required properties.

At this point **Sesam Sesam** had consulted some UML experts. They had already made a domain object model (in UML) and used this as a basis for a first system object model. They were now in the position to do what really is the core of the development: to specify the behaviour of the objects so that they together provide the required properties. They had parts of the properties defined by use cases and now they wanted to specify the behaviour of the objects.

An obvious choice was to generate skeleton code from the UML object model and then provide the functionality in C++ or Java, but problems were reported to the TIMe consultant:

- "Some of these objects have intricate behaviour and a lot of interaction, so we wanted to specify them as state machines where the transitions are triggered by incoming signals"
- "I guess you have used the Statecharts notation in UML" - says the TIMe consultant.
- "Yes, but we also wanted code generation from the behaviour specification, and that is not supported - the object model and the behaviour model are not integrated".
- "Ah" - says the TIMe consultant - "then you are really looking for SDL: most of the UML object model can be represented in SDL (except general associations, but aggregation and inheritance are supported). Expressing the behaviour specified in terms of Extended Finite State Machines is an integrated part of SDL."

For design in SDL, object behaviour design amounts to identifying the required processes and specifying these by means of variables, procedures and behaviour in terms of states and transitions. The context design of the class leads to gate definitions in the corresponding SDL type, while the property models are input to the combined behaviour of the process.

In the object design, property models can be made more detailed and precise.

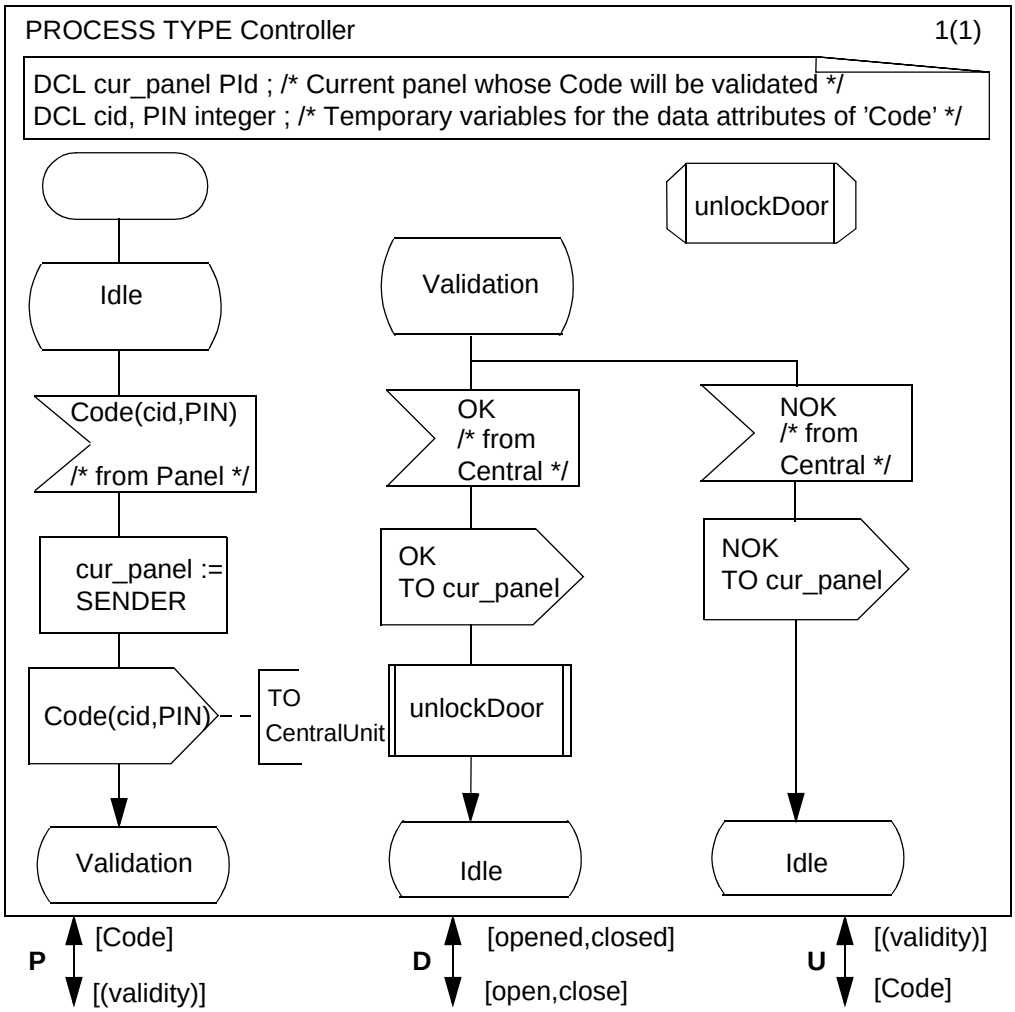
TIMe contains guidelines for how to come from a set of property models in terms of MSC to the corresponding process type in SDL. A short description of these guidelines are found in From MSC Property Models to SDL Object Models.

In the first round it is recommended to ignore the interface specific behaviour. We know that AccessPoint will have a part that handles the user without considering how the card code and the PIN are entered via the panel (UserServer in Figure 4-30 "Introducing PanelServer and DoorServer as part of AccessPoint" (p.4-45)). From the MSCs between

system objects we know that this part of AccessPoint shall also handle the communication with the CentralUnit, so we rename “UserServer” to the more neutral “Controller”, see Figure 4-36 (p.4-55).

Figure 4-36: Behaviour of Controller according to User Accepted & User Not Accepted

Open figure

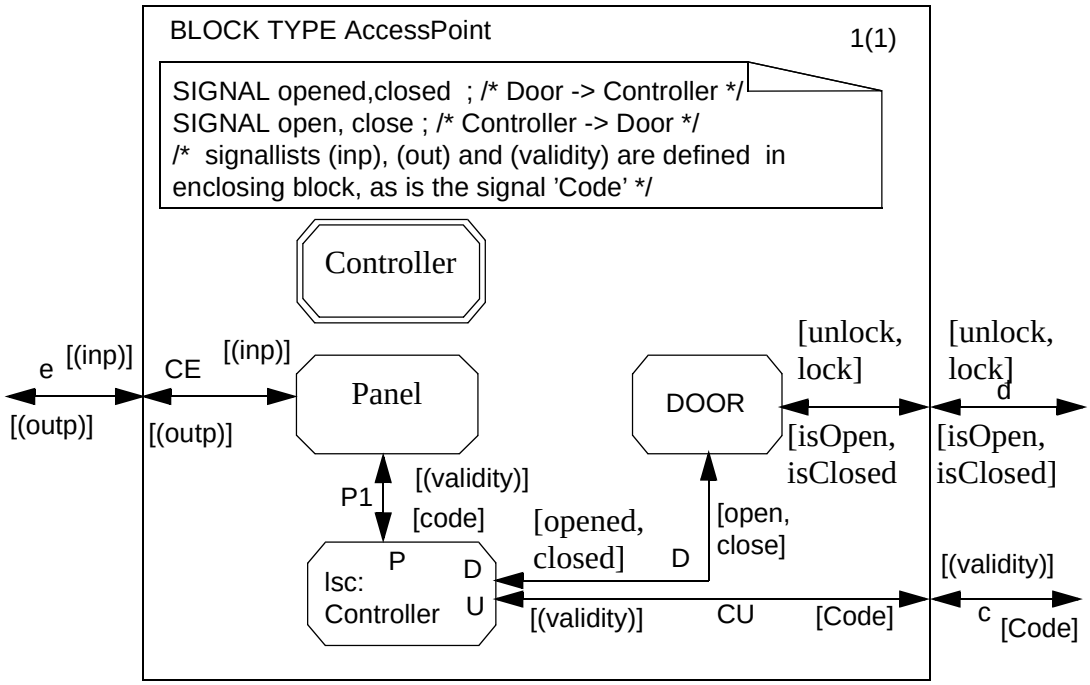


Access points shall handle the use cases where a user enters a code and gets either Ok or NotOK. The Controller object as part of AccessPoint shall provide the required properties. This leads to the process type above. It is made independently of how Code is obtained from the user and how OK and NotOK are presented (interface specific). It has been decided that the validation shall be done by the central unit. If we did not want to take this decision at this point, we could have made a transition that simply (informally) provided the validation, and then later changed this to a communication with the central unit.

This process type fits into a design of the AccessControl object as a block (defined by a block type) as in Figure 4-37 (p.4-56).

Figure 4-37: Block type AccessPoint with processes

Open figure



Designing non-domain given objects

Should all required properties lead to attributes and behaviour of the domain given objects? The answer is no!

The **Sesam Sesam** group had been successful in starting out with domain objects in order to get at the application objects, but ...

- "This is all very nice, but how do we introduce the new system aspects?"
- Says TIMe: "As mentioned before there are mainly two ways out: either introduce new system objects, or introduce specialisations of the domain object classes. If you have an UML domain object model and generate code from this, your extra classes or your specialisations can be done either in UML or directly in C++ or Java. If you have turned to SDL, then you have part of your domain object model represented as types in SDL, and you make new types or subtypes"

The main purpose of the distinction between domain and system given aspects is that special services should not be associated with domain objects, as these will probably not be of interest to other systems in the domain. Besides working as inspirations for application objects, domain object classes are candidates for re-use in different systems in the same domain.

Which objects should then provide a property that is not obviously covered by a domain given object?

The answer is:

- either a separate object,

- or an object of a subclass of the corresponding domain object class.

The last alternative requires that the domain object class is represented also in the design language, and it is recommended to document that the subclass is system given and not domain given.

"What about the introduction of the interface specific aspects? Does that follow the same pattern: either special	objects or specialisations of the domain object classes?"

If it is an interface property, and if the property has to do with the actual appearance or implementation of the interface, then it should be provided by a separate interface object, like the Panel process in Figure 4-37 (p.4-56). Low-level interface (protocols) or the window part of a user interface should be isolated in special objects, while interface given behaviour at the "application" level can be provided by specialisations of domain classes. The main thing is to isolate the objects that may change with change of underlying technology. The answer can also be given by how the interface is to be provided (existing protocol implementations, user interface toolkits).

If it is a system given property and if it requires e.g. a separate computation or interaction with other non-domain given objects, then it should be provided by a separate system object. An example of this is the operator handling object. It should be defined as a separate object, but its class may e.g. be a specialization of a class that exists, e.g. AccessPoint.

It may be tempting to take each use case and make a kind of "control" object that takes care of this use case - then it will at least be easy to trace it when considering new requirements related to the use case. Most often, however, the instances in the MSC diagrams for the use cases only represent Roles or partial behaviour of some role. The challenge in design is rather to distribute the required behaviour to objects, and objects will often play several roles.

Evolution of domain models, including design issues

The distinction between domain, system and interface given aspects may change over time. The domain may be narrowed to include some of the other aspects, and the classes of the domain models may include more and more of the properties that appear to be common for many systems.

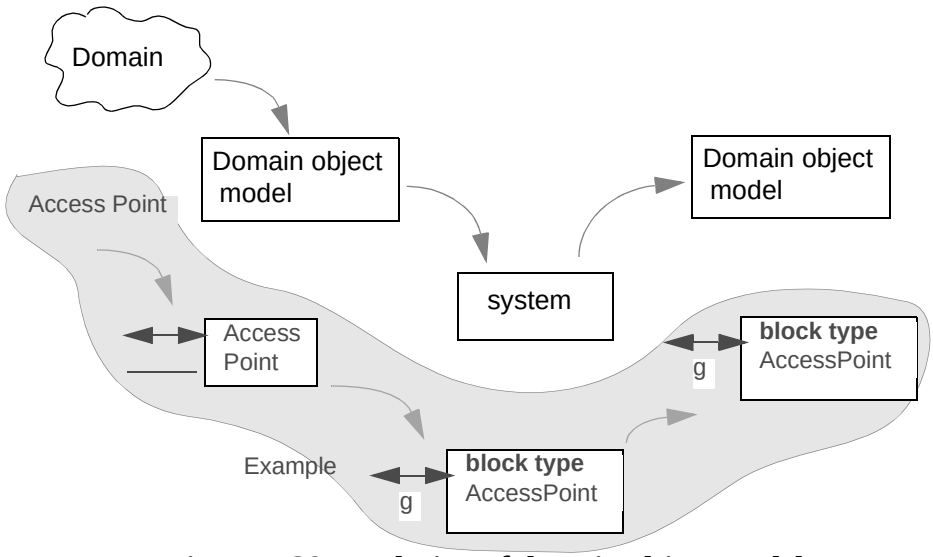


Figure 4-38: Evolution of domain object model

Locally defined types or types in packages?

Where shall the application given design types be defined? A priori they are defined as part of a system model.

Types may be defined in SDL packages and used by the system model. Such packages may either be system specific or more general.

Types in the latter kind of package will have to be more general than in the first kind of package, as they shall be usable in more than one system. As a starting point, design types are defined as part of the system and shall at least fulfil their “mission” there. In addition, it is recommended that types are turned into general types that can be used in other systems.

TIMe provides guidelines on how to achieve generality:

- by *generalization*, that is by defining supertypes with virtual properties for redefinition in subtypes, and
- by *parameterization*, that is types with context parameters, so that types can be fully defined without being in their actual contexts.

Even when defining a general type it is advocated to specify possible requirements on the contexts in which the type can be used. These come from the context models and are readily expressed by gate constraints in SDL types.

Architecture Design: choice of implementation platform

Architecture Design designs an implementation architecture that will behave as defined in the application object model and that satisfies the non-functional properties, taking the actual platform in terms of hardware and support software into account. It will also define a process for (automatic) generation of application implementation code and for configuration and building of system instances.

The purpose of architecture design is to answer *how* the system is going to be realised. This is expressed using Architecture descriptions that show:

- the overall architecture of hardware and software;
- how Frameworks and Applications are mapped to the Architecture.

While the Application and the Framework has focus on functional properties and behaviour, the Architecture has focus on non-functional properties and physical structures. The purpose is to give a unified overview over the implementation and to document the major implementation design decisions.

Architecture design determines critical architectural issues such as physical distribution, global addressing schemes and fault handling. Some of these may subsequently be reflected in the Framework model in order to describe the complete system behaviour.

The Architecture consists of two main parts:

- The Platform, which consists of the hardware with support software (such as the operating system, a DBMS and middleware) and the Infrastructure.
- The Application implementation.

Associated with the architecture it is recommended to define a process for (automatic) generation of code and for configuration and building of system instances.

Architecture design is only performed when the implementation mapping is undefined or needs to be changed. This occurs during the initial development of a system family and during maintenance when changes in the platform are made.

During normal application evolution, the Architecture will stay the same, and system evolution can take place mainly at the Application level.

Hardware and software architectures are defined to a level of detail from which implementation is well defined. The architecture shall separate between support mechanisms, such as an operating systems, and applications.

In an initial development, Architecture design will come before Framework/Infrastructure design. Architecture design involves the choice of implementation platform, what should be done in software and what in hardware, etc. The design may have to be adjusted according to this choice. SDL tools may e.g. impose restrictions in order to support code generation.

TIMe has a 5-step procedure for making architecture design. This is not applied to the example in this overview and is therefore not covered here.

Framework Design: from Infrastructure to Framework

Framework Design defines an abstract and generic framework object model and a method for instantiating the Framework with Applications. In this activity the implementation dependent functionality is taken into account, e.g. distribution support, error handling and configuration. It develops a layered approach which separates the Application and the implementation dependent Infrastructure. The infrastructure part will be nearly complete, and the rules for mapping Applications to the Framework will be well defined.

Making infrastructure

The infrastructure part of a system contains additional behaviour needed to fully understand what the system does (i.e. the complete system behaviour). Here we find objects and parts of objects that support distribution, system administration and other facilities not directly related to user services. Whenever practical the Application and the Infrastructure should be put together in a Framework that serves to simplify the definition of new systems. This implies that the objects that are mainly application specific objects will get some infrastructure specific elements in order to work on the given Infrastructure.

When taking infrastructure aspects into consideration, the system as designed from an application point of view may be redesigned. Restructuring does not mean that everything has to be redefined. A majority of the processes from the first application design may be left unchanged. As they are defined as stand alone types, it is a simple matter to put them into a new structural context together with some new processes.

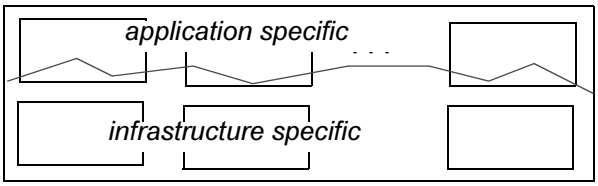
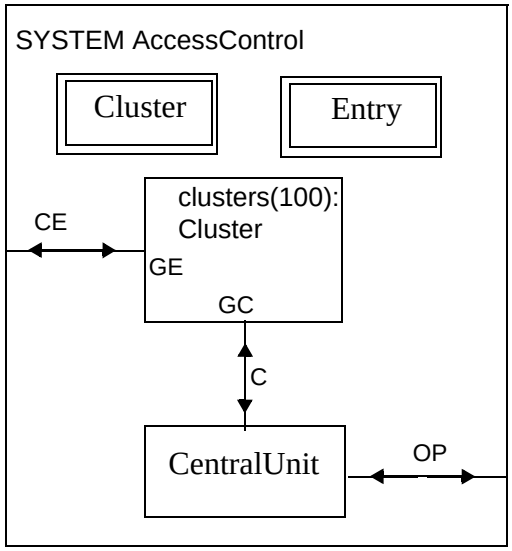


Figure 4-39: Application and infrastructure specific parts of systems into a framework

In general it will be an advantage if the application design has been done by means of types that are as general as possible. General types can be used in more than one context, and when redesigning, the context of the “application” types may change slightly.

Figure 4-40: Redesigned Access Control system V3

Open figure



In the access control system the channels between the AccessPoints and the CentralUnit are candidates for distribution. We therefore decide to let these channels be the ones that cover distances.

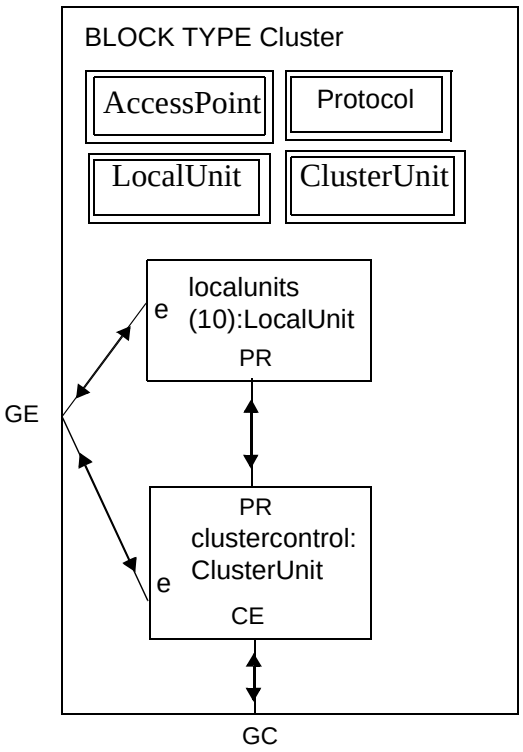
There will be at least one central computer and from zero up to 100 local computers. In this architecture we shall implement the AccessPoint and CentralUnit processes in software running on the computers. We structure the system accordingly: a block set Cluster for the part of the application running on the local computers, and the CentralUnit for the part running on the central computer.

Note that this distributed architecture is different in structure from the application design, and that some communication protocols will be needed to support the communication between the local and central hardware.

With the redesigned system, the application types are possibly modified in order to fit into the new structure. If this has been done, a division of the system into application and infrastructure parts has been obtained, and for the next systems (with the same infrastructure) it is a matter of exchanging the application types with either improved versions or new application types with e.g. new functionality.

Figure 4-41: Cluster with LocalUnits and ClusterUnits

Open figure

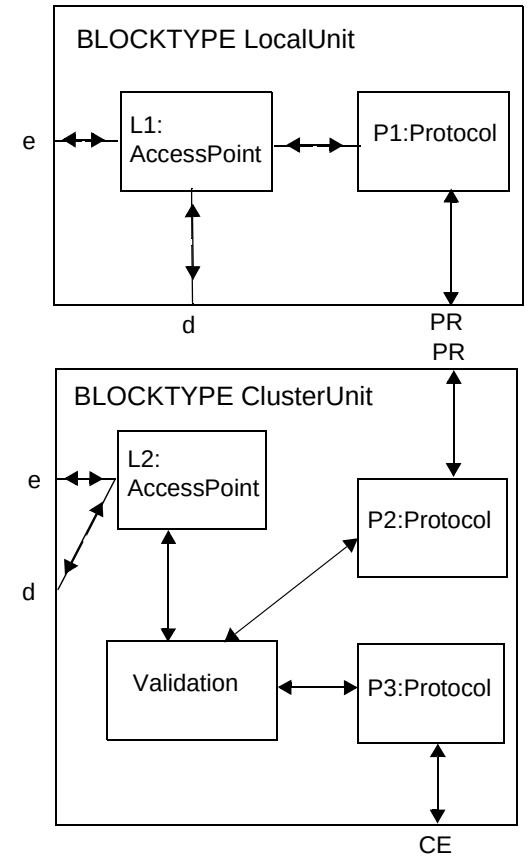


In this solution the validation database will be distributed. There will be a copy of the central Validation process (and its database) in each cluster. This means that the CentralUnit must handle updates in a distributed database. This introduces a new problem to solve in the functional design, but the Access Points and the Validation processes in each cluster may (hopefully) work just as before.

application specific parts	infrastructure specific parts
access point validation	protocol cluster unit local unit

Figure 4-42: AccessPoint used in both LocalUnit and ClusterUnit

Open figure



We see that AccessPoint will be used both in the LocalUnits as well as in the ClusterUnits. Those in the ClusterUnits will have direct, local access to the Validation process, whereas those in the LocalUnits must communicate via physical links and protocols (represented by the block P1 of type Protocol). The signals to and from the AccessPoint blocks will be the same.

application specific parts	infrastructure specific parts
access point validation	protocol cluster unit local unit

Making frameworks

Having identified an infrastructure that seems to be common for many systems with almost the same application properties, TIMe advocates the re-designing of the system into a framework. TIMe gives guidelines on how frameworks can be defined in SDL. The following is a short introduction to how it is done.

As mentioned above, a usual definition of a framework is the following: *"In object oriented systems, a set of classes that embodies an abstract design for solutions to a number of related problems."*

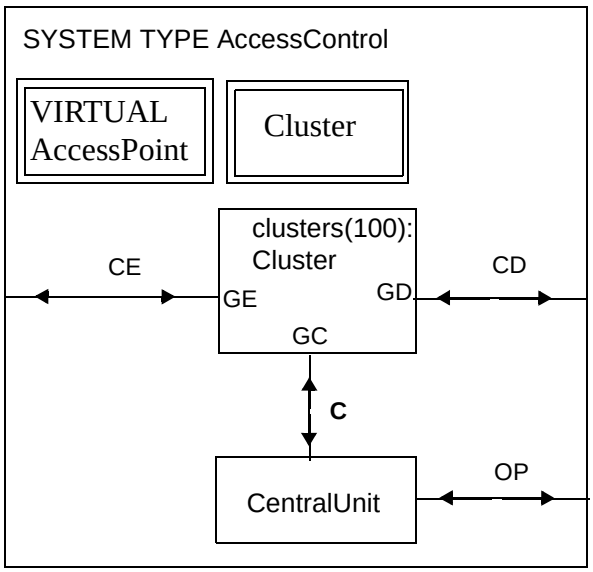
TIMe puts a little more into frameworks than the definition above, and one reason is that SDL can specify the static structure of systems and not just a set of types.

A framework is a class/family of systems, with predefined structure so that a specific system only has to provide the specific "contents" of part of this structure. Frameworks often come about because an abstract (application specific) system description has to be supplemented by a large infrastructure part in order to be executable on a given platform. Instead of making the infrastructure part again for the next system with the same infrastructure on the same platform, a framework that embodies both the application and the infrastructure part is defined. In a framework the infrastructure is stable, while the application part may vary from system to system.

In the infrastructure design (see Making infrastructure (p.4-60)) the infrastructure part consists of the restructuring of the system into cluster units and local unit and the introduction of the protocol units. The application specific part is represented by the block type AccessPoint.

Figure 4-43: Access Control System type as a framework

Open figure

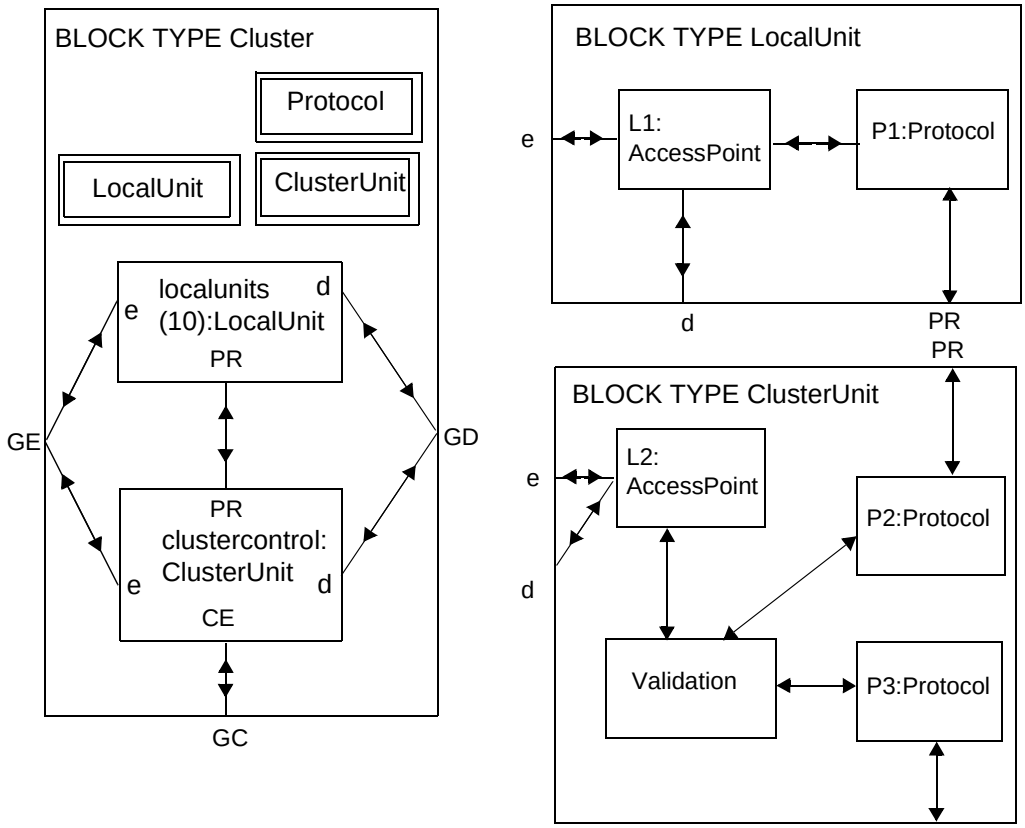


The system description of Figure 4-40 (p.4-61) is turned into a framework simply by defining it as a *system type* and defining the application specific types as *virtual types* (in this case AccessPoint), see Figure 4-43 (p.4-64).

The Cluster block is almost as before: it uses the virtual block type AccessPoint (but it does not contain its definition), and it embodies the infrastructure parts needed for distribution (ClusterUnit, LocalUnit and Protocol), see Figure 4-44 (p.4-65).

Figure 4-44: Block type Cluster as part of framework for Access Control Systems

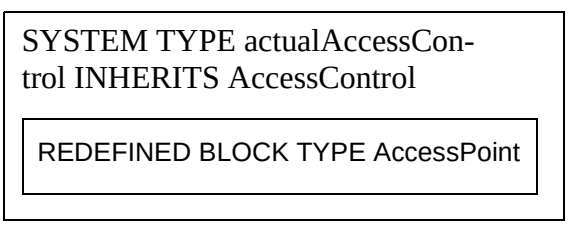
Open figure



An actual system based upon a framework definition is described by defining a subtype of the framework system type, and redefining the virtual, application specific types, see Figure 4-45 (p.4-65). The rules for redefinitions of virtual types in SDL ensures that the redefined AccessPoint will have the same interface as specified in the virtual definition (as a constraint) and thereby assumed by the rest of the system type.

Figure 4-45: An actual system based upon a framework

Open figure



Implementation

Implementations are detailed and precise descriptions of the hardware and the software that a concrete system is made of. They define the physical construction of systems in a family. The software part will be expressed in programming languages such as Java, C++ or Pascal, while the hardware part will be expressed in a mixture of hardware description languages such as circuit diagrams, cabinet layout diagrams and VHDL. Software plays a dual role. Firstly, as a description to be read and understood outside the system, and secondly as an exact prescription of behaviour to be interpreted inside the system.

Concrete system

Concrete systems consist of:

- The Application and the Framework software. State-of-the-art tools allow this software to be automatically derived.
- Special Application and Framework hardware. This will be special hardware designed to perform part of the Application or the Framework.
- The Platform, which consists of:
  - the support software which normally is a layered structure containing operating systems, middleware for distribution support, SDL runtime systems, DBMS and interface support;
  - the general hardware which normally is an network of computers.

What to do

For every new system development, the platform is an important design issue, as it determines important properties such as cost, reliability and flexibility. It also influences the way that Applications and Frameworks are implemented. The code which is generated for the Application and the Framework must be adapted somehow to the Platform. Here the Vendors of code generators use two different strategies. One is to adapt the code generator so the generated code fits the platform. Another is to adapt the generated code to fit different platforms by means of interface modules and/or macros.

Once the platform and the code generation strategy is defined, it is possible to rely on automatic code generation for Application and Framework evolution for those parts where SDL is used.

Instantiation

The main thing in this activity is to configure and to build system instances. Configuration can be applied both to the Application, the Framework, the Architecture, and the Implementation levels. Ideally we should perform configuration at the level where it belongs: functionality at the Application and/or Framework levels, and implementation at the Architecture and/or Implementation levels.

It is possible to perform some configuration at the Application and Framework levels using SDL, but due to limitations in the language, this is restricted.

The common practice in most companies today is therefore to do configuration on the implementation level using configuration files and tools like Make. (An alternative is to use special configuration languages.)

We recommend that a method for configuration and building of system instances is defined as part of the Architecture design work.

Object and Property Models - and the Languages for describing them

Systems in the scope of TIMe are characterised by consisting of concurrently executing objects that communicate by sending signals and whose behaviour is best described by states and transitions (reactive systems).

These systems tend to become large and complex - therefore it is not sufficient to describe the objects - the system also has to be structured in some way. Important properties are often described by use cases and by interactions between objects of the system.

UML [147], OMT [165] and many other methods use object diagrams and informal sketches in the specification and design of structure and a Statecharts-like notation for the specification of behaviour. TIMe uses one language for both: SDL.

SDL is a language recommended by ITU [102] for specifying structure and behaviour of systems that are reactive, concurrent, real-time, distributed and heterogeneous (not just telecommunication systems).

MSC is a notation recommended by ITU [110] for describing interaction scenarios.

Object Modelling

TIMe recognizes that UML and SDL have slightly different approaches to object modeling, that these differences in some cases are beneficial (UML provides e.g. concepts for associations, while SDL does not) and that they in other cases may cause problems. Instead of a clear cut between object modeling in UML and SDL, TIMe defines its underlying approach to object orientation and provides guidelines on how to use both UML and SDL to support this.

This section will give an short introduction to the elements of this underlying approach to object orientation, and then introduce both UML and SDL, describing how they match this approach.

What is object modelling

The approach followed in this method is that an object model is regarded as a *physical model*, simulating the behaviour of either a real or imaginary part of the world. The main property of physical modeling is that it is based upon a conception and understanding of the application domain in terms of *phenomena* and *concepts*, and that physical models will have elements which directly reflect these phenomena and concepts. The physical model will consist of

- objects, that represent the phenomena, and
- classes that represent concepts.

Objects are characterised by variable *attributes* (data attributes), *procedures* (potential behaviour patterns) and *behaviour*. Objects in this approach may execute their behaviour concurrently with other objects. This kind of object is sometimes called “*active* objects” in contrast to “*passive* (data) objects”.

Associated with objects and classes are a number of *structure* and *abstraction* mechanisms:

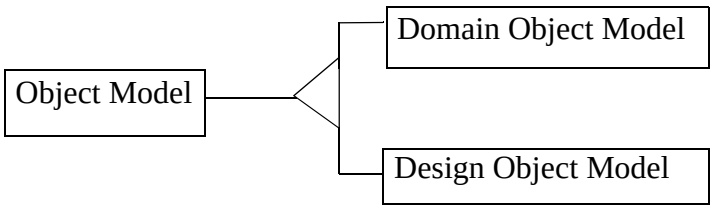
- Identification of objects and the *classification* of these into classes.
- *Part/whole aggregation*, that is objects as part of other objects.
- *Relation* composition, that is an object has relations to other objects instead of having them as parts.
- *Specialization* of classes. Classification relates all objects with the same set of properties into a class. Specialization is a mechanism for the structuring of sets of classes with similar properties into general and specialized classes.
- *Localization* of definitions: Some objects and classes are only meaningful within the context of a specific object or class.

class library

In addition, object oriented languages have support for some kind of library concept, enabling sets of related classes to be used in many different applications.

Domain and design object models

In order to bridge the gap between domain object modeling and design object modeling, TIMe provides guidelines for object modeling in *general*, and *specialized* guidelines for analysis and design.



UML for Object Modelling

TIMe uses UML for describing object models in case the formality of SDL is not required (or desired). The full TIMe book contains a tutorial on UML; the following is just an overview, covering the most important elements.

classes

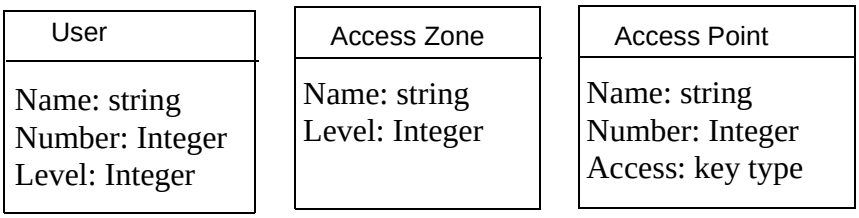
UML object models consist of a set of classes. A class is defined by a class diagram with definition of attributes and operations.

attributes

In Figure 4-46 (p.4-69) three classes are defined with attributes, and no operations.

Figure 4-46: Attribute specification

Open figure



Relations and communication connections

Classes may be related, as e.g. in the domain object model in Figure 4-47 "The access control domain" (p.4-70). AccessPoint and User are *connected* in order to specify that objects of these classes *communicate*.

The endpoints of the relations may have cardinalities.

specialization

Classes may inherit properties from a superclass, as in Figure 4-48 (p.4-70), and thereby define more specialized classes.

Figure 4-47: The access control domain

Open figure

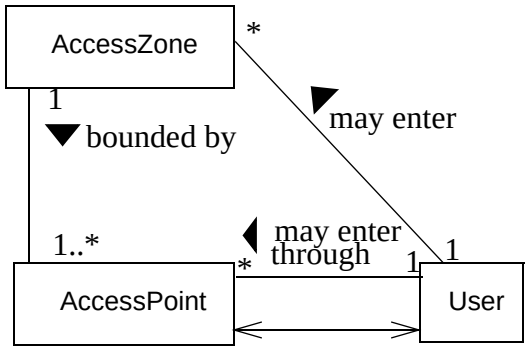
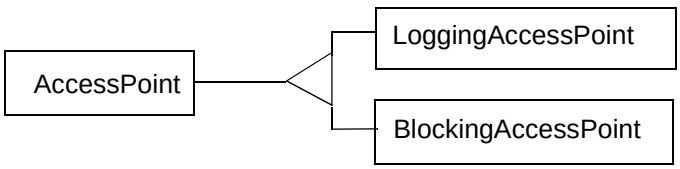


Figure 4-48: Possible classification of Access Points according to logging and blocking functionality

Open figure



Although UML supports multiple inheritance, TIMe advocates the use of single inheritance. One reason is that this is by far the best understood concept - another reason is that SDL only supports single inheritance.

part/whole -real aggregation

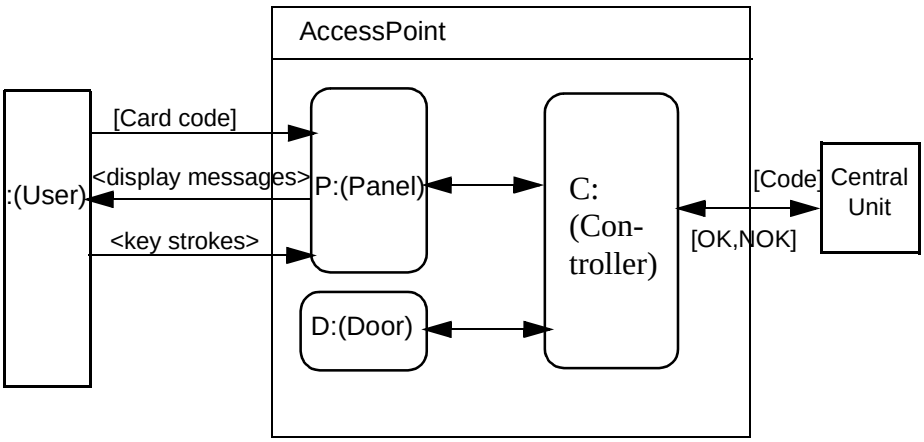
The fact that an object contains other objects is in UML specified by an aggregation association.

In order to really specify that the objects are part of the containing object and that relations to these part objects are only meaningful when contained in this object, the SOON notation [24] can be used, see Figure 4-49 (p.4-71). It is here specified that each AccessPoint consists of three objects (of classes Panel, Door and Controller) and that the

environment communicates with some of these part objects. In UML the User in the environment would have associations to the class Panel in general, while what we want to express is that they only have associations with Panels as part of AccessPoints.

Figure 4-49: Environment entities interact with parts of the system

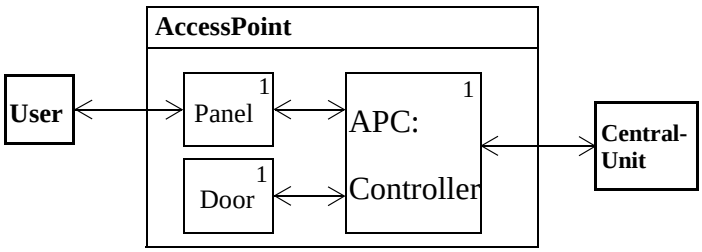
Open figure



The corresponding can be expressed in UML using the Composite relation, preferably using the nested graphical alternative, see Figure 4-50 (p.4-71).

Figure 4-50: Composite aggregation in UML

Open figure



Localiza-tion

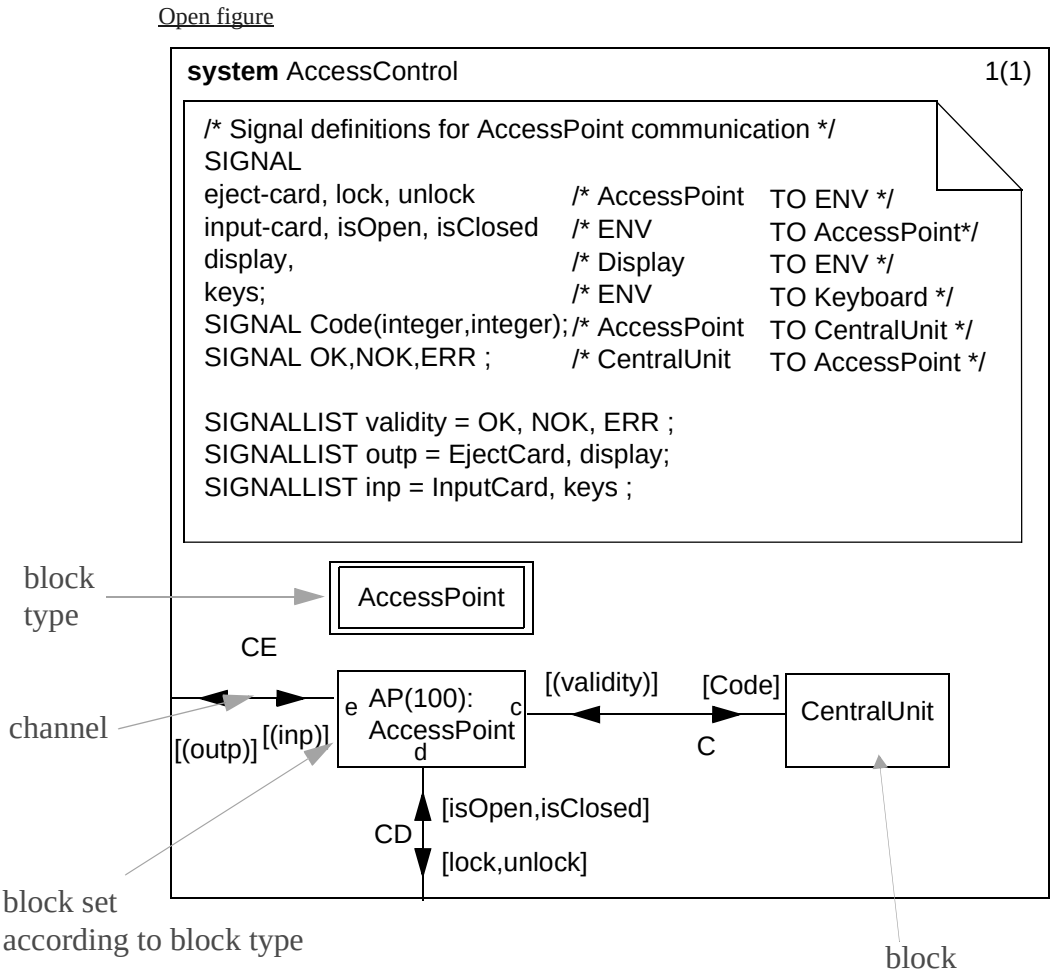
Classes defined locally to classes is not supported by UML. If this is important to express, then it may either be express informally or it may be specified in SDL.

SDL for Structure and Object Behaviour

system

An SDL *system* consist of a number of *blocks,* connected by *channels*. Possible communication by means of *signals* is indicated on the channels.

Figure 4-51: Application design in SDL



The system diagram in Figure 4-35 (p.4-53) defines a system with one block CentralUnit and a set of 100 blocks of block type AccessPoint.

block

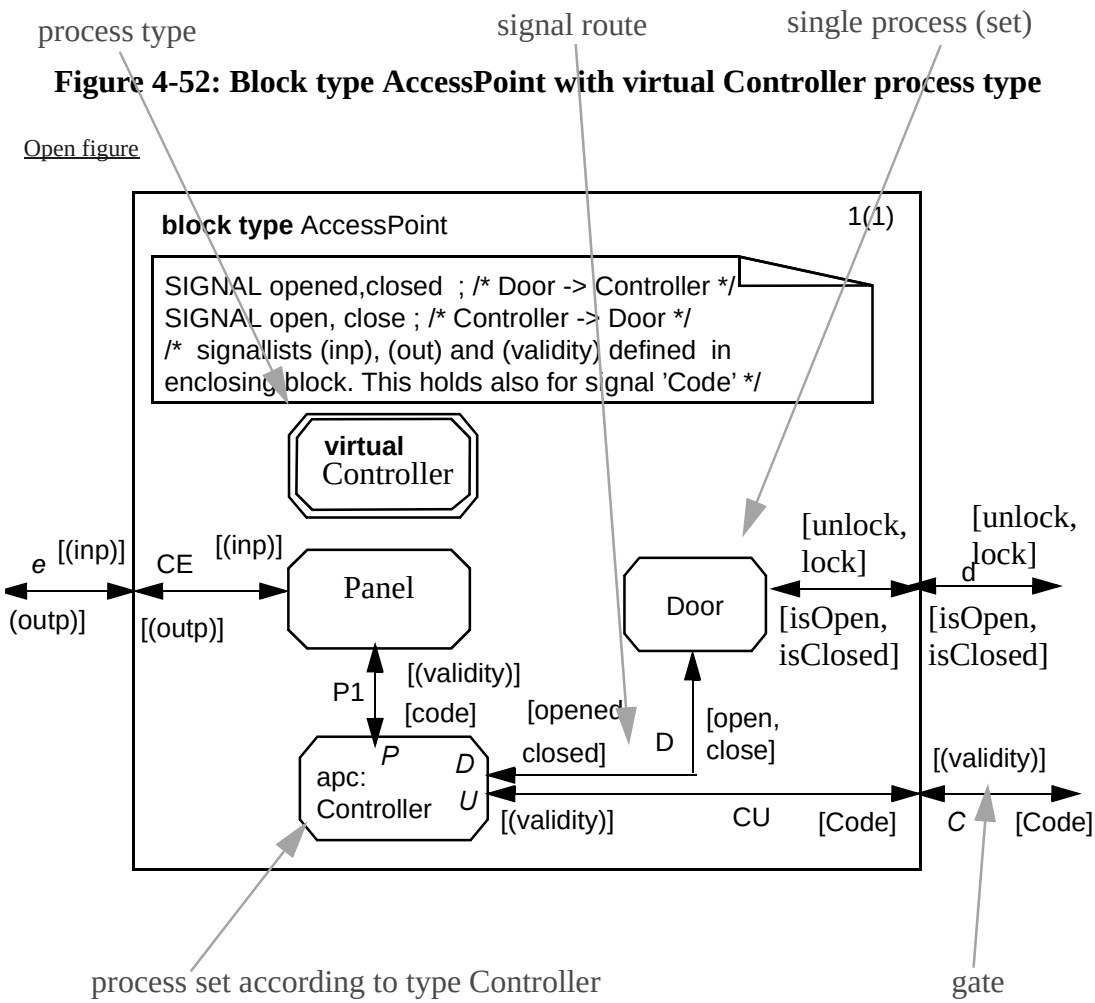
A *block* may either be further *structured* into blocks, or it may contain a number of processes. A *block type* defines a category of blocks with the same properties. The block type diagram in Figure 4-52 (p.4-73) defines the AccessPoint referenced in the system diagram.

Each AccessPoint block will consist of three processes: Panel, Door and apc (access point controller) of *process type* Controller. The fact that the process type controller is defined to be *virtual* implies that it may be redefined in *subtypes* of AccessPoint.

The ***e*** and ***C*** on the outside of the frame are *gates*, that is connection points for channels - they are used in the system diagram above.

"class" The block type in SDL corresponds to a class in UML. The instances of a block type are objects that contain other objects (blocks or processes).

Figure 4-52: Block type AccessPoint with virtual Controller process type



communi-cation connections The processes of each AccessPoint block are connected by signal routes, and the signals on these indicate the possible communication between the processes. The signals used between the processes of a block can be defined locally to the block.

process: objects with behaviour Processes execute concurrently, communicate by means of signal exchange (or remote procedure calls), and have the behaviour represented by an Extended Finite State Machine. The extensions are that processes may have variables and actions as part of transitions.

The process type Controller in Figure 4-52 (p.4-73) is defined by the process type diagram in Figure 4-54 (p.4-75). It defines the behaviour of Controller process by means of states and transitions.

attributes The process type also defines the variable attributes of Controller processes: cur_panel of type PId (denoting a Panel process instance) and two integer attributes cid and PIN.

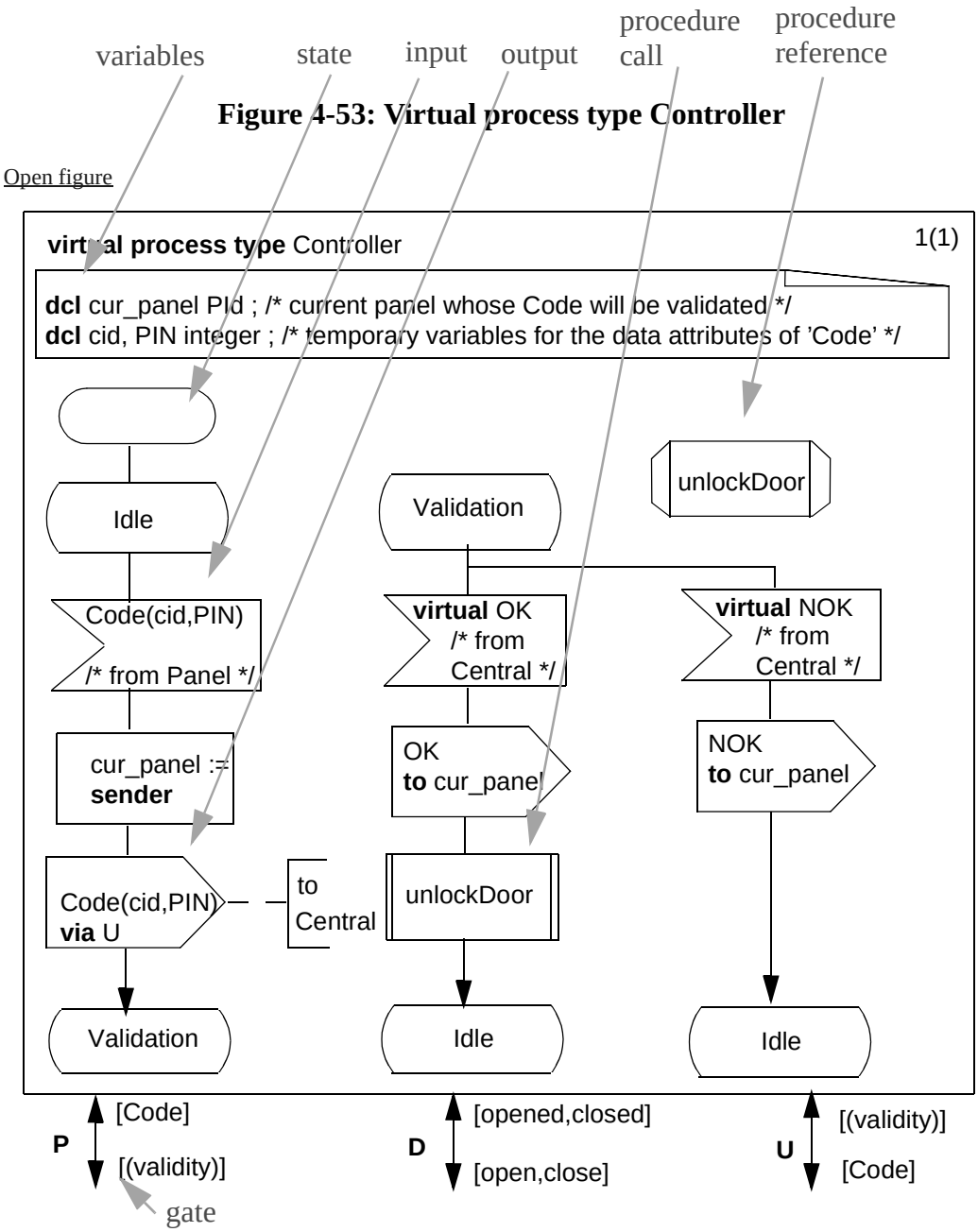


Figure 4-53: Virtual process type Controller

specialization

A type may be defined as a *subtype* of another type (the *supertype*), thereby *inheriting* all the properties defined for the supertype and possibly redefining the virtuals of the supertype.

The subtype hierarchy which is specified in UML in Figure 4-48 (p.4-70) will in the corresponding SDL design be represented by two block types inheriting the block type AccessPoint. In Figure 4-54 (p.4-75) this is illustrated for BlockingAccessPoint.

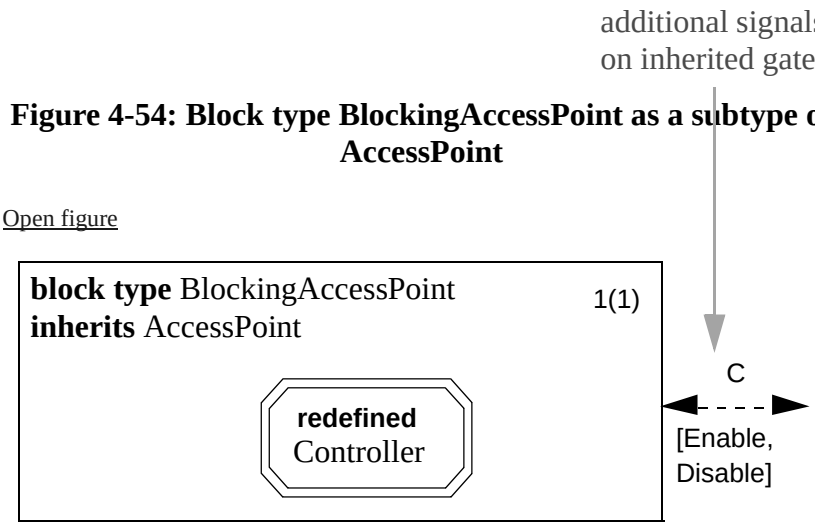
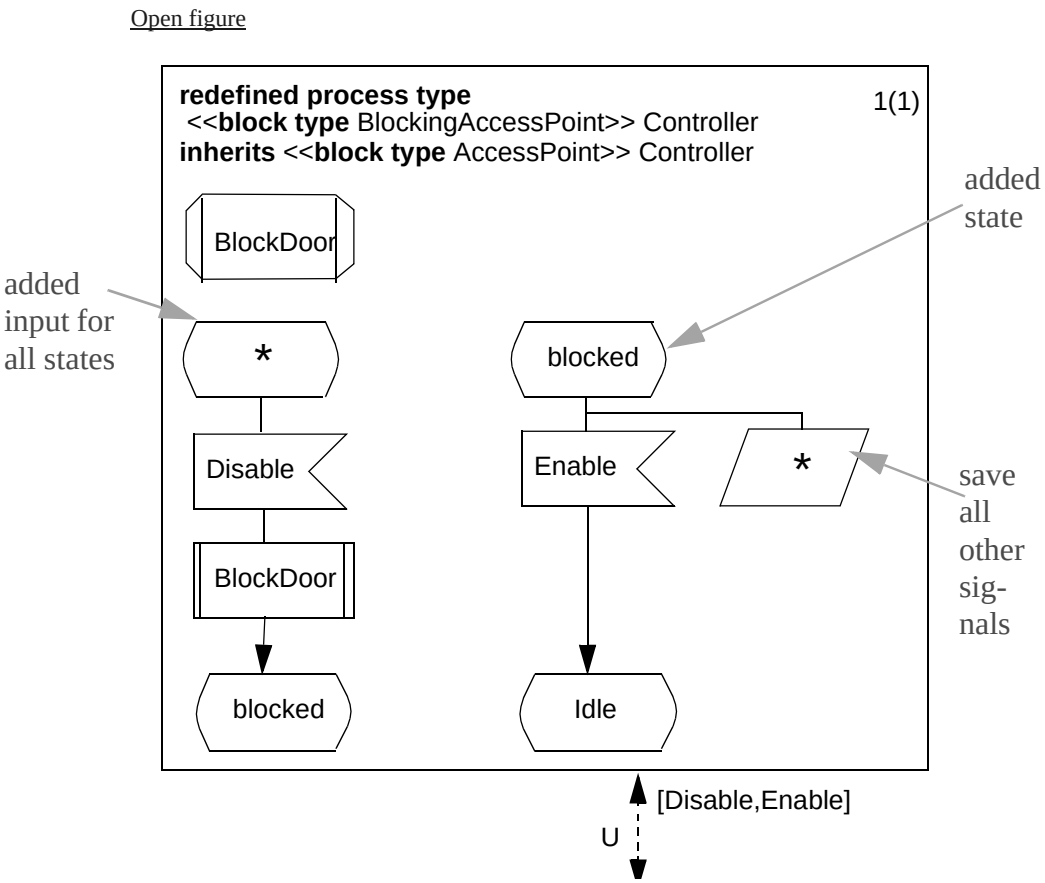


Figure 4-54: Block type BlockingAccessPoint as a subtype of AccessPoint

The redefined process type Controller inherits the states and transitions of the virtual Controller from AccessPoint, and it adds states and transitions, as shown in Figure 4-55 (p.4-75).

Figure 4-55: Redefined process type with added states and transitions



When the redefined Controller gets a Disable signal (in all states) it will enter the state Blocked, where it will only accept Enable, while all other signals will be saved (for consideration in other states).

package: the SDL library concept

In addition to the structuring of systems into blocks of blocks or processes, SDL specifications can be organised in packages. A *package* is a collection of type definitions.

In Figure 4-56 (p.4-76) the signal definitions for the access control domain have been collected in a package, and in Figure 4-57 (p.4-77) they are used by a system diagram.

Figure 4-56: Package diagram SignalLib

Open figure

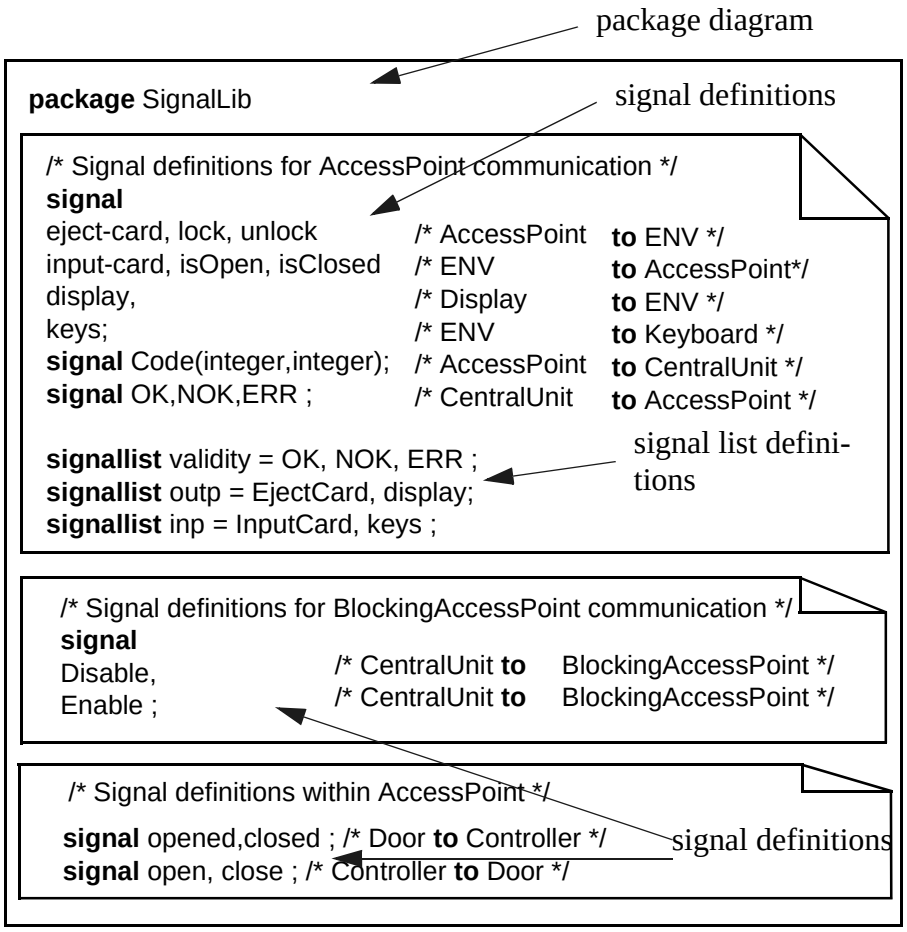
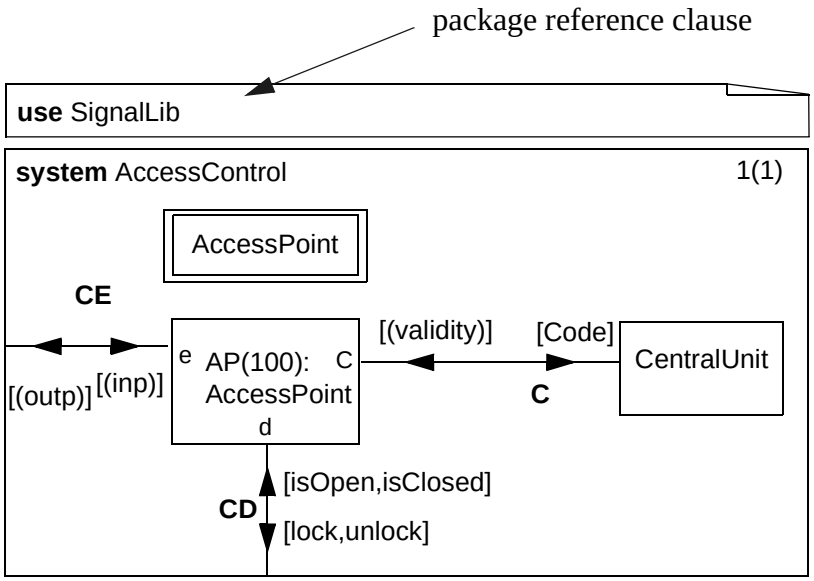


Figure 4-57: System using a package of type definition

Open figure



Guidelines on Object Modeling

Guidelines for Domain Object Modeling

Domain Object Modeling is a special kind of Object Modeling. In addition to the general guidelines for Object Modeling found in TIMe, the following *special guidelines* apply:

- Object classes with attributes, relations and connections
  If attributes are not known, just introduce the class. Include any relation or communication link that may be important - in the design activity these will be refined and detailed (or thrown away). Do not use too much time on signals or communication links, unless they are stated in the Domain Statement.
  Communication connections between classes indicate that there will be interaction property models between instances of these. For each of the communication connections check if this is important enough to call for interaction property models.
- Relations
  Do not be afraid to use illustrative relations, but be aware that they may have to be “implemented” during design, while constructive relations may be implemented automatically through a data base part of the system.
- Attributes
  If the type of an attribute is not known, simply introduce the attribute without any type, or introduce the attribute type as a class - this will then be defined during design.
- Aggregation
  Use only real aggregation when it is obvious that this is the case. If in doubt, use relation aggregation, as this the most flexible.

- Behaviour associated with the object model
  This will mostly be in terms of Interaction Models by use of MSC. If state information is important for the behaviour of an object, sketch an SDL process graph fragment for this part of the behaviour.
- Localisation (nesting)
  Do not consider this unless it is quite obvious. In case SDL is used for domain object modeling, it will produce a set of packages of type definitions. These will mostly be independent of the actual context. If domain modeling goes so far as to define system and block types, then apply the general rules of localization.

Guidelines for Design Object Modeling

Object modeling for Design is a special kind of object modeling. The general guidelines of Object Modeling applies, with the following additions:

- Object classes with attributes, relations and connections
  Attributes will be defined by attribute types that are either reused or designed.
  Associate signal lists with communication links.
  Turn communication connections into signal routes or channels when designing in SDL.
- Relations
  Stick to constructive relations if part of the product is to be implemented by a database component; otherwise “implement” all relations in SDL as data or signals.
- Attributes
  Types of attributes must be defined, preferable as ADTs.
- Aggregation
  Use real aggregation when it is obvious that this is the case, and use the SDL kind of aggregation.
- Behaviour associated with the object model
  This may still be in terms of Interaction Models by use of MSC, but more SDL process graph fragments should be developed during Design.

From UML Models to SDL Models

SDL is more formal than UML. That is the reason why SDL is chosen for specification and design, and the reason for using UML for analysis and sketches.

SDL has more specialised concepts, so in a mapping from UML to SDL a number of decisions must be taken. Most UML classes of objects will map to process types, but in UML we may define attribute types as classes, while attributes in SDL are mapped to variables of data types. Aggregated objects in UML may either map to blocks (containing other blocks or processes) or to processes (containing services).

TIMe provides guidelines on this mapping - some of them are given below. Most of them are given in a short form just to give an impression of what kind of guidelines we have.

classes -> types

Classes in UML map in general to types in SDL. Classes of objects with their own behaviour and with communication with other objects map to processes types, classes of container objects map to block types, and data object classes map to SDL data types.

attributes -> variables

Attributes of objects map to variables of data types. A difference between UML and SDL is that attributes of UML objects are just of predefined types, while variables of SDL can be of user-defined types.

Operations

Operations are either mapped to remote procedures or to signals in combination with the corresponding transition and possible reply signal.

Relations -> ?

Relations are not easily mapped to SDL. TIMe makes a distinction between *constructive* and *illustrative* relations. Being aware of this distinction when defining relations helps perform the mapping. Constructive relations will readily be implemented by a corresponding data base part of the system, while illustrative relation must be “implemented” in SDL.

Connections

Connections are mapped to signal routes/channels and corresponding gates on the types involved.

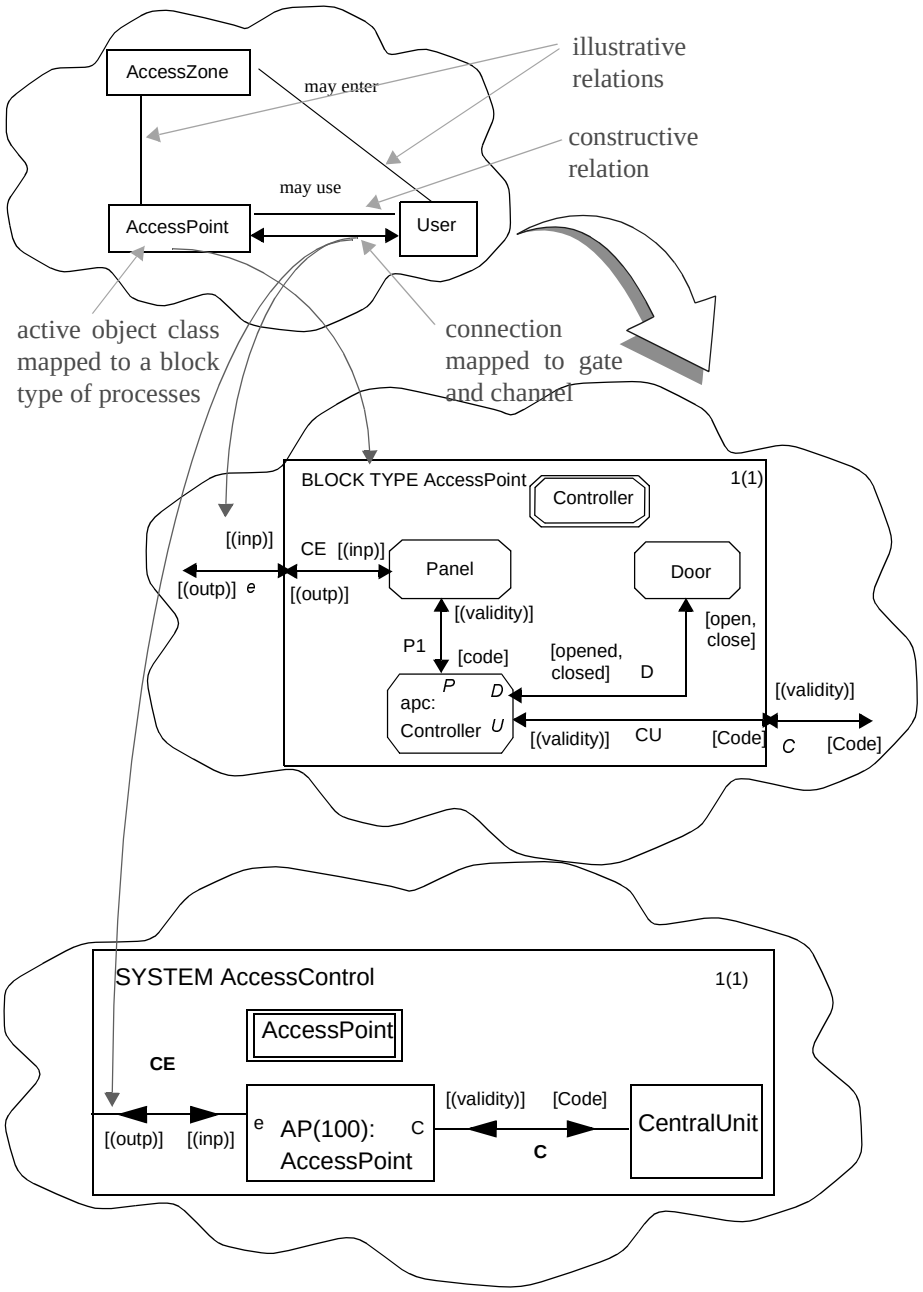


Figure 4-58: Mapping classes, relations and connections to SDL

The relations in Figure 4-58 (p.4-80) are for illustrative purposes in the mapping of the AccessPoint class to the AccessPoint block type, while the connection between Access-Point and User maps to a gate ***e***. The User class of objects is “mapped” in the first round to processes in the environment of the AccessPoint and in the second round to processes in the environment of the system.

In a further mapping of the classes in Figure 4-58 (p.4-80), the classes be in addition be mapped to classes of objects in a database of which users may enter which access zones through which access point. In that mapping the relations are not just illustrative but may map to corresponding relations in the database.

Single Inheritance

It is recommended to use only single inheritance. This is readily mapped to the corresponding mechanism in SDL. The difference is that inheritance will have more implications in SDL than in UML, especially for inheritance between process types. While UML only specifies the inheritance of attributes and operations, inheritance for process types implies also the inheritance of behaviour also.

Single inheritance for data classes is mapped to corresponding inheritance for data types in SDL - the only problem being that only operators can be inherited.

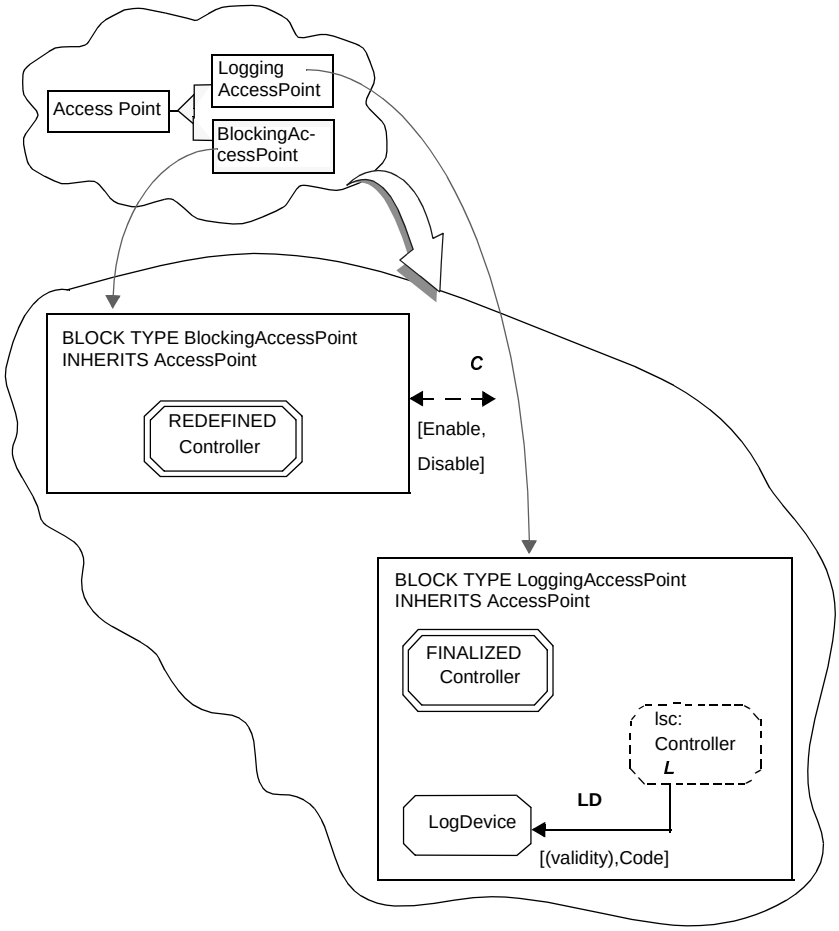


Figure 4-59: Subclasses of container object classes mapped to block types in SDL

Inheritance between classes are not restricted to UML classes that map to process types or block types. Architecture of systems can be represented by a special system class in UML and if using the real aggregation of UML the content of the system objects can be readily expressed. Subclasses of such system object classes are mapped in the same way as in Figure 4-59 (p.4-81), just substituting BLOCK with SYSTEM in the headings.

If the UML model contain inheritance between the types of events in use cases, then the mapping of this is to a corresponding inheritance between signal type definitions in SDL, see Figure 4-60 (p.4-82).

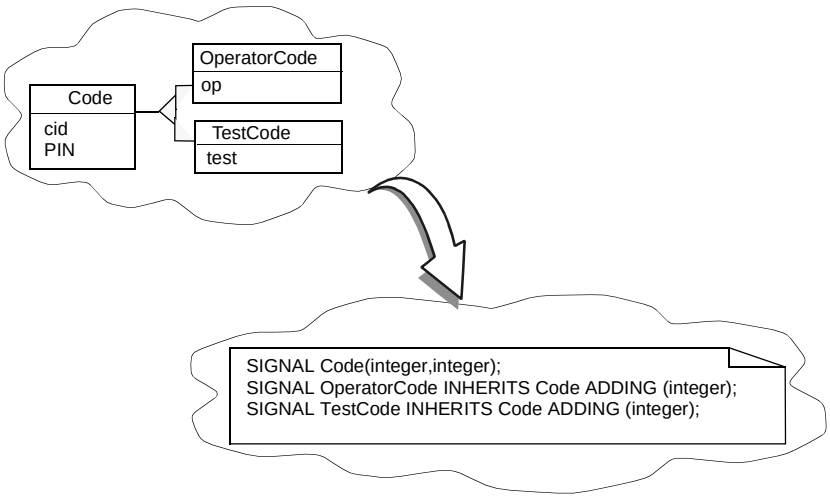


Figure 4-60: Inheritance for signals

Multiple inheritance

Multiple inheritance of the special kind where just one of the superclasses is a real superclass and the other are just “interface classes” (that is classes with only operations with no specification of behaviour, and no attributes) can in SDL be represented by inheritance combined with a gate for each interface superclass.

Multiple inheritance in general can be mapped into a type where the properties of the superclasses are copied into the type corresponding to the subclass (resolving the inheritance) or in some cases by aggregation. The first is not recommended, but must be done in some cases. The second alternative take different forms:

- If the superclasses are container classes, then the resulting block type may get a block for each superclass.
- If the superclasses are active classes corresponding to process types, then careful specification of these process types - so that they can work both as process types and as service types (that is no start transitions and input signals context parameters) - makes it possible to represent multiple inheritance by composing the process type by means of services. These services are then defined as subtypes of the services types corresponding to the superclasses, with one of them getting a start transition and actual signal parameters provided so that services do not have overlapping valid input signal sets.
- If the superclasses are data classes, then the resulting data type can be defined as a struct with each field being of the types corresponding to the superclasses.

Normally a problem with representing multiple inheritance by means of aggregation, in languages with object references, is that the objects of the resulting subclass cannot be referenced by object references typed with the superclasses. SDL does not have a general object reference concept and process instances can only be referenced by untyped PIds, so this is not a problem in SDL.

part/whole -real aggregation

In order to really specify that the objects are part of the containing object and that relations to these part objects are only meaningful in their property of being contained in this object, TIMe uses the notation in Figure 4-61 (p.4-83). It is specified that a AC-System object consists of two objects (of class AccessPoints and CentralUnit), and that the environment communicates with some of these part objects. In UML the User in the environment would have associations to the class AccessPoint in general, while what we want to express is that they only have associations with AccessPoints as part of AC-System. The mapping to SDL is straight forward - here it is indicated that CentralUnit is not an object of a class but specified directly. The definition of the block type AccessPoint is left out in the mapping - it can be defined in a package or as part of the system.

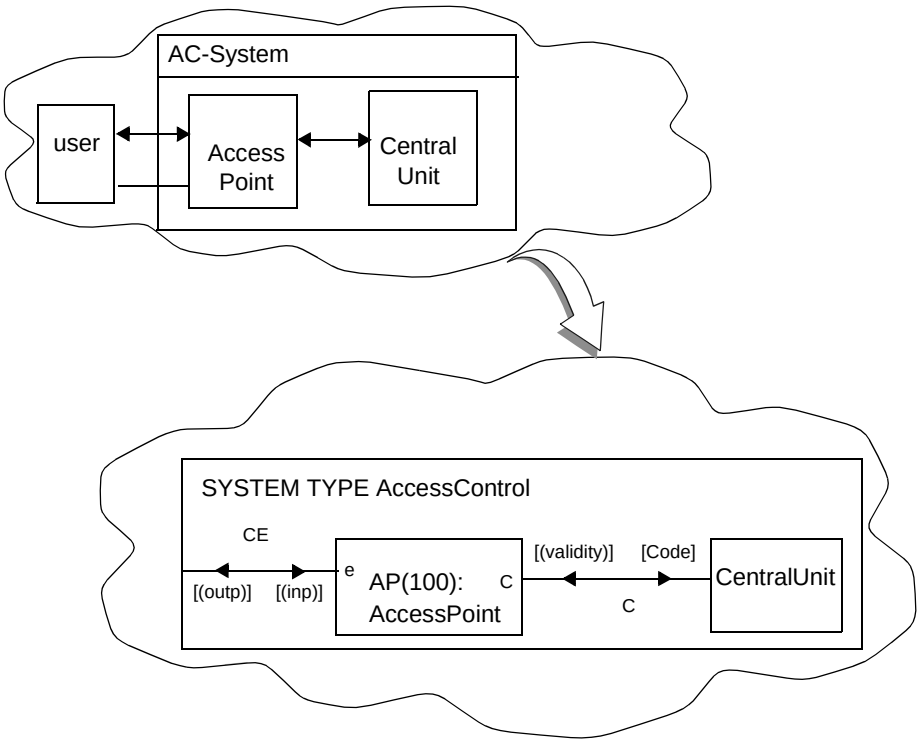


Figure 4-61: Mapping real aggregation to aggregation in SDL

Relation aggregation

UML supports a special aggregate association. Depending on how this is used, it maps

- either to whatever kind of relation mapping in SDL is chosen,
- or to relations in a corresponding data base model,
- or to real aggregation in SDL, see Figure 4-62 (p.4-84).

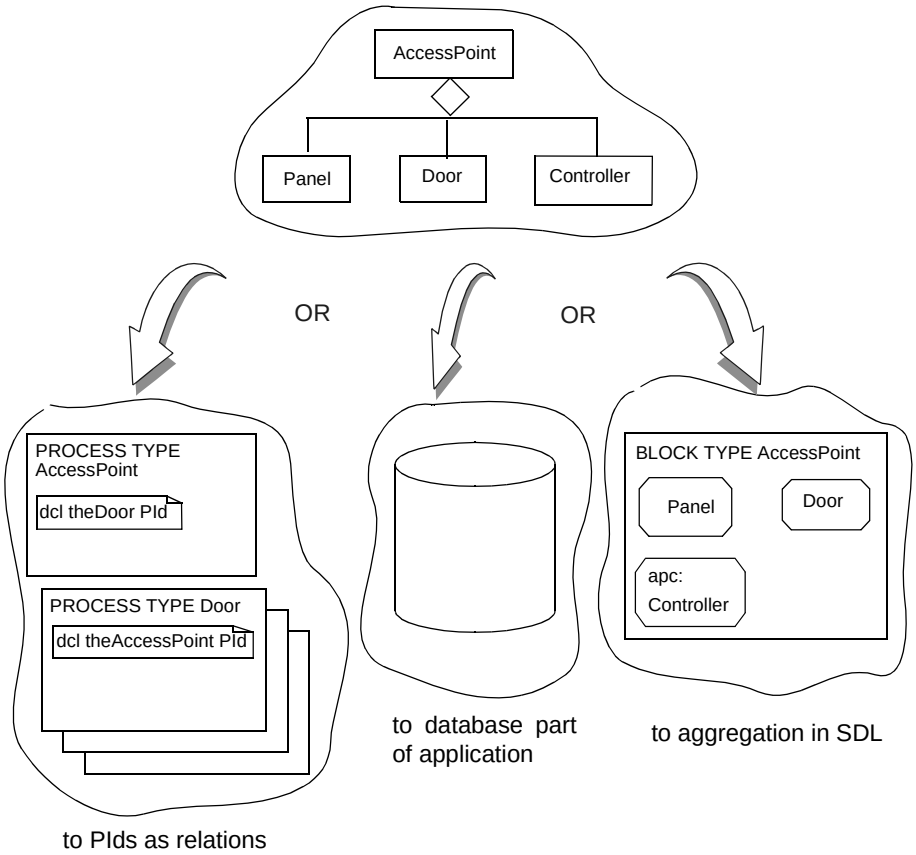


Figure 4-62: Mapping relation aggregation in OMT to SDL

Property Modelling

What is property modelling

The properties characterize the objects identified in the Object Modelling. It is, however, not always the case that the object model has been created before the property model. During the identification of the objects, properties become clear, and during the description of properties, the objects and their relations must be established.

The following are some common properties of property descriptions:

- Property descriptions cover *specific aspects*;
  - *liveness* properties: something good will eventually happen;
  - *safety* properties: something bad will never happen;
  - *overview* of functionality (functions and function lists, functional roles);
  - focus on *interaction* (use cases, MSC diagrams);
  - *capacity* and *timing constraints*;
  - *physical constraints*: temperature, humidity, power consumption, concrete interfaces,

- other not so easily formalized properties: modifiability, security, error handling

- Property descriptions may *overlap* and *underlap*;
  As an example we are used to accepting that the MSC document will not comprise a description of all traces possible of the SDL model (object model).
- Property descriptions are often *declarative* rather than imperative;
  While the object model in SDL may be seen as a complete imperative description of the system, property models are often declarative meaning that they express something which either holds or does not hold in the model.
- Property descriptions *supplement* object descriptions;

MSC for Property Modelling

The basic notation for property modelling is MSC-96. MSC highlights *interaction between instances based on messages*. MSC is most effective when the sequencing of messages between the acting objects is of major importance.

The full TIMe contains tutorials on MSC-92 and MSC-96 - the following is just an overview.

MSC concentrates on describing the message-sending between instances. The important invariant for messages is that a message must be sent before it is received.

Figure 4-63: An MSC

Open figure

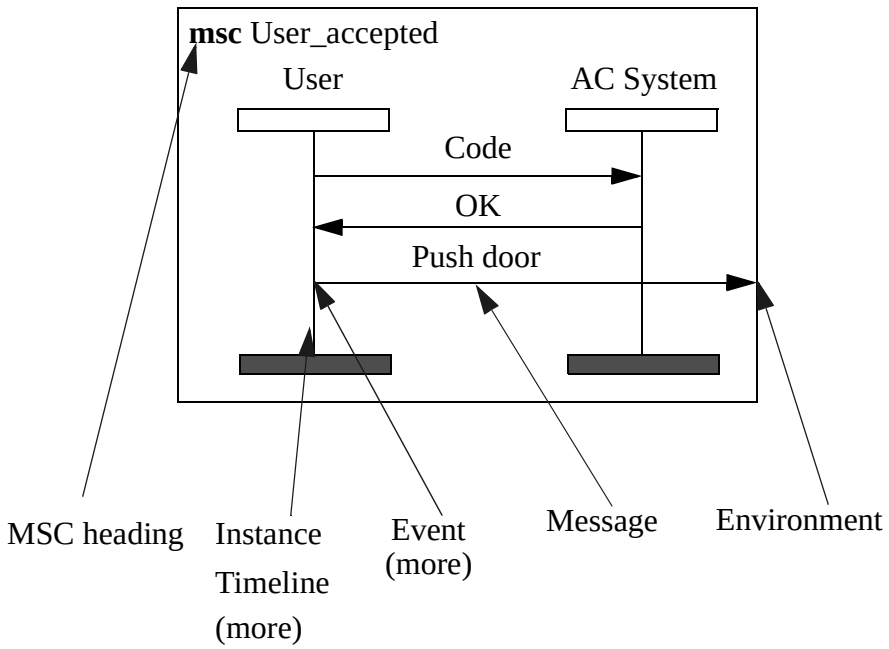
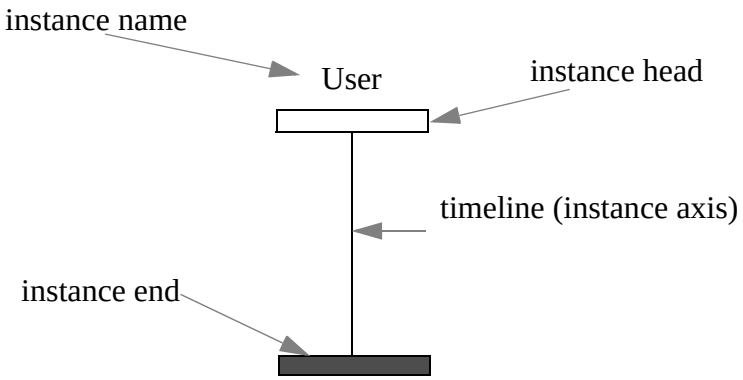


Figure 4-63 (p.4-85) describes a very simple interaction between a user and an access control system. The user presents the personal code to the system which then returns that the user is eligible to enter the door. The user then pushes the door open.

instance The actors of an MSC are called instances. They are described by an *instance head* and an *instance end* connected by a *timeline* as shown in Figure 4-64 (p.4-86).

Figure 4-64: Instance

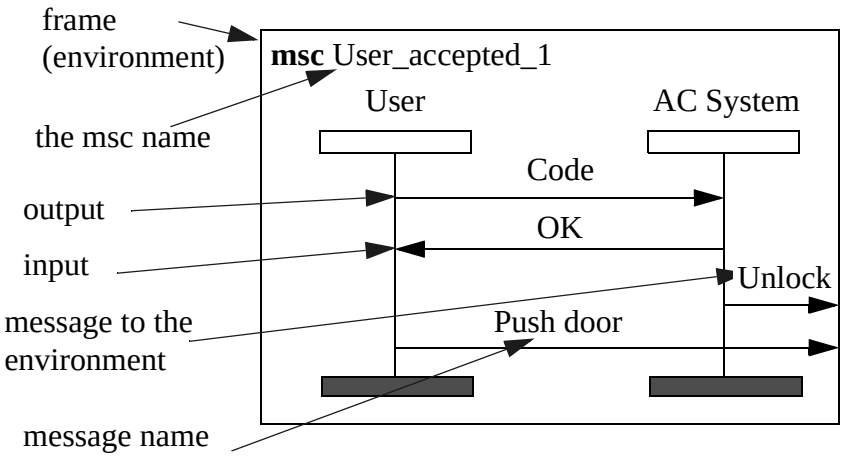
Open figure



events The instance head and instance end represent the start and end of events on the instance timeline within the MSC. The timeline of an instance contains a sequence of events.The most basic events are *output* and *input* of a message. Each message has exactly one output event and one input event. Messages are communicated between instances or between an instance and the *environment*. The environment is represented by the *frame* around the MSC diagram.

Figure 4-65: MSC diagram

Open figure



timeline The events are ordered along each timeline, but events on different timelines are not ordered.

MSC describe communication between instances. An instance need not be a process in SDL terms. In Figure 4-65 (p.4-86) we see that *AC System* is an SDL system.

MSC describes asynchronous communication. Input is normally interpreted as consumption of the message.

MSC document and Conditions

The set of mscs that are used to describe a specific piece of reality is called an MSC document. Relations between different mscs within a MSC document are called conditions. Combining two mscs where the end condition of the first is equal to the start condition of the second is legal. Combining mscs with unequal conditions is not legal. In Figure 4-66 (p.4-87) there are two conditions, *Idle* and *Door unlocked*.

Figure 4-66: Conditions

Open figure

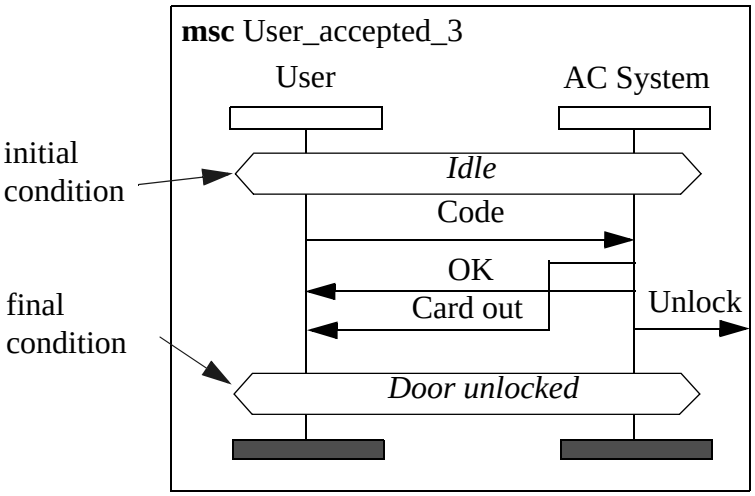
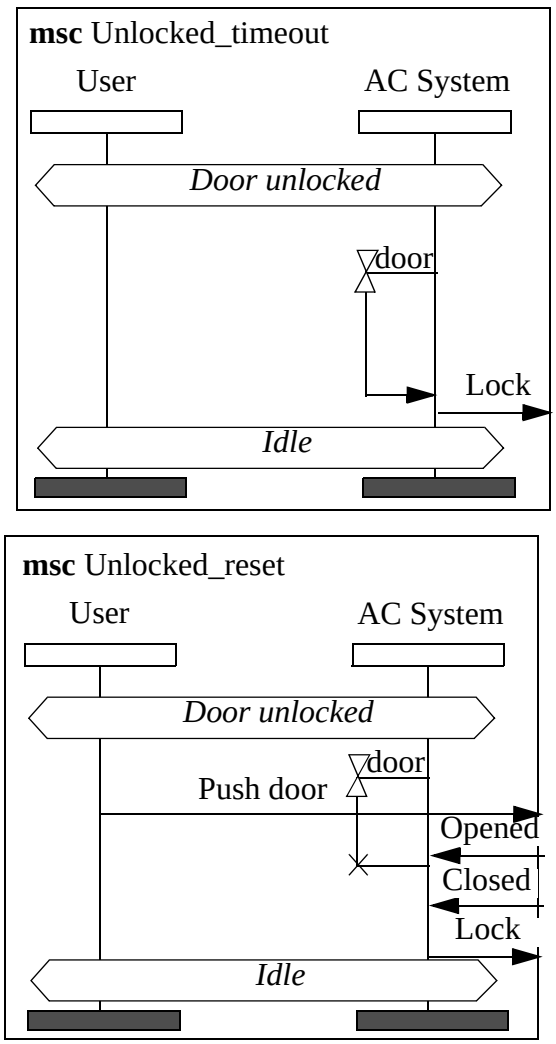


Figure 4-67: Alternatives by conditions

Open figure



The two mscs *Unlocked_reset* and *Unlocked_timeout* in Figure 4-67 (p.4-88) represent alternative courses of action from the state *Door Unlocked*.

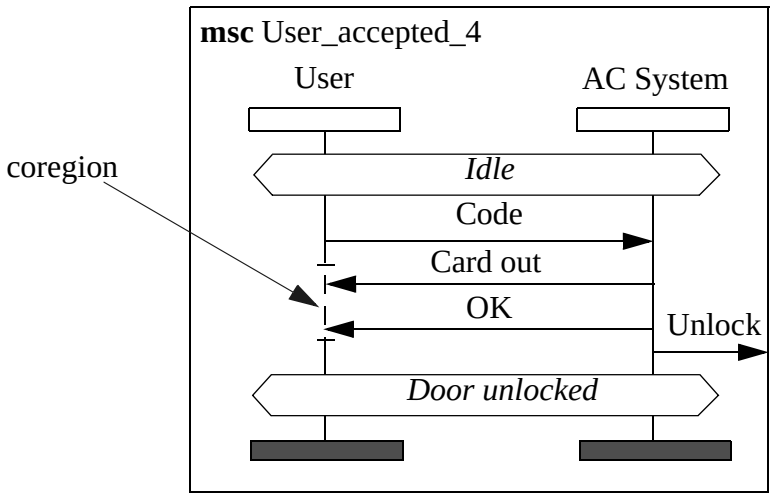
Conditions are not synchronization primitives meaning that the different instances are not “within the condition” all at the same instant. The conditions are merely there for the combination of mscs.

Coregion

Coregion is a concept which is motivated by the fact that sometimes one does not care in which order a set of events occur.

Figure 4-68: Coregion

Open figure



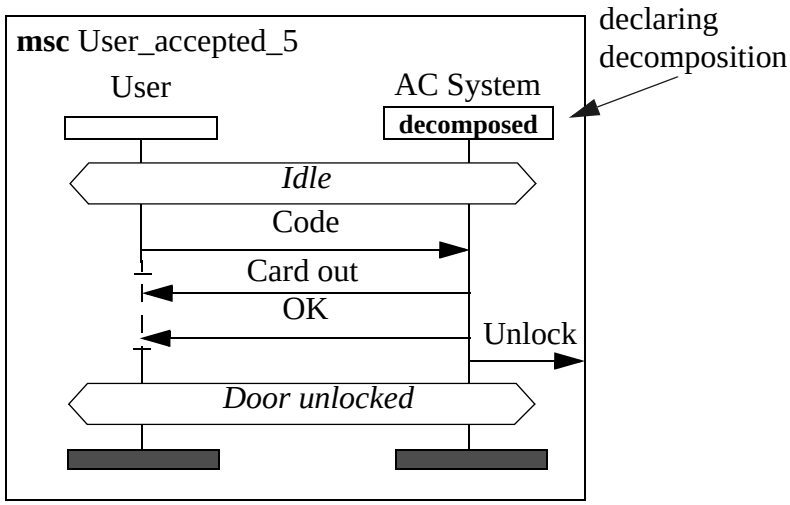
In Figure 4-68 (p.4-89) the *User* does not care whether he receives/consumes *Card out* or *OK* first.

Submsc

Submsc is motivated by the need to look into an instance for more communication details. Our *AC System* instance obviously contains a number of “smaller” instances. The requirement analysis may want to express details about the internal behavior of the system.

Figure 4-69: Decomposed

Open figure

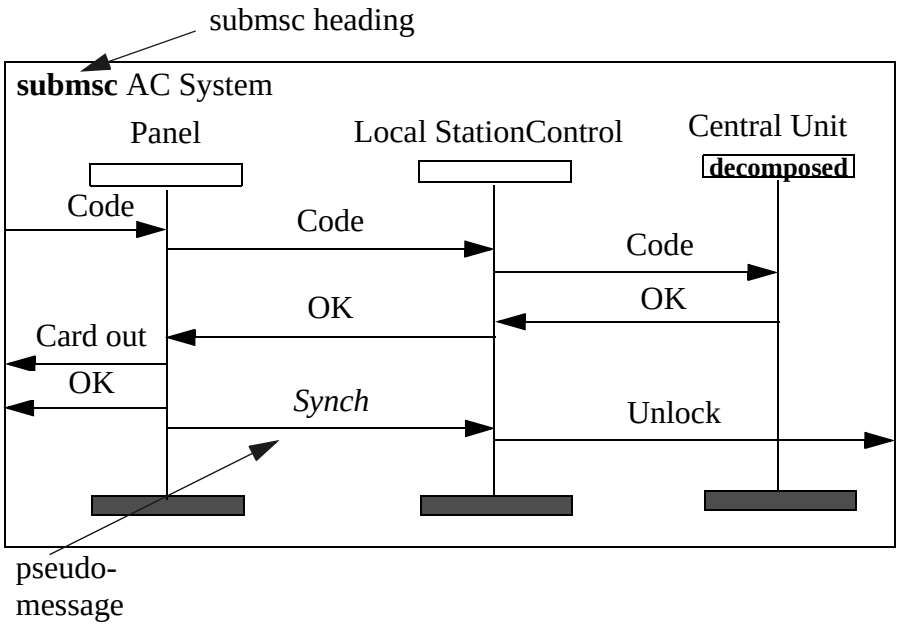


When we want to define a submsc of an instance we depict that in the instance header, see Figure 4-69 (p.4-89). The decomposed instance must have the same interface as given by the instance in the MSC of higher granularity.

AC System of Figure 4-69 (p.4-89) states that input of *Code* is followed in sequence by the outputs of *Card out*, *Ok* and *Unlock*. To ensure this in the submsc, we sometimes have to introduce additional (pseudo) messages, see Figure 4-70 (p.4-90). This is an unfortunate aspect of this mechanism.

Figure 4-70: Submsc

Open figure



Guidelines on Property Modeling

Guidelines for Domain Property Modeling

1. Identify separate services which should be offered in the domain.
2. For each service, provide a prose description.
3. For each service, define which roles provide the service.
4. For each service, make the description more precise by:
   - *Formalizing (1)*: Transform those aspects which may into a formal language. The behavior should preferably be described in MSC or SDL.
   - *Formalizing (2)*: Those aspects which do not lend themselves easily to descriptions in MSC or SDL should be described in semi-formal prose and structured comments.
   - *Narrowing*: Find out what questions were not addressed in the prose version and make decisions on these matters.
   - *Supplement*: Make sure that the precise description covers all those cases which the prose covers.
5. Associate every role with objects of the object model (Alignment).

Guidelines for Design Property Modeling

1. Take every service of the corresponding domain model and make sure that all roles are played by objects in the design structure. Remake all domain property descriptions such that they refer to the design software structure which is preferably in SDL.
2. Make the descriptions more detailed by:
   - *Decomposition*: Transform the descriptions such that they apply to the substructures of the objects and not only to the objects themselves.
   - *Breaking down*: Break down the messages and higher level protocols such that their internal structure becomes known.
   - *Revelation*: Reveal new instances and messages which prove to be interesting when a more detailed view is to be described.
3. Having reached a precise and detailed description, make sure that it is covered by the precise, but more abstract corresponding domain description.
4. Make sure to retain the structured comments and associated semi-formal prose of the domain descriptions in the corresponding design descriptions.
5. Use the design MSC property model as base for producing SDL process skeletons. The automatic production of skeletons can be used for discovering inconsistencies in the MSC property model. The produced skeletons should then be compared with the design object model and a complete design SDL model should be produced.

From MSC Property Models to SDL Object Models

The title of this section can be a little misleading - the fact is that what may be obtained is the construction of SDL Skeletons from MSC Property Models.

MSC is a formal language which is well suited to express cases of interaction between instances. SDL is a formal language which is well suited to express the total imperative behavior of processes one by one. The two notations have different perspectives on a system which supplement each other well.

We shall not always expect the MSC descriptions to cover all possible situations, but those situations which are covered are important. We should make sure that at least these situations are properly handled in the corresponding SDL descriptions.

TIMe provides a simple technique to produce SDL process skeletons for instances of MSCs. In order to have the produced SDL be a part of the final design it is necessary to make the MSCs so detailed that the instances of the MSCs correspond directly to processes of the SDL design. By careful use of local and global conditions in the MSCs, the SDL skeleton can be automatically derived.

From the SDL skeleton, the design process will add more behavior in order to cover all aspects of the process behavior. These supplements should not violate the behavior which was already generated in the skeleton. Since MSC does not have a formal data concept, the addition of tasks and decisions is one major activity when supplementing an SDL skeleton.

Even though a skeleton is only supplemented, it may be necessary to perform analysis to ensure that the final version of the SDL actually is consistent with the requirement MSCs.

List of figures

TIMe activities, descriptions and languages 3
The core themes of TIMe covered in this introduction, and supplementing themes 5
Verification and Validation 6
Sesam Sesam Inc 9
Matching objects and properties 15
Required and provided properties 16
Simple interaction property model. 16
Interface and application given aspects 17
Domain, environment, and systems 19
Context/content 21
UML for object modelling 24
MSC for interaction properties 25
SDL for design and specification of behaviour 26
The main activities in TIMe. 28
Analysing 29
Domain Analysis Models and Descriptions for the Access Control Domain 31
Domain Statement V1 32
The access control domain. 34
Attribute specification 34
Domain specific Dictionary 35
Domain Models 36
MSC User_accepted. 37
Analysing requirements 38
System and its environment 40
Contributions to the different aspects of a system 41
Context models 41
MSCs for domain- and system given properties 42
Property model from domain: MSC User_not_accepted by system 42
System specific property: Blocking Status provided by system and initiated by Operator 42
System Context/Design Outline. 43
Introducing PanelServer and DoorServer as part of AccessPoint 45

Concrete system reference model 46
Application framework reference model 47
Specification and design related 49
From domain objects to design objects 50
Application design in SDL. 53
Behaviour of Controller according to User Accepted & User Not Accepted 55
Block type AccessPoint with processes 56
Evolution of domain object model. 58
Application and infrastructure specific parts of systems into a framework. 60
Redesigned Access Control system V3 61
Cluster with LocalUnits and ClusterUnits 62
AccessPoint used in both LocalUnit and ClusterUnit 63
Access Control System type as a framework. 64
Block type Cluster as part of framework for Access Control Systems 65
An actual system based upon a framework 65
Attribute specification 69
The access control domain. 70
Possible classification of Access Points according to logging and blocking functionality 70
Environment entities interact with parts of the system 71
Composite aggregation in UML 71
Application design in SDL. 72
Block type AccessPoint with virtual Controller process type 73
Virtual process type Controller 74
Block type BlockingAccessPoint as a subtype of AccessPoint 75
Redefined process type with added states and transitions 75
Package diagram SignalLib 76
System using a package of type definition 77
Mapping classes, relations and connections to SDL 80
Subclasses of container object classes mapped to block types in SDL 81
Inheritance for signals 82
Mapping real aggregation to aggregation in SDL 83
Mapping relation aggregation in OMT to SDL 84
An MSC 85

Instance 86
MSC diagram 86
Conditions 87
Alternatives by conditions 88
Coregion 89
Decomposed 89
Submsc 90