6 Activities and Descriptions in TIMe

Content and scope 2
Description overview 3
Activity overview 20
Analysing 24
Designing 39
Implementing 41
Instantiating 43
Domain statement 44
Domain dictionary 49
Domain models 53
System family statement 70
System family dictionary 73
Family implementations 76
Family auxiliary 78
Application 80
Application reference model 81
Application models 86
Framework 120
Framework models 126
Developing framework 143
Architecture 154
Architecture models 164
List of figures 181
List of definitions 183

Activities and Descriptions

Content and scope

Scope

The core of The Integrated Method (TIMe) will be presented in the following. TIMe is centered around a number of descriptions which are developed by *activities*. These descriptions express domain knowledge, specifications in terms of external properties, system family designs in terms of structure and behaviour, implementation mappings and system instantiation. They are organised in three main areas of concern: the domain, the system family and the system instance areas.

It is assumed that the Languages and Notations used to make descriptions and the Foundation of TIMe are known to the reader.

Content

The content is organised as follows:

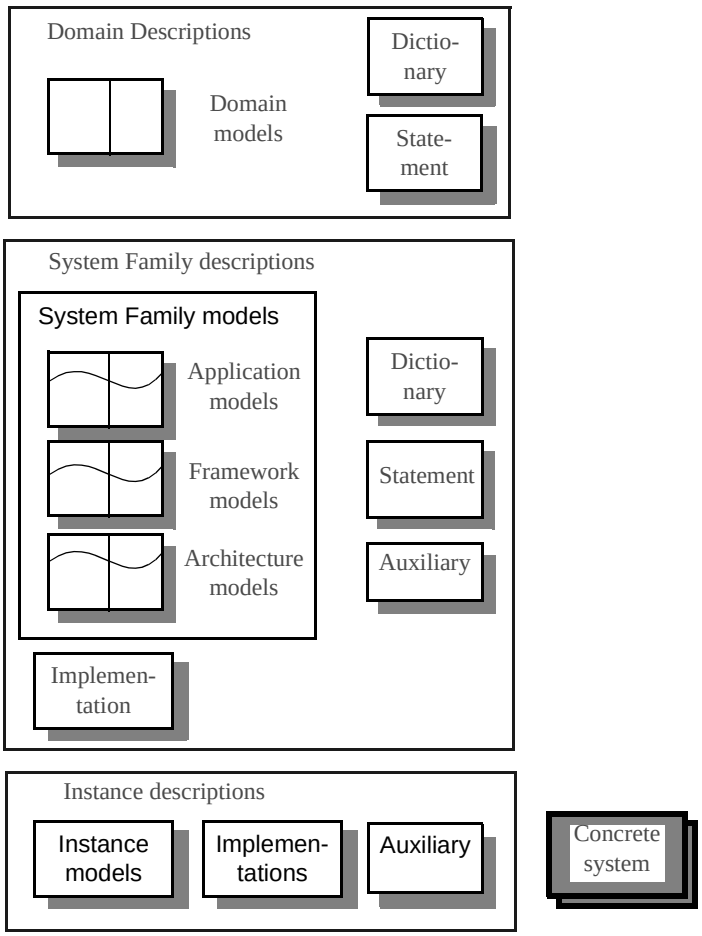
3. Description overview (p.6-3) which introduces the main descriptions and how they are organised.
4. Activity overview (p.6-20) which introduces the main activities and how they are organised.
5. Description modules. Here we present the various descriptions as modules with associated operations. They are organised according to the three areas of concern:
   - Domain: Domain models (p.6-53), Domain statement (p.6-44) and Domain dictionary (p.6-49);
   - Family: Application models (p.-106), Framework models (p.-155) , Architecture models (p.-202), System family statement (p.-86), System family dictionary (p.-90) and Family implementations (p.-93);
   - Instance: instance models, instance implementations.

Description overview

The main descriptions

Figure 6-1: The main descriptions used in TIMe

Open figure



Areas of concern

As can be seen from Figure 6-1 (p.6-3), the descriptions are organised within three areas of concern: domain, system family and system instance. In each area there are formal models and there are other descriptions (dictionaries, statements and auxiliary). The formal core of the methodology is the models which are expressed using the well defined languages UML, MSC and SDL.

Goals

These descriptions are necessary and sufficient to achieve central goals of TIMe:

1. to improve *common understanding* and *communication* among the people involved in all areas of concerns;
2. to achieve a *controlled* process towards *quality* results;.
3. to achieve *flexibility* in services and system designs;
4. to *minimise* cost and lead times and to *increase* reuse.

The descriptions have been carefully selected. They are neither too few, nor too many. There is little redundancy, as they describe different aspects and complement each other in a complete, concise and readable documentation. TIMe keeps the amount of temporary (throw away) descriptions to a minimum, and emphasizes descriptions that end up as final documentation. This does not prevent us from identifying partial descriptions that are useful in their own right, such as specifications, and to issue them in separate documents when needed.

Textual explanations

Textual explanations may be attached to models as well as to other descriptions.

Domain

Domain descriptions are organised in:

- Domain models which are collections of classes with attributes, relations and associated properties. They may be organised in several abstractions. Since the domain is about general concepts and processes that are common to many systems, it is likely that some parts of the domain models will be used within the family models. These parts will often be quite stable, reusable and resilient to change.
- Domain statements which are concise statements about the domain normally expressed in prose.
- Domain dictionaries over common domain terminology. It is important that the terminology used in other domain descriptions is harmonized with and defined in the dictionary.
- Domain auxiliary descriptions, which are any other description used. Will often be informal text and illustrations used to help reading the models.

Family

Family descriptions are organised in:

- Family models which are object models and property models describing the family on several levels of abstraction:
  - Applications that describe what the user environment want the system to do (user services).
  - Frameworks that describe how applications are distributed and supported by an infrastructure. Frameworks and applications together define the complete system behaviour.
  - Architectures that describe how frameworks and applications are realised in terms of hardware and software nodes.

- Family implementations which are implementations of family concepts. Here we find the general parts of implementations that are stable over all instances.
- Family statements which are concise statements about the family: its main purposes, its market and qualities.
- Family dictionaries which define the family specific terminology.
- Auxiliary descriptions which are any other description used, for instance test plans.

Instance

Instance Descriptions are organised in:

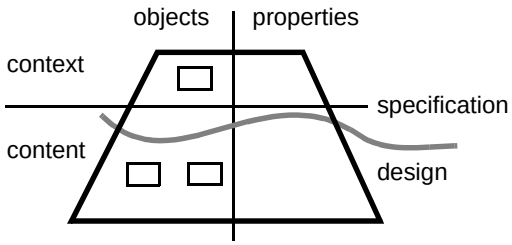
- Instance models which define the particular system instance on all abstraction levels. These may be self contained system models, but it is recommended to define instances as configurations of families.
- Implementations which are the instance specific implementations such as configuration files.
- Auxiliary descriptions which are any other description of the instance, for instance a test suite.

Model organisation

As explained in Objects and properties, models have the facets illustrated in Figure 6-2 (p.6-5). In general there are object models and property models. Seen together they define a context and a content. The context represents the entity being defined (e.g. a system type) as a “black” box and details its environment, while the content details its internal composition in terms of object structures and behaviour.

Figure 6-2: The facets of a type model

Open figure



A specification covers those aspects of a model that are relevant for its external representation and use, while the design cover the internal composition and the internal properties.

The distinction between specifications and designs is not so important in domain models, while in system family models it is important. This has been illustrated for application, framework and architecture models in Figure 6-1 (p.6-3).

Specifications

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.

Specifications are partial models

TIMe emphasises that specifications are not special models, but integral parts of type models. The reason is that we want to minimise the amount of descriptions that are thrown away. In stead we want to make use of specifications throughout the lifetime of a model:

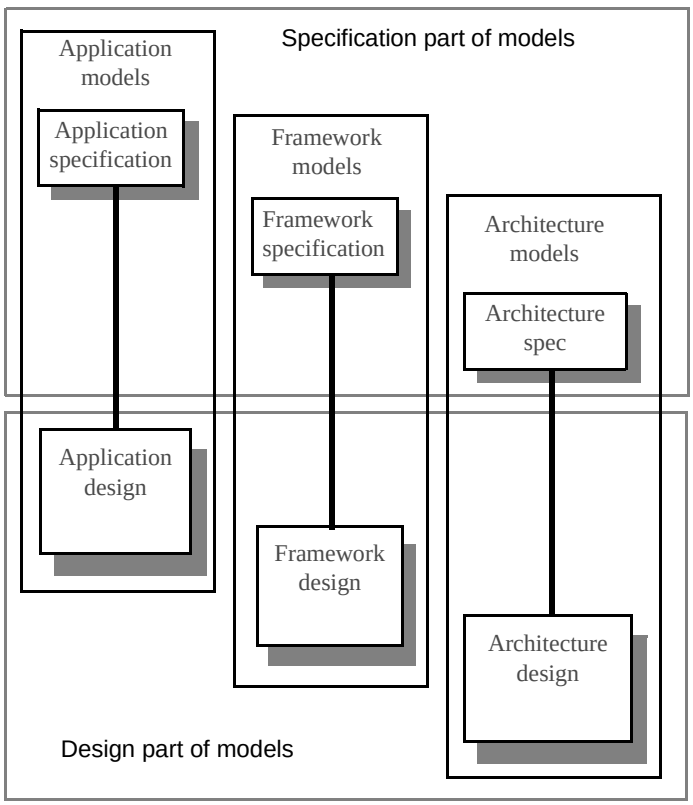
1. first to express the required properties so they can be verified and validated;
2. then to synthesise the design in a way that satisfies the specification;
3. finally to describe its provided properties for later assessment, (re)use, validation and evolution purposes.

Specifications vs design

System Family specifications contain the specification parts of application, framework and Architecture models. These are related to the design parts, as indicated below.

Figure 6-3: Specification and design related

Open figure



Specifications and designs are often developed in different phases. Specifications are produced early and play a central role in quality and process control. Designs are produced later. Therefore, in a development project they are developed by separate steps as illustrated in Figure 6-4 (p.6-8).

Requirements specification

In TIMe we consider a requirements specification as a document. It 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.

Specifications should be kept consistent with the properties provided by a design. We foresee that specification are used:

- for marketing;
- for retrieval;
- for validation of applications;
- for evolution.

Qualities of specifications

Important qualities of specifications are:

- precision and detail;
- unambiguity;
- traceability and verifiability;
- modularity that will support evolution and change.

Some specification rules

Context specification

- *Make a context diagram where the entity being specified is identified and the system environment is detailed. Describe communication interfaces and other relations the entity will handle. <x>*
- *Do not show everything in the environment, only the parts that are related to the entity being specified.*

Content specification

- *Avoid to state content requirements in specifications unless absolutely necessary and well justified.*
- *Identify the parts that are subject to requirements. Avoid describing more than required. <x>*
- *Use open aggregation to illustrate how entities in the environment are related and connected to components.* <x>

Development steps

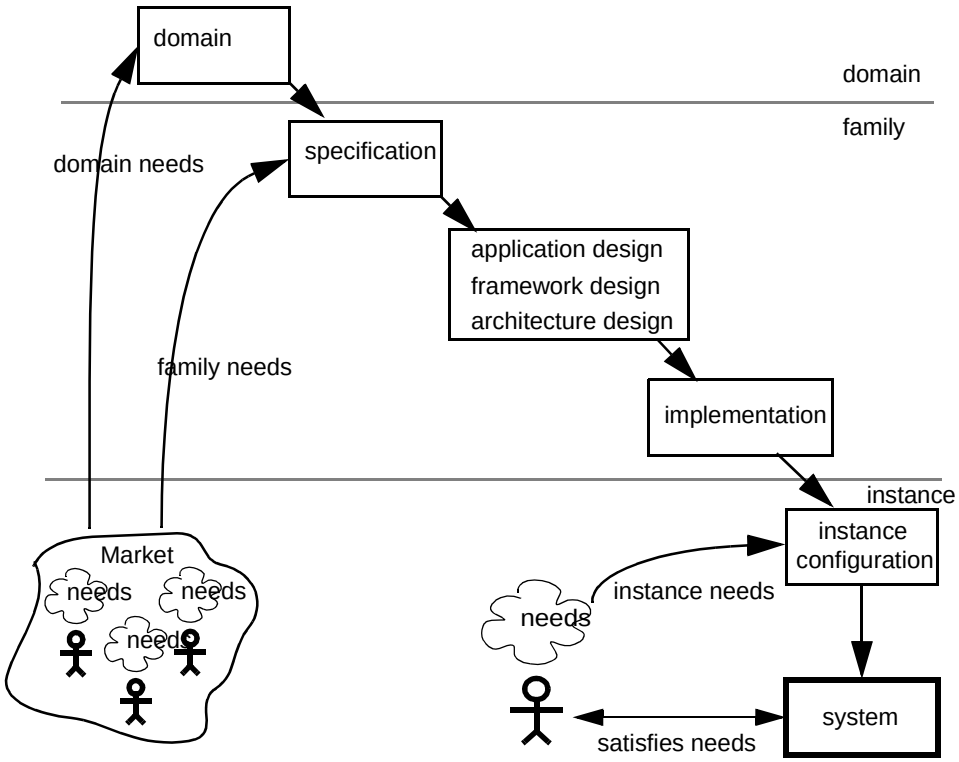
Steps

The descriptions within different areas of concern and on different abstraction levels are developed in steps that help to reduce risk, and to improve quality and control. This help to give better control and also to use the skills of different people better and to run activities in parallel.

Each model is developed in two main steps: first the specification step where the specification part (interfaces and the required properties) is made, and then the synthesis step where the design part is developed.The main development cycles are illustrated in Figure 6-4 (p.6-8).

Figure 6-4: The main development cycle

Open figure



This is of course a simplified illustration of the main steps. Considered over time we will see that the descriptions evolve gradually, that there are many iterations and that changes take place due to better insight, new requirements and new technology. We will also see that there are other, smaller cycles. For instance: to add a new service or feature to an existing product we need not modify the domain. To produce a customized instance we only need to add a new instance configuration.

Not only waterfall

Please do not jump to the conclusion that TIMe only supports the classical “waterfall” model! It is up to the actual projects to determine whether they will adopt a waterfall strategy, a prototyping strategy, use incremental development or whatever.

What TIMe provides is a set of descriptions and some general development activities including strategies and rules, see Figure 6-7 (p.6-20). The activities are described in Activity overview.

Domain descriptions

What is a domain?

A domain is a part of the world where a system instance may be a (partial) solution to some need (the problem).

As a part of the real world the domain may be very large and not well delimited from other parts of the world. Obviously, the domain descriptions cannot describe it all. It is necessary to focus on those parts that are important for the family of systems in consideration. A natural consequence is to consider only a subset of the world that may somehow be served by a family member

The domain consists of physical entities such as persons, cards and doors as well as conceptual entities such as transactions and rules. The entities may be linked by physical connections that enable entities to communicate or by associations.

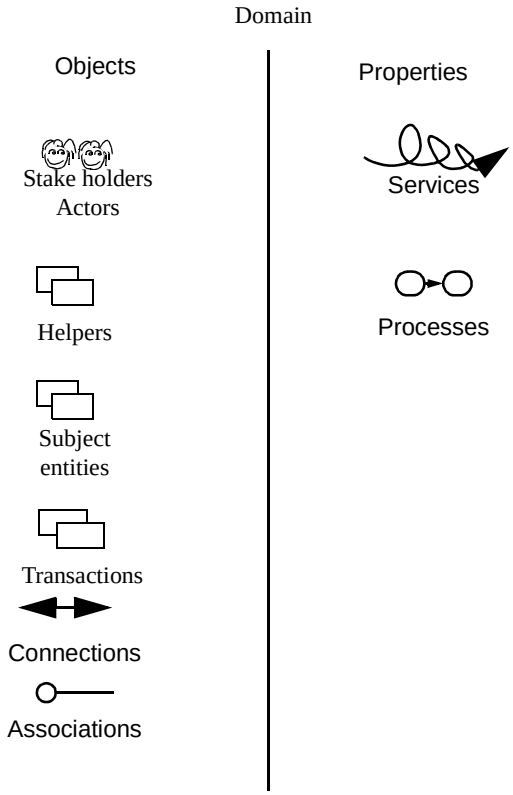
Important entity classes are:

- Stake holder. These are persons, institutions or systems with direct or indirect interest in the domain, such as users, owners, and producers. Needs originate from the stake holders, so it is extremely important to have a clear understanding of who they are, what they want to achieve and what their priorities are. These stake holders may have quite different needs that are not easily conveyed in a single model. Consequently, it will be important to identify the various classes of stake holders and possibly develop a set of models to properly describe their different *views* on the problem domain. In the Access Control system for instance, the operator´s view is different from the user´s view. A very important category are the Actors of the domain, the stake holders that take part in the services or processes of the domain. These are later going to be supported and/or represented by the system.
- Helpers. These are general tools that are used by the actors to provide services. Examples are communication systems, radar equipment and keys.
- Subject entities. These are entities that are subject to manipulation, representation or control in the domain. They may be materials in the case of a material transformation domain, e.g. moulding, or they may be entities represented in an information system, e.g. flights and seats, or they may be controlled machinery, e.g. a paper mill.
- Transactions. These are entities representing transactions or events in the dynamic behaviour of the domain, e.g. the purchase of a car, or a user passing a door.

In general the entities seeks to accomplish some services or transformation processes. Their basic need is to provide these in the best possible way.

Figure 6-5: Elements of the Domain

Open figure



The domain descriptions help to describe and clarify the needs of all the stake holders. Needs may be classified as needs to:

know about entities and relationships, e.g. the identity of users and which doors they are allowed to pass;

communicate among objects, e.g. for two persons to communicate;

transform objects from one form to another, e.g. to peel shrimps, to send a fax or to compile a program;

control something, i.e. to control or manipulate knowledge, transformations and communications, e.g. to control a fax machine.

Difference from systems

The problem domain has a wider scope than any system in the domain, as its focus is on what the systems are (to be) used for, i.e.WHY the systems are needed. Compared to a system context, it should be wider and more general. By studying the whole it is easier to see the purpose of the parts.

Thus, by studying a wider context, it is possible to get a better understanding of WHY the system is needed and what properties it should have. For instance, if the system is a communication system, we may consider why the communication is needed. In what kind of processes is the communication to be used? What are the success criteria for those?

By asking such questions we get a better understanding of the underlying needs and may be able to suggest new solutions. Perhaps the real need is not for a communication system after all? Or, perhaps the need is for a completely new kind of integration between communication and transaction systems?

A problem domain may well be modelled as any other system. Compared to systems in the solution domain there are some differences, however:

- The domain will normally include people, so it is a socio-technical system.
- The domain need not have well defined interfaces as it is not going to be used as a component.
- The domain need not be completely composed or operational to the same degree as systems. It may well be described by many fragments.
- The domain avoids system specific solutions.

System independence

The enterprise view in ODP serves the same purpose as the domain here. However, in the domain we do not wish to identify a particular system or system context. We want the domain to be general and independent of particular systems for several reasons:

- Open system analysis. We want the domain as a foundation for system analysis. It should not bind the solution but allow us to investigate many alternatives.
- Common concepts. We want the domain to describe common concepts and terminology that can help different communities to communicate: users, marketing, development, engineering.
- Reuse. We want to identify concepts and properties that will be needed in many different system families, and thereby promote reuse. The domain given parts of systems (see System reference models) contain such reusable parts.

Modelling

In principle, the problem domain can be modelled as a (socio-technical) system, more ore less, like any other system. In practice a more fragmented view, with focus on the needs of various stake holders is more useful. It enables us to describe and to analyse different viewpoints without enforcing the completeness and consistency required in a system model at this stage.

The domain may be described on all three abstraction levels (see Abstractions in models). Normally only the abstract application level is considered and this is what most methods consider as the “domain”. It is also what we shall consider the domain unless otherwise stated.

However, it is also possible to describe e.g. the infrastructure domain and the platform domain. Here we may describe common features of infrastructures, say user interfaces, and platforms, say middelware like CORBA [145]. These other domains will end up in other parts of the system such as the interface given parts or the platform.

It is not always obvious what to include in the problem domain. One should avoid system specific solutions, but concentrate on more general concepts and properties that will be common to many systems. For information systems this will typically be types of passive objects (see Active and passive objects) and associations that will be represented in every system in the domain. For instance it is reasonable to believe that every seat reservation system will handle flights, seats and passengers.

System specific solutions should be avoided, but general objects/types and functions performed by (parts of) systems shall be covered.

Why make domain descriptions?

Domain descriptions are used to define a domain and its terminology. domain descriptions serve three main purposes:

- To provide a firm basis for communication and common understanding amongst different Stake holders such as: end users, owners, developers, marketing and production.
- To provide a firm basis for product planning, i.e. to analyse needs, to seek possible solutions and to specify the goals for a new product development. In this it is extremely important to consider variability and to plan a system family from the beginning that will cater for the variability. During product planning it is also important to consider existing systems and competition.
- To boost productivity and quality by systematically capturing and maintaining common knowledge in one place (possibly at a strategic company level) rather than arbitrarily in various system specific documents. Domain descriptions will hold for many different systems and identify stable, reusable concepts and properties. Components that are derived from them are likely to be quite stable and therefore reusable across systems and system families. The domain given part of the application reference model has been introduced to highlight this aspect, see Domain given (p.-101).
- To establish a common understanding of terminology, phenomena, concepts, and tasks by making domain descriptions (a set of models and a set of rigorous prose specifications).
- To describe general objects/types and properties that is common to many systems in the domain and thereby achieve a high degree of reuse.

Domain description content

Domain descriptions contain the following description modules:

Domain statement (p.6-44), Domain dictionary (p.6-49), Domain models (p.6-53) and auxiliary domain descriptions.

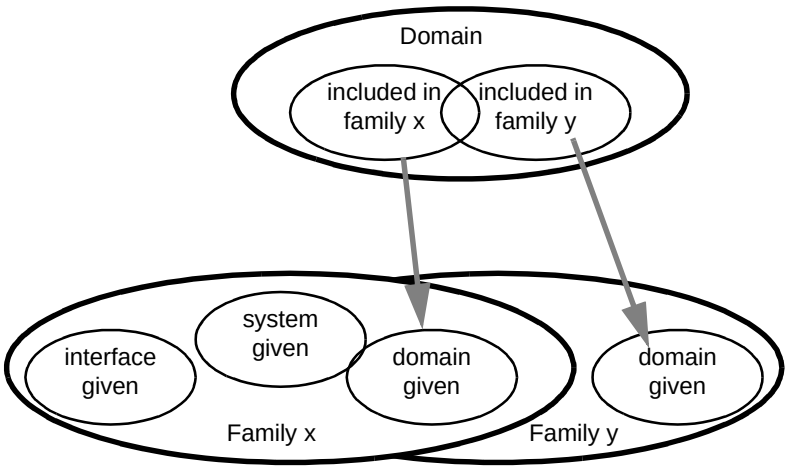
Domain relationships

With Families

The domain will normally include entities that will be part of (the domain given part of) systems, see Figure 6-6 (p.6-13).

Figure 6-6: Parts of the (application) domain will be realised in the domain given part of systems

Open figure



We seek to keep the domain given parts of systems formally related to the domain. These parts may not only contain “data objects” (passive objects), but also active objects. In the end, the domain models should cover all the domain given parts of systems.

Harmonising domain descriptions

Within domain

The various domain descriptions should be *harmonised* as much as possible with each other: phenomena and concepts in the dictionary should have their counterparts in the domain object model. If properties are properties of entities in the dictionary, then the terminology introduced there should be used in the property models.

Domain harmonisation

- *Align the domain object model, the property model, the statement and the dictionary with each other. At least define every main term in the statement in the dictionary.*

As an example of this guideline, look at the example descriptions and see how *Access Zone*, *Access Point* and *User* are treated in them.

Details will be given for the component descriptions: Domain statement (p.6-44), Domain dictionary (p.6-49), Domain object models (p.6-54) and Domain property models (p.6-60).

Domain Statement, Dictionary, Domain Object Model, Property Model.

With family

When system families are introduced, domain descriptions will be harmonised and extended to cover what is common for many systems or system types. Domain descriptions are extended and detailed so that they take domain specific system elements into consideration. The purposes are:

1. to promote reuse;
2. to improve traceability;
3. to simplify maintenance;
4. to simplify design.

There are two sub-activities:

- Reuse analysis: to analyse the design looking for object types and properties that will be common to most systems in the domain. They will then be re-used in system designs (families and system instances):
  - add new object types and properties to the domain descriptions;
  - adjust the domain descriptions to correspond as closely as possible with design without introducing system specific details.
- Establish links: define the relationships between design and domain descriptions.

Details will be given for each description module: Domain statement (p.6-44), Domain dictionary (p.6-49), Developing domain object models (p.6-63) and Domain property models (p.6-60).

Family descriptions

What is a family?

What

A system family contains the generic result of systems development. It represent a product base on which a company may capitalise by producing and selling system instances. As explained in Abstractions in models, systems are modelled on several levels of abstraction:

- application;
- framework;
- architecture;
- implementation.

Each level is developed in two main steps: the specification step which emphasises external properties and the design step which emphasises the internal content. The general approach is to work from the application level towards the implementation level and from the specification towards the design within each level.

Note that applications, frameworks, and architectures are somewhat independent and interchangeable. An application may, for instance, fit into several frameworks, and a framework may accommodate several applications. Each may use components that are shared between families.

In most cases a system family is designed to serve one market segment within a domain. As the application level is closest to the market needs, it seems natural that the system family should be defined on the application level. However, to fully generate system instances, we need information provided in the framework, architecture and implemen-

tation levels. Therefore these levels must be included when we want to produce concrete system instances. To produce an abstract system instance for simulation purposes, on the other hand, the application level is sufficient.

Types and Families

Object models in the family area will contain types defined with a context and a content, see Anatomy of object models. To some extent it is possible to define a system family formally as a type, e.g a System Type in SDL. But to define the family as just one type would probably be too restrictive:

- it would make the family specific to the level (application, framework, architecture) where the type belongs;
- it would not capture the component libraries involved;
- it would be restricted to what is formally expressible in the languages.

In stead we consider the system family as a more loose organisation (library) of types and other generic entities, which are used to generate system instances within a domain. A system family need not be self contained. It may well import parts from and export to other system families.

When to use

There are cases (one-of-a-kind systems) where a family may seem to be overkill. But even in such cases it appears that most results are generic in the sense that they can be used to produce many similar system instances. They may also contain parts that can be used in different systems. Therefore there is an implicit system family, which is made up by the generic system descriptions that are produced anyhow, even in these cases.

In more interesting cases many instances with a range of different properties shall be produced. In such cases it is desirable to analyse the needs for variability, and to design the family so that instances can be configurated and build as cost effectively as possible. This may require some additional design effort that has to be balanced by savings later on when system instances are configurated and built.

Variability

Variability can be represented in two principally different ways:

- By different versions of descriptions. This kind of variability (outside the descriptions) is traditionally handled by configuration management systems, see Software configuration management.
- By variable parts in the descriptions, typically by parameters, configuration variables, virtuality. This kind of variability is expressed in the notations and languages we use.

Our main focus here will be on the latter kind of internal variability. It involve all the descriptions in a system family.

Why make family descriptions?

The central idea behind the system family concept is that companies will benefit from being more “product oriented”. Rather than being driven by the short term goals of individual system deliveries, which is tempting since each system sold give immediate payback, development effort should be invested in lasting results that can generate a better business in the long run. Companies that focus too much on individual systems tend

not to have a clear picture of what their products are and how they relate to the market. They often have difficulties in keeping the cost down and to keep schedules. After some time they get trapped in vicious circles where they never have time nor money to improve.

Ideally:

- development effort should be invested where the results can be resold many times;
- the cost of each adaptation and sale should be as low as possible

The family concept is introduced to achieve these goals and to given an overall cost reduction.

Content of family descriptions

A system family contains the following main description modules:

System family statement (p.-86) - a concise prose statement that characterises systems in the family.

System family dictionary (p.-90) - a dictionary over family specific terminology.

System studies (p.6-17) - which are temporary descriptions of alternative system principles made in order to assess alternatives and choose the best

Application models (p.-106) - which define the system behaviour related to user needs.

Framework models (p.-155) - which define additional behaviour needed to support the application.

Architecture models (p.-202) - which define the physical platform and how the framework functionality is implemented.

Family implementations (p.-93) - which contain the implementation details in programming languages and hardware description languages.

Family auxiliary (p.-96) - which contain other descriptions such as methods for evolution and instantiation.

The needs for variability is considered on each abstraction level and for the system family as whole. An important result is a method for binding the variability and configuring system instances.

Another important result is a method for flexible introduction of new services. One main goal is that the method shall lead to a flexible framework and Architecture that will allow us to add new services mainly by working on the application level.

System studies

Objectives

System studies are made in order to create deeper understanding of the technical problems and solution possibilities at an early stage. By exploring the "solution space" for a given problem they seek to find innovative solutions that will lead to better systems. "Better" in the sense that the quality is good, that the cost is competitive the solution is flexible for change.

What

System studies are not a special kind of descriptions, but rather incomplete family descriptions that are sufficiently detailed that alternative solutions may be compared and assessed. This means that it must be possible to compare development costs, production costs, risks and potential user satisfaction. It may be necessary to perform prototype developments and laboratory experiments.

Note that system studies involves all the abstraction levels, and builds on the activities described for those. It can be seen as the start of system specification.

The results are system sketches consisting of rough specifications and content outlines. They need not be very detailed, but sufficiently detailed to identify all critical parts and the principle for implementing all services. They should be more detailed where there is uncertainty concerning technology, performance or cost.

Note that the resulting descriptions (for the selected system concept) are not to be thrown away. They are to be gradually elaborated into complete descriptions. However, if well formed, they may later serve as an executive introduction to the system.

Two cases:

1. Feasibility studies which are performed when a new system family is to be made.
2. Impact studies which are performed when an existing system family is to be evolved.

System study notations

- *Do not feel obliged to use only UML in all sketches needed during this activity. It is more important that all possible elements of the system and its environment come forward. Make competing sketches. Have possibly one person responsible for "formalising" these sketches into UML sketches.*

Instances

A system instance is a (real system which can perform behaviour and provide services.

The system instance area of concern contains system instances produced from system families.An instance description describes a system instance.

When a system family is defined, a system instance can be defined by reference to a system family using relatively simple configuration statements to binds the variability in the family. If there is no family however, it is necessary to define each system instance completely in self a contained instance description.This can only be recommended for one-of-a-kind developments.

In TIMe, system instance descriptions are organised in:

- Instance models that formally define the system instance, usually by configuration of some family;

- instance auxiliary descriptions that provide supplementary documentation.
- instance implementations which are the instance specific implementations such as configuration files.

Relations and traceability

Rationale

It is clearly necessary to establish and maintain clear and traceable relationships between the various descriptions and models. There are several reasons:

- Since each model or description is concerned with a limited area of concern and abstraction, a complete documentation is made up of a set of interrelated models/ descriptions. In order to read and understand this complete documentation it must be possible for a reader to navigate in the descriptions and to understand the relationships as easily as possible.
- We need to trace the relationships from required properties to the design objects where they are provided. There are at least, two reasons for this:
  - quality assurance need to check that every requirement is satisfied;
  - when a requirement is changed we need to analyse what impact it will have on the design.
- We need to ensure consistency both between models and within models. This can be achieved either correctively by comparing models or constructively by ensuring that models are derived according to rules, e.g. by automatically translating from abstract models to implementations.

Relationships

Relationships between all the models and other descriptions shown in Figure 6-1 "The main descriptions used in TIMe" (p.6-3) must be defined. There are:

- Domain to family relations:
  - object to object;
  - property to property.
- Family to instance relations:
  - object to object;
  - property to property.
- Family internal relations:
  - application to framework relations,
  - implementation relations,
  - property to object relations,
  - specification to design relations,
  - validity of interfaces,
  - dictionary, statement and model relationships.
- Domain internal relations:

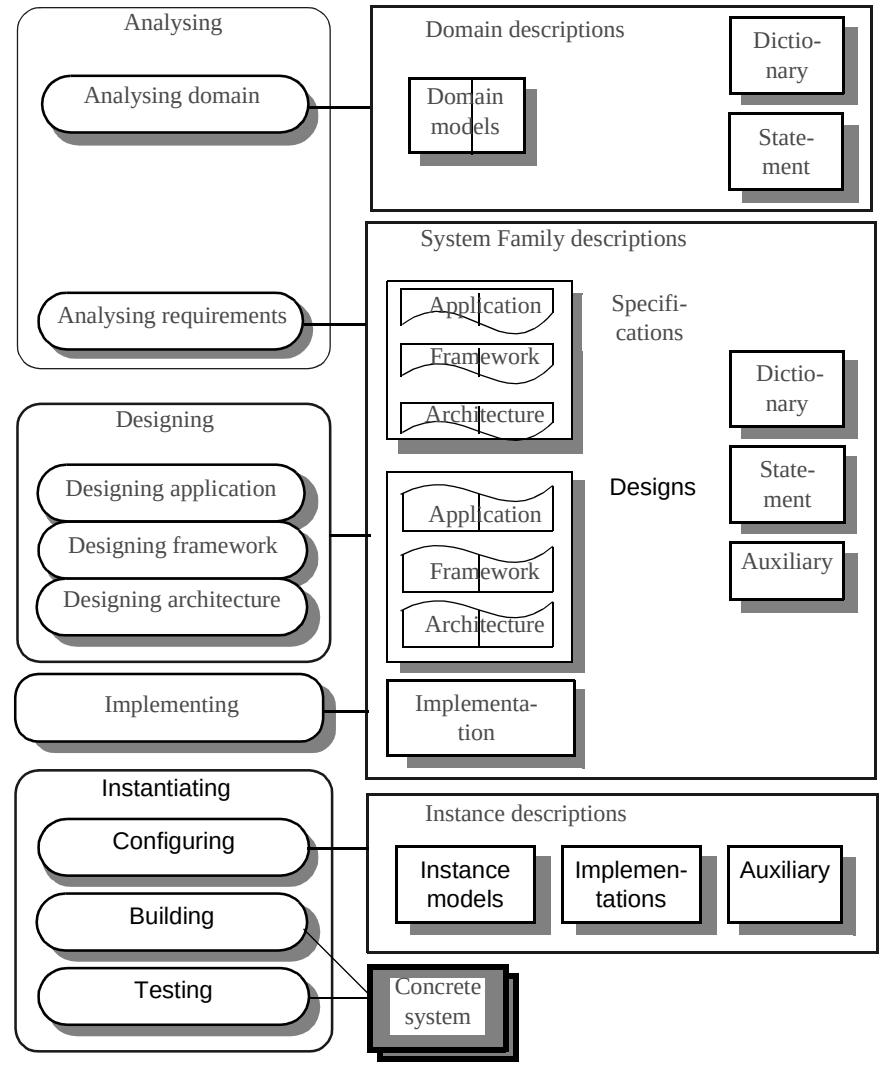
  - dictionary, statement and model relationships.
- Instance internal relations

The precise definition of relationships depend on the languages that are used. They will be elaborated under the various description modules.

Activity overview

Figure 6-7: The main activities in TIMe

Open figure



General activities

As illustrated in Figure 6-7 (p.6-20), TIMe consists of five main activities:

- Analysing (p.6-24). The purpose of this activity is to analyse a domain and to specify the requirements to a new system family. It makes the following descriptions for the first time:
  - domain models;
  - domain and family statement;
  - domain and family dictionary;
  - application, framework and architecture specifications.

- Designing (p.6-39). The purpose of this activity is to develop the design part of application, framework and architecture models, including the detailed behaviour of all objects.
- Implementing (p.6-41). In this activity all the detailed implementation descriptions are produced including source code for the software and detailed hardware descriptions.
- Instantiating (p.6-43). The purpose of this activity is to produce (executable) system instances that satisfy specific suer needs.

For each activity we seek to give practical strategies and rules that will help to produce high quality descriptions and target systems. As far as possible the approach will be illustrated by practical Examples.

These main activities consists of sub-activities. At the lowest level, sub-activities are seen as tasks (operations) that "belong to" description modules, for instance domain object models. Such tasks will be described under the description modules they belong to.

Specifications and designs have been identified as separate entities in Figure 6-7 (p.6-20) in order to distinguish the specification and design activities. The reader should we aware that this does not imply a separation of the models.

System development is rarely a straight forward process where the optimal solution is found at first attempt. It is more of a trial and error process. By working with a particular solution we often learn how it can be improved or get ideas for radically better solutions. Therefore it is recommended to spend some time investigating the solution space before selecting the system concept to develop. For this purpose we develop System studies (p.6-17) which are compared and analysed with respect to cost, technical feasibility and market potential. The most favorable alternative is then selected for further development, if indeed it has sufficient business potential.

Ordering

There is no specific ordering imposed on the activities themselves, but in practical projects they will be ordered, and indeed the ordering is significant. It is essential that specifications, designs and implementations follow each other in that order for each part of the system. But it is not necessary that every part is developed at the same time. Therefore it is possible to start design before all parts are specified, and to start implementation before all parts are designed. This is further elaborated in the Process models.

It is important to note that the activities are not independent, but influence each other in various ways. As an example, a design activity may lead to insight that triggers a change in the domain models. These mutual interactions are not indicated in Figure 6-7 (p.6-20), but will be explained in the text. We have sought to describe the activities in a generic way that can be used in most processes. However, the best way to carry out an activity depends on the actual state of its input descriptions and output descriptions at the time when it is invoked. In order to take this into account, we provide some alternative strategies and activity subcategories.

Activity categories

Activity categories

As explained in Activity categories there are three main categories of development activities:

- *Make activities* which make or synthesise descriptions for the first time, possibly based on other descriptions, e.g. to make SDL process graphs from requirements expressed using MSC.
- *Evolve activities* which perform (incremental) development of existing descriptions. They may either *add* new properties, e.g. add a new service to existing application models, or *change* existing properties, e.g. correct errors.
- *Harmonise activities* which ensure that models/descriptions are consistent with each other, e.g. to make the domain dictionary consistent with the domain object models.

Making family

We may distinguish two cases:

1. Developing from scratch, where a new domain and a new system family is developed without any product base to start from. The starting point is just some needs or a product idea.
2. Developing from existing families where a new system family is developed using existing families as basis. In principle we go through the same steps as when developing from scratch, but now we seek to utilise as much as possible from the existing Families. Design by reuse is a central concern. This case is considered the normal for development of new system families in a company that have used TIMe for some time. Such companies have established extensive domain descriptions and component libraries which are taken into account in all new developments.

Both cases will follow the same overall strategy, but the detailed activities will be different. Normal development will put more emphasis on reuse and adaptation of existing solutions. If, for instance the framework and the architecture is reused from an existing family and only the family is modified, then the effort will be far less than when developing from scratch.

Evolving family

Evolution is caused by new requirements. It may be requirements for new services or service features, it may be requirements for a new platform, or requirements for a new interface. The abstractions and the system reference model we use in TIMe has been chosen with evolution in mind. Areas that are likely to change independently of each other are separated:

- Changes in services or user interfaces are defined on the application level. Here the distinction between domain given, system given and interface given parts, help to further isolate changes.
- Changes in implementation platform are defined on the architecture and the implementation levels.
- Changes in infrastructure are defined on the framework level.

The situation today is characterised by increasing demands for service flexibility. In order to stay ahead of competition, the lead time for specifying and implementing new services must be as short as possible. At the same time, many products must be supported on a range of different platforms. This means that system Families should be designed with flexibility and diversity in mind and supported by tools.

The various evolution activities are dealt with for each description module separately.

Harmonising family

The harmonising activities take care of the iterations and feed-back between the various descriptions.

Analysing

Objectives

The objectives of analysing are:

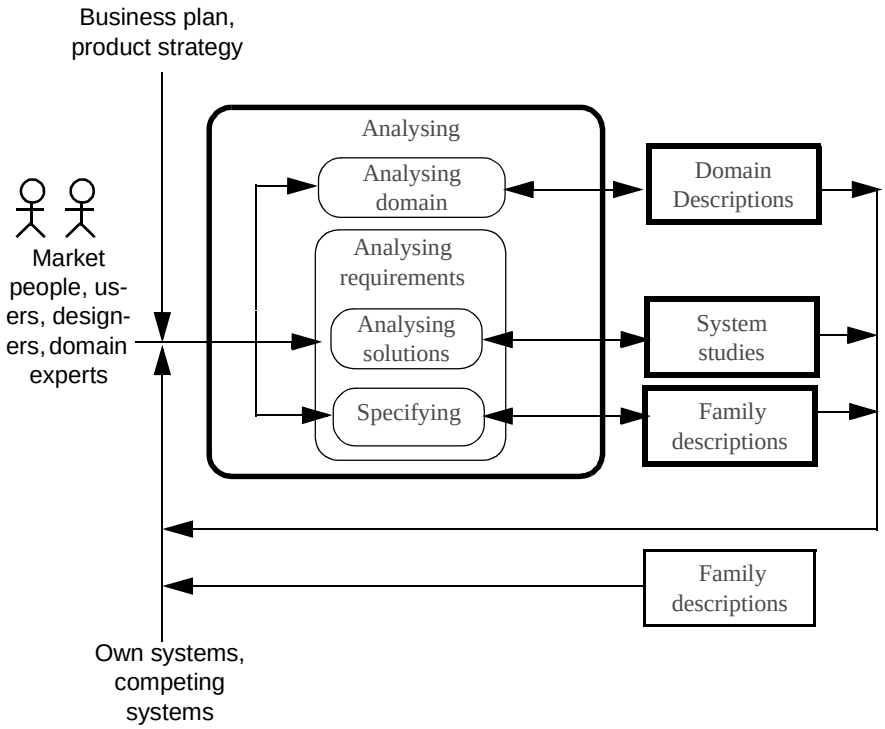
- To understand the domain and what users and other stake holders want to achieve, i.e. their needs.
- To find improvements they will consider valuable.
- To plan new product families that will give valuable improvement and thus create business in the future

What

Analysing is part of product planning which is a more or less, continuous task in any prosperous company. It is constrained by the company's strategic plans and seeks to come up with ideas for new products or enhancements to existing products.

Figure 6-8: Analysing

Open figure



Central to the task is a deep knowledge about a problem domain and existing system solutions. It seeks to identify a *problem*, i.e. some deficiency or opportunity for improvement, that a new system family may solve. This should not be just any problem, but something that would mean considerable improvement and thus have sufficient value for some stake holders to justify an investment in the new system. Thus the core of the task is to understand needs and find solutions.

The concrete results are domain descriptions, system studies and specifications for system families that may create future businesses. A business is created when a product and a production meets a market. Thus, commercial success depends on more than the prod-

uct qualities. It also depend on the marketing and the production. Consequently it is recommended to develop the product, the market and the production in parallel.The domain relates clearly to the market, and the instances to production. In this way TIMe supports integrated development of all three areas.

Sub-activities

Analysis consists of two activities:

- Analysing domain (p.6-28)which produces domain descriptions. The main purpose is to understand what various stake holders want to achieve, not what the product is. The domain descriptions must be sufficiently complete to: achieve common understanding of terminology and concepts; understand the real needs of stake holders, and analyse improvements.
- Analysing Requirements (p.6-32) which produces:
  - System studies (p.6-17). These are design studies for alternative system solutions. The idea is to asses the feasibility and the business potential of each and to select the most promising alternative for further development, or to stop further development if no alternative is sufficiently attractive.
  - System family statement (p.-86).
  - System family dictionary (p.-90).
  - System family specifications, i.e. Application specification (p.-115), Framework specification (p.-158) and Architecture specification (p.-206) which are used as input to designing the family.

Actors

Analysis require multidisciplinary teams. Key personnel are domain experts such as marketing people and real users, but system developers should also participate both to improve their own problem understanding and to help finding improvements. It is a task where experts in the problem domain meets experts in the solution domain and new opportunities arise out of their combined effort.

There may also be existing literature, existing systems and possibly existing domain descriptions to use. Existing systems is an important source. Consider what parts of the domain they cover. Can competitive advantage be gained if we cover more? or less?

What to do

“How can you possibly find the solution if you haven’t grasped the problem?” (adapted form a wise saying at Ericsson).

This wisdom tells us that understanding the problem should come first: start by describing and analysing the domain, and go from there towards possible solutions. However, when the domain is well defined and the challenge is to gain competitive edge through innovative solutions, the solution comes first.

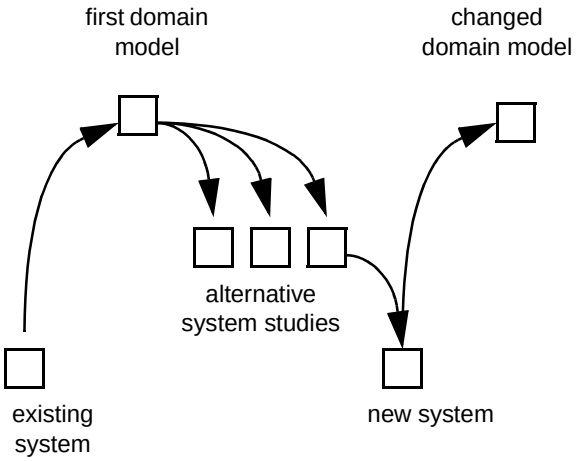
Therefore, the strategy will depend on the quality of existing domain descriptions, and whether the challenge lies in the problem or in the solution.

Iterations

Note that domains are not entirely independent of design concepts or technology. New technology enable improvements that were not feasible and perhaps even not imaginable before. For instance: the access control domain was different before cards were introduced than after. Before there were keys, but no need for person identification. After, person identification is a central issue.

Figure 6-9: Domain - system iterations

Open figure



The old solution was traditional keys that were hard coded according to access zones. With new technology it became feasible to use personal cards and PIN codes instead and thereby achieve more flexibility and security. This change led to a changed domain model where person identification and authentication is a central issue.

In order to get a good start with the first domain descriptions it may be a good idea simply to model the current reality first. If the challenge is to re-engineer existing systems, then first establish a domain description corresponding to the usage of the existing systems.

When initial understanding of the problem domain is established, the next step is to start looking for improvements in terms of new technologies, innovative system solutions or just better ways to organise work. What are the problems today, and what may be improved? Answering these questions and setting goals for improvement is the soul of product planning and should not be taken lightly.

If the improvements have effect on the domain, then make an new or updated domain description.

When the goals for improvement has been set it is time to start looking for technical principles and solutions. We start by Analysing solutions (p.6-36) seeking to come up with alternative system ideas that may be evaluated and compared. How this is to be done depends on the problem, but it will involve making a mixture of object models and property models for the new family (or system instance if it is a one-shot development). These models are called System studies (p.6-17).

In the beginning, these models may be independent of any particular system boundary. In fact it may be a good idea not to identify the system initially, but use the models to analyse what shall be done, and how it may be done. When this is well understood, the next step is to decide what shall be done by the system, and what shall be done by the environment.

The result will be a number of alternative product outlines that may be analysed with respect to technical feasibility, risks, cost and schedules.

Finally, the most promising system concept is selected among the alternatives, and a go/no-go decision is made based on business criteria, available resources and schedules. If it is go, then a specification is developed.

A major goal for analysing is to decide what parts of a problem domain to include or support by a new product. Obviously the domain analysis should be more detailed for those parts. Therefore there is a mutual influence between domain analysis and solution analysis.

Strategies

We define two main strategies: analysing from scratch and analysing from existing domain.

Analysing from scratch

If the problem domain is new or the quality of existing domain descriptions low the following strategy can be used.

Analysing from scratch (new domain)

1. Start by analysing the new domain in order to get the fundamental understanding of the domain and its needs. Use the strategy for Making domain descriptions (p.6-30).
2. Continue by analysing requirements, performing solution analysis and, if a decision to go ahead is made, specifying a system family that will satisfy (a subset of) the needs. Use the Making requirements (p.6-34) strategy.
3. Finally harmonise the domain descriptions with the

It is quite normal that domain models and system family models influence each other. Therefore it may be a good strategy to start analysing solutions before the domain models are completed.

Analysing from existing domain

If the problem domain is well described, and the main challenge is to find new technical solutions, then the following strategy is recommended.

Analysing from existing domain

1. Use the existing domain descriptions as input, and analyse requirements to specify a system family that will either satisfy some new domain needs or provide better solutions than previous systems. Use the strategy for Evolving domain descriptions (p.6-30).
2. Then harmonise the domain descriptions with the specifications if necessary, see Harmonising domain

Analysing domain

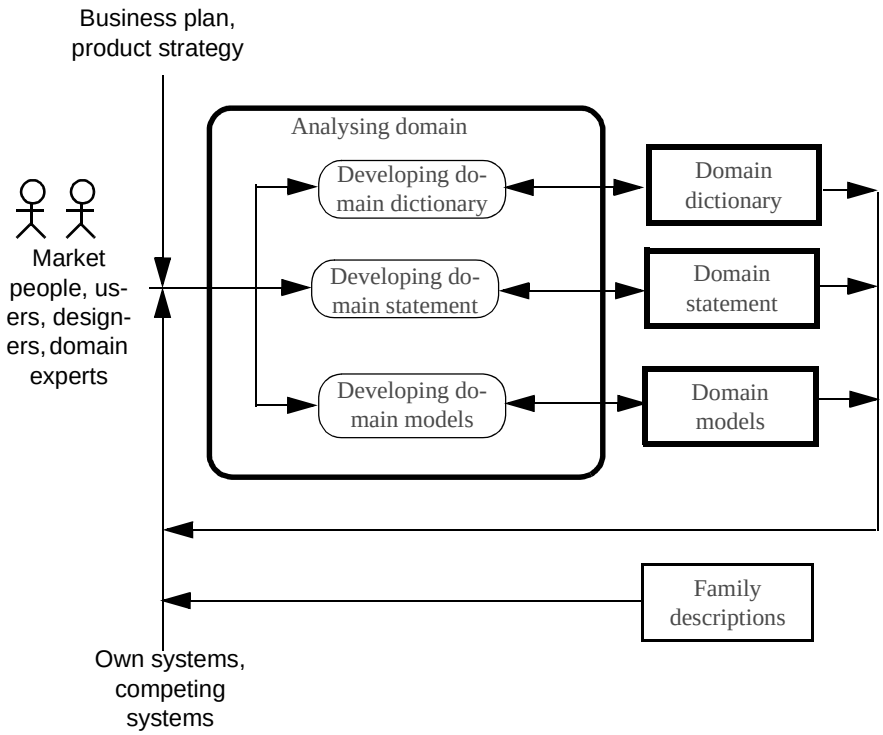
Objectives We perform domain analysis in order to:

- understand the domain and make domain descriptions;
- analyse the domain seeking improvements of value for the stake holders;
- evolve and harmonise the domain descriptions.

What Domain analysis is primarily concerned with the problem domain. In addition to general knowledge of the problem domain, knowledge about existing systems is important. It is important to identify the strong and the weak sides of existing solutions. It may be useful to first model the existing domain including existing systems, and then look for ways the domain may be improved. As a result a future domain may be described. If systems are used in different markets for different purposes, several domains may be described.

Figure 6-10: Analysing domain

Open figure



Sub-activities

We perform domain analysis by means of the following sub-activities:

- Developing domain statement (p.6-46)
- Developing domain dictionary (p.6-51)
- Developing domain models (p.6-62)

Actors

The people to involve are stake holders in the domain such as users and other actors, market people and other domain experts.

What to do

Which sub-activity to start with will be a matter of choice. For a new domain it is recommended to start with the domain statement and the dictionary, and continue with the object models and the property models. For an existing domain, it may be better to start with object and property models. In both cases there will be iterations between domain descriptions and design descriptions.

Making domain descriptions

In this activity we seek to define a domain for the first time. The main purpose is to understand the domain in terms of concepts, objects and relationships and to understand the needs that various stake holders have (the problem). We have no existing system family to consider (but there may be other existing systems in the domain).

Make domain descriptions (new domain)

1. As a first step make the first domain statement. This will normally be a first sketch that gradually will be improved. See, Making domain statement (p.6-46).
2. Based on the domain statement, make the first domain dictionary. This will also be gradually improved. See Making domain dictionary (p.6-51).
3. Then continue in parallel with:
   - Making domain object models (p.6-64) to formally model the objects, classes and relationships in the domain. This will explain how the concepts in the dictionary relate to each other.
   - Making domain property models (p.6-67), to formally model services and other domain properties.
   - Harmonising domain descriptions (p.6-13) (within

Work on the object and property models will normally lead to clarification and more complete understanding which should be used to improve the dictionary and domain statement.

Evolving domain descriptions

Here the purpose is to analyse and improve existing domain descriptions. A typical reason may be that we have revealed imperfections or are looking for improvements that will lead to a new and better domain description or that we need to add details. This will

typically be the case when we plan new products in an existing domain. The strategy will depend on the circumstances, but since we already have a domain statement and a dictionary, it may be a good idea to start with the formal models in this case.

Evolve domain descriptions (existing domain)

1. First update the domain models by:
   - Evolving domain object models (p.6-64) to include the new or changed parts.
   - Evolving domain property models (p.6-68) to include the new or changed properties.
2. Then ensure that the domain descriptions are internally aligned and consistent by Harmonising domain

Summary of dynamic domain rules

Domain modelling approach

- *Establish contact with domain experts and other sources of information about the domain.*
- *Present your results and have frequent discussions with domain experts.*
- *Do not use too much time on problem domain analysis if experiences with making systems in the domain are poor. Iterate with system analysis and possibly system design.*
- *Study existing systems, and make use of existing descriptions and literature about the problem domain.*
- *Iterate with requirements analysis to explore new solutions and to find ways to improve the problem domain. (Remember that some problems may go away, and new opportunities be opened, when new ways are found. E.g. with personal cards, lost keys are not a problem any more.)*

Finding domain objects

- *A problem domain may be very comprehensive. Do not try to model everything that might be considered part of the domain in some wide sense. Concentrate on parts that may be supported by new products.*
- *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.*
- *When systems are defined, use the* Application reference model (p.-99) *to classify the entities into interface, system, and domain specific parts. Generalisations of the domain specific entities should be included in the problem domain.*

Model the actors

- *Be sure to represent each type of stake holder in the domain statement, dictionary and object model.*
- *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.*

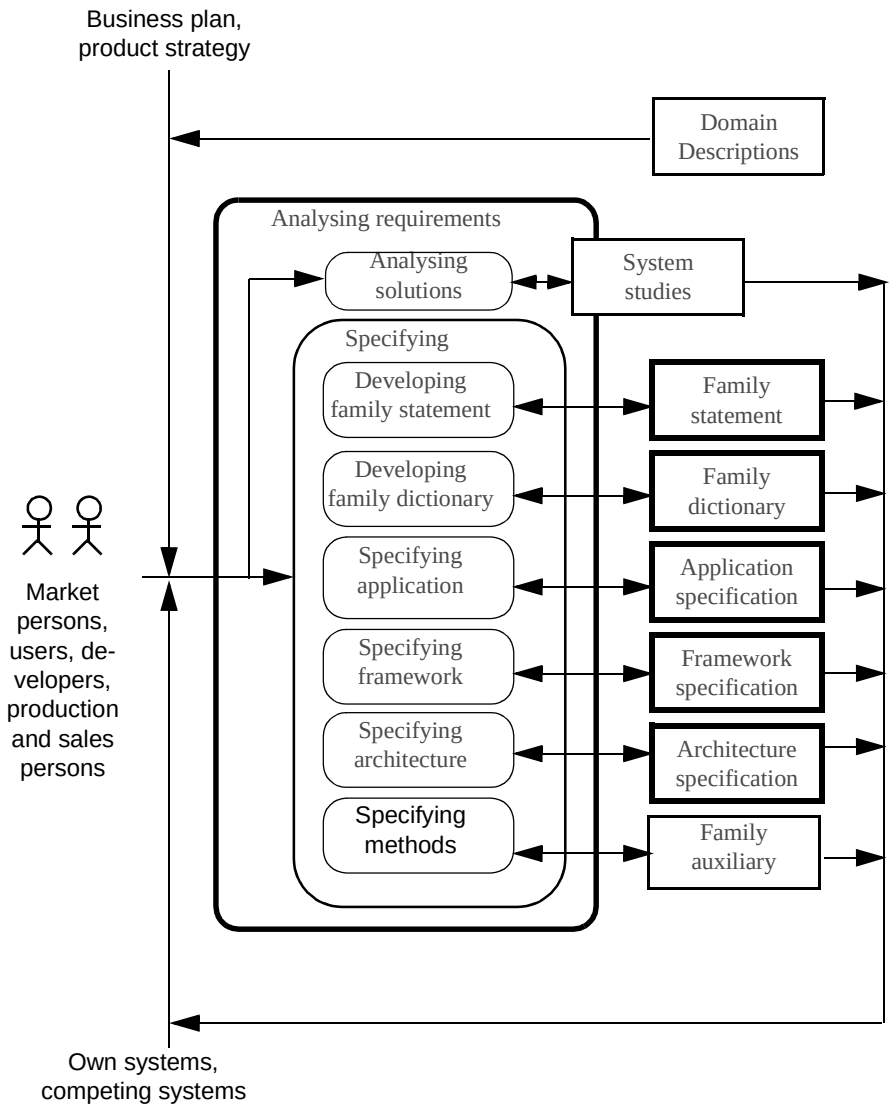
Analysing Requirements

Objectives

Analysing requirements is performed in order to analyse needs and to develop specifications. Not just any specifications, but specifications for system Families with a business potential. domain descriptions where needs for improvement are identified is a good starting point for this activity.

Figure 6-11: Analysing requirements

Open figure



The activity is crucial for the success of a system family as it seeks to answer questions like:

- what parts of the problem domain should be supported by the system family;
- what should be the environment and the interfaces of the system family;
- what should be the positioning properties and the duty properties of the system family;
- what should be the main technological principles;
- what are the requirements to future instantiation and evolution.

What

A central issue is to balance the needs against technical feasibility and cost. To this end, system studies are made where alternative system concept can be evaluated and compared. (These may be seen as mini developments.) If any of the alternatives seems feasible and attractive from a business viewpoint, the best is chosen for further development.

For the selected concept a specification is produced.

It is of course possible to make specifications without (explicitly) analysing the domain and analysing solutions first, but it will be more difficult. Parts of these activities will then have to be done implicitly in the specification activity. With explicit domain descriptions and system studies, the goals for the system development are known, and the task of specification is to express them precisely and detailed.

Actors

Developers, market persons, and stake holders in the domain should all participate. Go/no-go decisions will of course involve management.

What to do

The first thing to do is to make system studies seeking for a viable overall system concept. If such a concept is found the next step is to develop a system family statement and start on a dictionary, and then to develop specifications.

Making requirements

When making requirements for the first time, we assume there are domain descriptions to use as input, but there is no existing family (apart from those that other companies might have on the market).

Make requirements

1. Start by Analysing solutions (p.6-36). This activity explores alternative concepts for a new system family and selects one concept having a good business potential for further elaboration. Once a system concept has been chosen, time has come for Specifying (p.6-36) the new system. Do this by:
2. Making system family statement (p.-89). This sets forth the goals for the new system family.
3. Making system family dictionary (p.-91) to establish the first version of the family specific dictionary.
4. Then specify the system family formally by:
   - Evolve Application specification (p.-139) to express the functional requirements.
   - Making framework specification (p.-183) so the requirements to infrastructure and handling of the application in a framework becomes clear (if anything can be said at this stage.)
   - Making architecture specifications (p.-218) so the non-functional requirements to the implementation becomes clear.
5. Make requirements to methods for code generation, system instantiation and application evolution.
6. Harmonising family (p.6-23) to ensure that the family descriptions

Evolving Requirements

In this case we have existing system family descriptions to start from. We assume the reason for evolution is that some external requirements have been added or changed. (Changes caused by internal iterations are handled by harmonising) In this case too, we start with the formal specifications

Evolve Requirements

1. Start by stating the new requirements.
2. Perform impact study: analyse the new requirements and decide which parts that must be changed. Then evolve the appropriate specifications by:
   - Evolving application specifications (p.-139)
   - Evolving framework specification (p.-184)
   - Evolving architecture specifications (p.-219)
3. Then update the family statement by Evolving system family statement (p.-89),
4. and the dictionary by Evolving system family dictionary (p.-92)

Summary of dynamic requirements rules

Requirements analysis approach

- *This is a creative activity, so creative techniques such as brain storming may prove useful.*
- *At an early stage do not be afraid of unconventional solutions. On the contrary, look for them.*
- *Remember that the purpose is to find a new and better way to satisfy the needs of stake holders. Therefore, close interaction with stake holders and domain modelling is strongly recommended.*
- *The main purpose is not to find design solutions, but to determine the external properties and interfaces, in particular which services the system are going to provide. Therefore the focus should be on context, and not content.*
- *Elaborate content only where this is necessary to assess the feasibility or to estimate cost an risk. Be prepared to reconsider content that is sketched at this stage.*
- *Use prototyping when this is cost effective, i.e. when the additional cost of a prototype is justified by improved communication, decision making and/or reduced risk.*

Initial requirements

State explicitly the project and product requirements. These requirements are often requirements to the tools and languages of the project.

Analysing solutions

What

The purpose of analysing solutions is too consider the needs and how they may be satisfied. It is an interplay between the what and the how.

Inputs to the activity are needs and domain descriptions. Central decisions to be made are what needs to satisfy, in particular which services to provide, and what technology to use.

The activity is the first step in turning needs into solutions. It can consist of many mini-development projects with the purpose of exploring alternative solutions. The activity may therefore produce a number of alternative system Sketches. The detailing of these sketches depends upon the knowledge about the desired system and the risks involved.

Feasibility study

For each system concept:

1. Selecting services to support.
2. Sketching the environment.
3. Sketching the application.
4. Sketching the architecture.
5. Identification of and assessment of risk and critical issues.
6. Assessing cost and schedules.
7. Evolution and maintenance.

When some promising alternatives have been sketched they shall be compared and the best selected. This forms the input to make family statement and Make Specifications.

Specifying

What

The main inputs to this activity are informal statements about needs, the domain descriptions and system Studies.

Outputs are precise specifications expressed on the relevant abstraction levels. The focus is on properties and context. The challenge is to structure the input information, find missing detail and organise the specification. Clarifications and iterations will be needed.

Actors

Specifications are developed by development persons in close cooperation with users and other stake holders. System and market departments should also take part, if not in the detailed work, at least in major decisions.

What to do

Turning needs into specifications is not always straight forward. On one hand it is necessary to clarify the needs in order to make the specifications sufficiently precise and detailed. On the other hand it is necessary to consider technical feasibility, economy and schedules. Finally it is necessary to consider how each particular requirement will interact with other requirements. An iterative process is required where users and other stake holders take actively part.

It is assumed here that specification is proceeded by system studies where an overall system concept has been selected. It is clear roughly what parts of the domain that shall be covered by the system family, and what properties it shall have. The goal is to state this more precisely and with more detail.

For a general approach to making specifications more precise and detailed, see The dialectics of refinement.

Making specifications

When specifications are developed for the first time, the following strategy may be used.

Make specifications

1. Making system family statement (p.-89). This sets forth the goals for the new system family.
2. Making system family dictionary (p.-91) to establish the first version of the family specific dictionary.
3. the specify the system family formally by:
   - Making application specifications (p.-137) to express the functional requirements including services.
   - Making framework specification (p.-183) so the requirements to infrastructure and handling of the application in a framework becomes clear (if anything can be said at this stage.)
   - Making architecture specifications (p.-218) so the non-functional requirements to the implementation becomes clear.
4. Make requirements to methods for code generation, system instantiation and application evolution.
5. Finally ensure that all descriptions are aligned and

Evolving specifications

When new requirements lead to changes in specifications, follow this strategy:

Evolve specifications

1. Start by stating the new requirements.
2. Perform impact study: analyse the new requirements and decide which parts that must be changed. Then evolve the appropriate specifications by:
   - Evolve Application specification (p.-139)
   - Evolving framework specification (p.-184)
   - Evolving architecture specifications (p.-219)
3. Then update the family statement by Evolving system family statement (p.-89),
4. and the dictionary by Evolving system family dictionary (p.-92)

Summary of dynamic specification rules

Incremental specifications

- *In order to achieve flexibility in the development process and the product, try to make the specifications modular so it is as easy as possible to add or change specifications.*
- *If flexibility is important, express this as clearly as possible in the specification (in the Auxiliary descriptions, in the statement or in textual annotations to the models).*
- State clearly the requirements to methods for (incremental) evolution, code generation and instantiation.

Designing

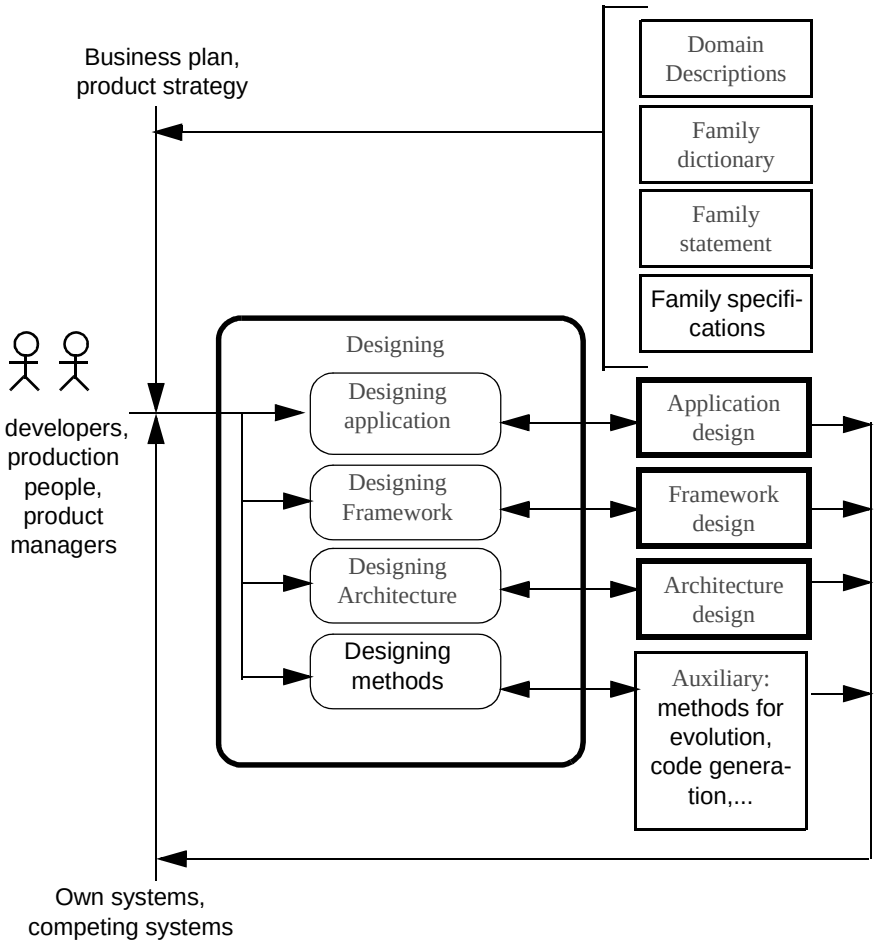
What

The main inputs to this activity are the formal specifications: Application specification (p.-115), Framework specification (p.-158) and Architecture specification (p.-206).

The main outputs are the corresponding design models:Application design (p.-119), Framework design (p.-160) and Designing architecture (p.-219). In addition one should consider how implementations shall be generated, how instances shall be produced and how evolution is to be done, in particular how services shall be added. This is described as part of the Family auxiliary (p.-96) descriptions.

Figure 6-12: Designing

Open figure



Designing is not a monolithic activity. It is composed from activities that each require considerable effort, especially when developing from scratch:

- Designing applications (p.-140) which synthesises the application design from the application specifications, and verifies that the specifications are satisfied by the design. Both when making and evolving a design it is recommended to take existing components into account (designing with reuse) and to keep a close relationship

between service roles, interface roles and design objects (service orientation). When a new object behaviour must be designed, try to synthesise object behaviour from service and interface roles.

- Designing architecture (p.-219) which synthesises the architecture design from the architecture specification and the application design. The goal of this activity is to define an implementation architecture which will implement the application and satisfy the non-functional requirements. This activity will often require a cut and try approach, where an architecture is suggested, and then evaluated with respect to performance, error handling, and other non-functional properties. As platforms become more standardised, this activity can focus more on the application specific architecture.
- Designing framework structure (p.-185) and Designing framework behaviour (p.-189) which takes the infrastructure required by the implementation architecture into account and define a framework. It also verifies that the specification part is satisfied. Once the framework has been defined it is likely that the application must be harmonised so that it uses the infrastructure part of the framework. It should be noted that a framework is useful only when there is an infrastructure to consider and may be omitted in other cases. It may also be the case that the framework is omitted in the first system development, but introduced later in order to better support evolution and production.
- Designing methods which define methods for the future handling of the system family: methods for automatic code generation, methods for system instantiation and methods for system evolution, see Family auxiliary (p.-96). Such methods are not needed for one-of-a-kind systems, but essential for cost-effective product handling. An important goal of TIMe is to free resources that traditionally has been tied up in maintenance, evolution and production activities.

Implementing

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.

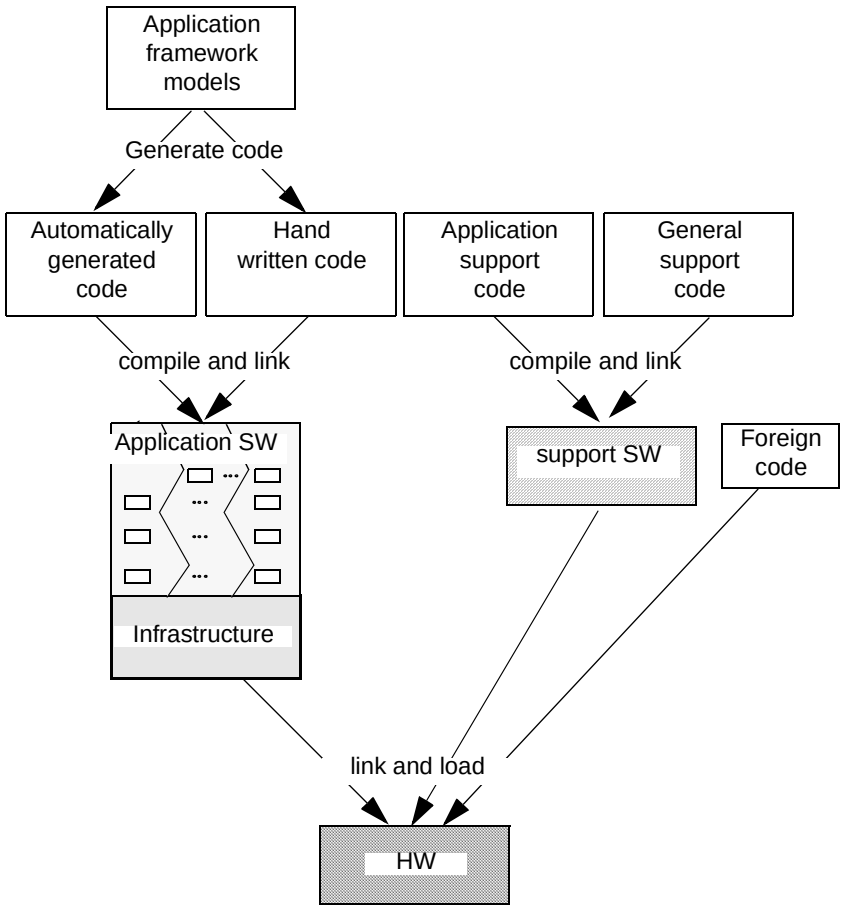
What to do

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.

Figure 6-13 (p.6-42) illustrates some aspects of implementations. Note that the code generated from application frameworks will interface with code coming from different sources. To produce a concrete system these various parts of code must be linked together and loaded on the hardware.

Figure 6-13: Software implementation

Open figure



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. 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.

It should be noted that even if automatic code generation is used, there is likely to be some hand generated code (e.g. for input/output drivers) and possibly legacy code from existing solutions. It may also happen that different parts are generated using different tools, e.g. SDL tools for the control parts and UML tools for database parts.

Consequently there are many issues to consider, and many tool problems to solve before the implementation activity is well defined for a new system family. However, once defined it may be used over and over again to produce new implementations.

Instantiating

In the instance area of concern, the main thing is to configurate and to build a system instance. This can be done both on the abstract level, using SDL, in the implementation architecture, and in the implementation. The common practice in most companies is to do this on the implementation level using configuration files and tools like Make. An alternative is to use special configuration languages in this area.

Instantiation

Objective

System instances are what companies sell and customers buy. The purpose of instantiation is to produce system instances from family descriptions and instance configuration descriptions.

What

In the instance area of concern, the main thing is to configurate and to build system instances. 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.

Domain statement

Objective The domain statement leads to the very first understanding of what this problem 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.

What A (problem) domain statement is a concise description of the problem domain with focus on stake holders and their needs, the essential concepts, functions and work processes, rules and principles. It should also clearly state the nature of the problem, i.e. what one tries to achieve.

It will normally be sufficient to express the domain statement informally using natural language and drawings, but one should try to be as clear and precise as possible.

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.

The domain statement may also contain required properties if these exist at this point of time, but for an initial development of a new system these may be vague or non-existing.

Domain statement outline

Executive summary

This should be a very brief summary with focus on the key issues. It may well be written in a style directly suitable for market purposes. Stake holders in the domain should immediately recognise and accept the description.

Area of concern/context

High level description of the area of concern or the context of the domain. E.g. the interbank financial market, the security in buildings area.

Stake holders

Stake holders are persons or institutions with direct or indirect interest in the domain: companies, users, operators, owners, etc. The various stake holders have responsibilities and tasks to perform that give rise to needs. Therefore the domain statement should mention every class of stake holders, their overall objectives, needs and responsibilities. The Actors are of course important, but do not forget other stake holders that have a more indirect interest (often economic), e.g. managers or owners.

Subject entities

These are all the entities that are manipulated, represented or controlled in the domain:

- Passive entities that need to be represented as data, and their associations.
- Manipulated entities that are transformed or handled in the domain such as : materials, commodities, assets.

- Controlled entities, such as some machinery, that are under control by other entities in the domain.

Be sure to mention every category of subject entities and explain their role in the domain.

Helpers

These are the entities that are used by the actors to perform services and transformation processes. Be careful not to make the helpers too system specific. Try to capture the general features needed irrespective of particular systems.

Services

List all the main functions, their purpose, which objects that collaborate and describe how they are performed in terms of textual use cases.

Work processes and materials

If the domain contains non-trivial transformations, describe each transformation stating its inputs, outputs and constraints. This is particularly useful for material transformation processes, e.g. to pick potatoes or peel scrimps.

Rules and principles

State general rules and principles that apply to the domain as a whole or to specific parts of the domain.

Trends

If there are trends in how the domain will develop: describe them.

Existing systems

Brief summary of systems that are common to use in the domain, their strengths and weaknesses.

Problems - improvements needed

Identification of problems or shortcomings that need to be solved.

Domain statement relationships

Within domain

Every domain term used in the domain statement shall be defined in the dictionary.

Every type of actor and entity shall be represented in the object models under the same name as in the statement.

Every service and transformation shall be represented in the property models under the same name as in the statement

With family

No particular constraints.

Harmonising domain statement

Make sure the relationships with other domain descriptions are satisfied. Make an entry in the dictionary for every domain specific term . Important objects, classes, relationships and properties in the domain models shall be mentioned in the statement.

Developing domain statement

Objective To collect domain knowledge and make a Domain statement (p.6-44), and to keep the domain statement updated and in harmony with other descriptions

What The source information will normally be fragmented, informal and imprecise by e.g. not using the same terminology or by stating the same required property twice, but with a slight difference. It may also be unbalanced in its coverage of essential versus not so essential information. This activity, together with the one making the dictionary, shall transform these informal specifications to a more precise, yet informal, specification, where the essential parts are emphasised.

Actors As for the domain at large.

Making domain statement

This activity may well be subdivided according to the topics of a domain statement, for instance like this:

Make Domain Statement

1. Describe area of concern/context
2. Describe stake holders
3. Describe subject entities
4. Describe helpers
5. Describe services
6. Describe work processes and materials
7. Describe rules and principles
8. Describe trends
9. Describe existing systems
10. Describe problems and improvements needed
11. Make executive summary

Make executive summary

This part is best written after the other parts of the domain statement. Consider what is the essential message of the problem domain descriptions. Express it in a way everybody can understand. Make it short (half a page).

Describe area of concern/context

Try to state concisely what the area of concern is about and what its environment is.

Describe stake holders

Ask yourself who has an interest in this domain? Try to identify every type of stake holder/person-role in the domain and describe their overall responsibilities and tasks. Note down what kind of problems they have and what kind of improvements they would consider valuable.

Describe subject entities

Try to identify all passive entities (including the stake holders), manipulated entities and controlled entities. Explain their roles in the domain and how they are connected and related to each other. Do they communicate? How? Consider only entities and associations that are relevant to some services or work processes in the domain.

Describe helpers

Are any helpers needed to perform the services? Try to abstract from particular system solutions and identify the general features needed. Note that the helpers are likely candidates for improvement. If it is very difficult to generalise, then describe the existing helpers, and consider improvements afterwards.

Describe services

List all services (functions) needed and give a concise description of what each does. Again it is important to abstract as much as possible from how services are implemented in existing systems. Try to focus on what essentially needs to be done.

Describe work processes and materials

For each material transformation, give a concise description of what it does. What are the inputs and the outputs? What are the sub-transformations? What are the constraints? Note that the purpose is not to describe technical solutions, but to clarify the goals.

Describe rules and principles

Rules that regulate the problem domain should be referenced, as should general principles. You will have to consult domain experts to find out about these issues. Be aware that law may severely constrain the technical solutions that are available, as well as it may create new business opportunities.

Describe trends

Important trends regarding functionality, work processes, regulations and technology should be stated.

Describe existing systems

The purpose here is to briefly explain the main features of existing systems with emphasis on their strong and weak points. Consider also the competition here. This will help to identify duty bound and success properties for new products.

Describe problems and improvements needed

Analyse the domain described so far and try to identify the problems with existing solutions and how they may be improved. Interviews with domain Actors and other stake holders will be important here.

Evolving domain statement

When new insight or new requirements leads to a modification of the domain descriptions, then the domain statement shall be updated accordingly. This activity follows from other activities in domain analysis or solution analysis.

Note that the domain shall be relatively stable compared to the system solutions. It shall not be updated so often. However, a new domain is likely to be updated more frequently than a well established one.

Summary of dynamic domain statement rules

Consistent domain statement

- *Check that the terminology is internally consistent, using the same terms for the same entities throughout the statement.*
- *Check that the statement is unambiguous, that each term has only one meaning.*

Maintainable domain statement

- *Check that the statement is maintainable. i.e. modular and without redundancy that makes it difficult to maintain.*

Domain dictionary

Objective

The objectives of a domain dictionary are to define terminology and thereby enable precision and efficiency through:

- common use of terminology;
- improved communication and coordination;
- common understanding.

What

A dictionary is a reference book listing words or terms and giving information about a particular subject or activity *(Collins 86)*.

A dictionary in TIMe need not be a book, but it shall contain a list of terms with an explanation of their meaning.

An informal Domain statement (p.6-44) is normally not sufficient. We will need a more precise definition of the most important phenomena and concepts and the corresponding terminology. We therefore improve our understanding by listing the concepts in a dictionary. For some domains, dictionaries are readily available, but for other areas, an important task is to define one.

A dictionary is based upon the three aspects of concepts: designation, intention and extension. The designation is the entry in the dictionary and the explaining text is a description of the properties that phenomena shall have in order to belong to this concept. The extension are all the phenomena covered by the concept.

Producing a dictionary adds to the understanding of the subject being analysed. Furthermore, it will help people to communicate more precisely. A dictionary helps to bridge the gap between people with specific knowledge of the application area and people new to the area. Later in the development the concepts in the dictionary will often find their way into the system description as types (of objects). We suggest that the dictionary should be maintained along with the other permanent documents.

Be aware of the difference between concepts and sets. A set has cardinality, and a given entity may or may not be member of the set. A concept is not a set of phenomena, but just a definition of the properties of the corresponding phenomena. This should be reflected in the wording of the dictionary - the reason is that the domain may have real sets.

Domain dictionary content

The dictionary may simply be an alphabetic list of terms with informal explanation. All terms used in the problem domain should be included, such as :

- terms for phenomena and concepts;
- terms for entities;
- terms for properties;
- terms for services;

- terms for relationships;
- terms for connections and interfaces;
- terms for processes;
- terms for roles in particular human roles like users, operators etc.;
- terms for materials.

TIMe may, in later versions, provide a template for dictionaries.

Relationships

Within domain

There is a more or less direct road from a dictionary to an object model. Concepts are represented by classes: the name of the class corresponds to the designation and the class definition to the intention of the concept. objects according to the class represent the extension. Therefore the names used for object types shall be the same as the name used for the corresponding concept in the dictionary.

With family

The domain given terminology used in Families shall be the same as in the domain dictionary. It is possible to define additional terminology in separate Family dictionaries.

Harmonising domain dictionary

Within domain

Ensure that the relationships with the other domain Descriptions are maintained. See Relationships (p.6-50).

With family

Ensure that the general domain terminology is applied in the Families. If family specific Dictionaries are developed, they should refer to the domain dictionary for all domain given terminology and avoid redundant definitions (which may develop into inconsistent definitions).

Summary of static domain dictionary rules

Domain dictionary entries

- *Represent each essential problem domain phenomena and concept by an entry in the dictionary.*
- *For entries in the dictionary that correspond to concepts that will be represented directly by types (classes), it may be a good idea (if this is known) to use the same name on the type as the designation of the concepts.*

Update d domain dictionary

- *Keep the dictionary updated throughout the development. If desired classify the entries as coming from analysis or design, domain, environment or system. This may help in updating the dictionary and also to answer questions like "Is this phenomenon covered by the domain of the system?" or "Is this type of entity handled by the system?"*

Domain specialisation hierarchies

- *For each concept in the dictionary (and the corresponding* Developing domain object models (p.6-63)*), ask whether all the objects that fall within the extension of the concept/class have the same properties. If they have not, find specialisations, which may or may not extend the dictionary. During the specialisation, extend the description in the dictionary with properties which add to the understanding of the concept.*

Developing domain dictionary

Objective To make a Domain dictionary (p.6-49).

What The activity makes an entry in the dictionary for each term in problem domain.

In general, the dictionary comes from the other domain Descriptions, as its purpose is to define the terminology used there. Thus, the dictionary is developed more as a spin-off from making the other descriptions, than as an independent activity.

When object models and property models are developed, the dictionary should cover all the objects/types, associations and properties represented in those models.

Actors Same as for domain Analysis at large.

Making domain dictionary

The dictionary is made for the first time as part of the first domain analysis, see: Making domain descriptions (p.6-30).

The first dictionary comes from a (prose) Domain statement (p.6-44). By studying the nouns in the Domain statement (p.6-44) we can make the initial dictionary. But the dictionary shall contain more than the nouns, it shall also include services or functions and associations which may be visible in the domain statement as verbs.

A practical approach is to analyse the text of the domain statement and mark every term that refer to a domain specific concept or phenomena.

For each term thus marked, make an entry in the dictionary which define the term.

editorial 1. should the dictionary be organised in some way? e.g. according to the domain statement categories, according to the kind of entity (concept, property, entity, relationship, service)

2. should the dictionary contain references to object models and property models?

Evolving domain dictionary

The dictionary is evolved as part of the evolution of domain Descriptions, see Evolving domain descriptions (p.6-30). Changes may be due to evolution of the domain statement as well as the domain models. Whenever existing terms are modified or new terms are introduced in any of the other domain Descriptions, the dictionary should follow up.

Summary of dynamic domain dictionary rules

Finding domain dictionary entries

- *Develop the dictionary as a consequence of other domain descriptions.*
- *Use the dictionary actively as a source of terminology when making the other domain descriptions.*
- *Do not add new terms to any model without checking that there is no appropriate term in the dictionary already.*
- *Analyse each entry checking that it is precisely and unambiguously defined. Look for similarities among entries. Avoid to use the same term for different things, and to use different terms for the same thing. If necessary define synonyms explicitly. Clarify relationships between related entries, e.g. subtype relationships.*

Domain models

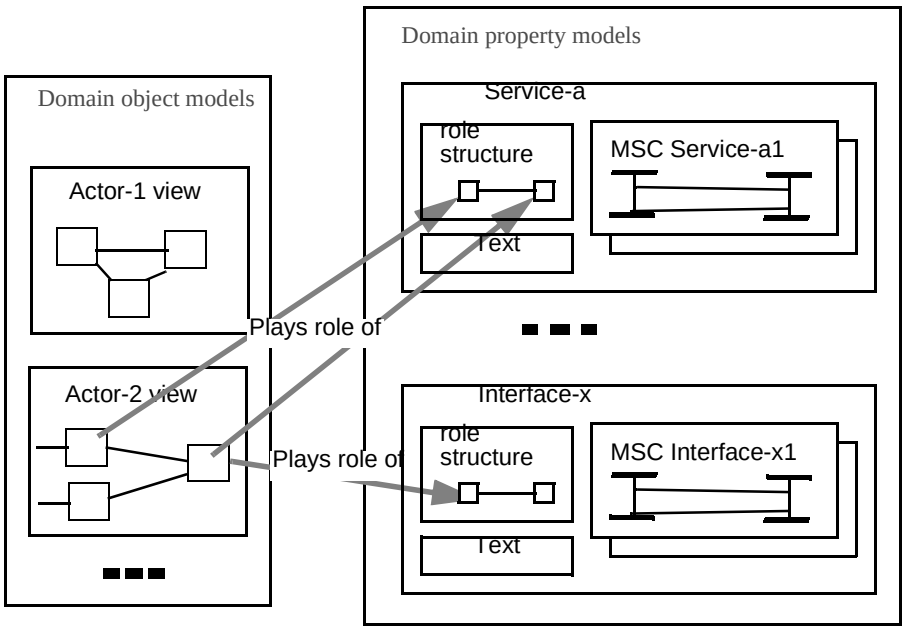
Objective The purposes of domain models are:

- To describe formally how objects and properties in the domain are related.
- To define generic types that may be used in many system families and system instances serving the domain.

What Domain models are composed from Domain object models (p.6-54) and Domain property models (p.6-60), and organised as illustrated in Figure 6-14 (p.6-53).

Figure 6-14: Domain Models

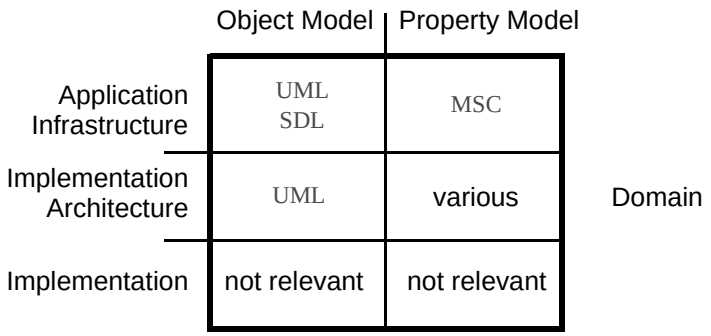
Open figure



Domain models will mostly be application abstractions. But the possibility to model more concrete aspects (e.g. infrastructure and platform) will not be excluded. It may for instance be useful to model various technical solutions which are common for the problem domain. It is perfectly relevant to make more concrete object models whenever there are concrete matters in the domain needing to be described, e.g. physical materials, concrete interfaces. Implementations are not part of domain models.

Figure 6-15: Domain model notations

Open figure



Domain object models

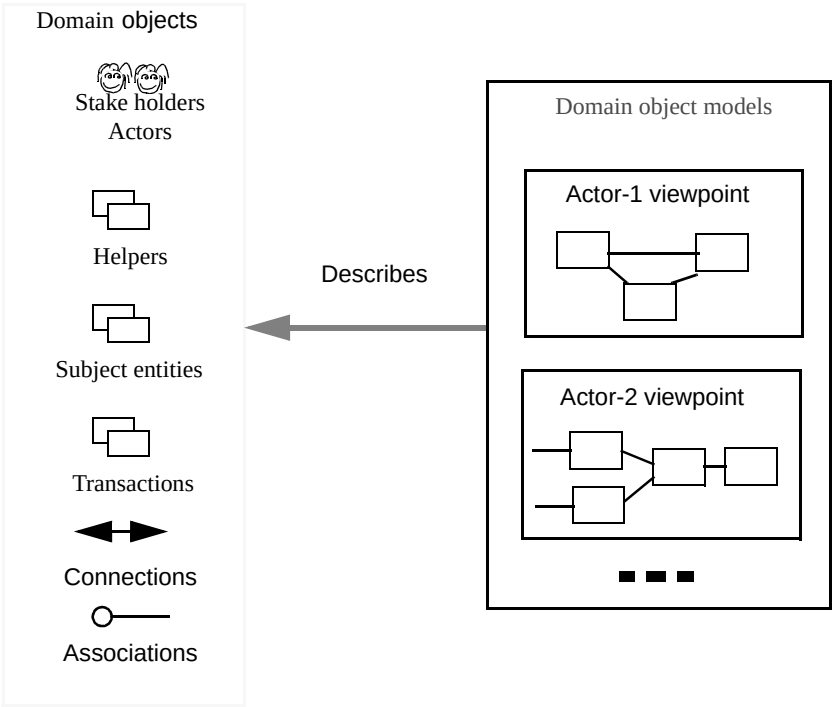
Objectives Domain object models serve:

- To improve understanding and communication by rigorously describing how concepts and objects in the domain are related.
- To give a more precise meaning to terms in the Domain dictionary (p.6-49).
- To clarify the basic needs existing in the domain.
- To promote reuse by describing objects and classes that are common to most systems in the domain.

What Domain object models (p.6-54) describes the problem domain from an object oriented perspective.

Figure 6-16: Domain Object Models

Open figure



It defines types which represent concepts in the problem domain, and objects which represent phenomena in the problem domain. It defines the attributes, the operations and the behaviour of objects as well as associations and communication links between objects. It shall be complemented by Domain property models (p.6-60).

Domain object models will describe active objects as well as passive objects.

Active and passive objects

Objects of the real world are by nature active. They interact with other objects and have a dynamic behaviour. When we consider the domain as a part of the real world, all objects are potentially active. However, we do not need to model them all as active objects in the domain models. Some may be active, some may be passive and some may be both. This is a decision we make in the models.

At some stage we must decide the level of detail and the kind of behaviour we will specify for each object type. For active objects, we specify a “real” behaviour, whereas for passive objects we specify a “data” behaviour which is very different from the “real” behaviour. Considered as an active object an AccessPoint in the Access Control domain have a reactive behaviour that interact with real users. As a passive object, represented in the validation database, is has a simpler behaviour that responds to validation queries.

Normally there will be only one active object representation of a domain entity, but there may be several passive object representations. The behaviour, the attributes and the context may be very different in each case. Therefore we may need to develop one active and several passive component models representing the same domain entity.

The advantage of the object models over prose descriptions are that they show the relationships between objects in a rigorous way, and that they use a graphical notation.

The domain object model is not only made in order to get a better understanding of the domain. It is also made to identify types that are common to all systems in the domain. It is a goal to describe at least every object type that will be part of the domain given parts of systems and system environments. It may be difficult to find all of those without considering any system. It will normally help to study existing systems, and also to work on the new system design. Consequently, when complete, the domain object models will be influenced by existing systems as well as the design of new ones.

However domain object models may well include more objects than those that will eventually be in systems or system environments. They should also describe related objects that are important for the purpose of systems in the domain.

Components for reuse are supposed to be identified in this activity, and correspondingly components identified in earlier domain object models are used here. Classes of objects that are identified as belonging to the problem domain will have more chances for being reused in other systems than classes being made in the design of a specific system. Types (classes) identified in the domain object model are therefore supposed to be made as general as possible, and there may even be sub-activities that have this as their main purposes: to prepare for reuse. Apart from this, TIMe believes in the fact that reuse comes after a successful first use.

Since the domain as we consider it, is a generalisation it may be modelled as a type (in SDL) or a class (in UML). It will normally contain components that are defined with reference to other types (SDL) or classes (UML), see Anatomy of object models. In most cases these component types are more useful than the domain model itself.

There are several reasons for this:

- In many cases the domain is not well defined enough or contains too much variability to capture in a single model. A collection of component types is less restrictive and more flexible. They provide more information than can be said in a single model.
- Within the domain there will be many different views that we need to understand and reconcile, e.g. the views of different actors. Component types offer a formal way to describe the relative views of each component type through its context expressions.
- Reuse will be in terms of component types and not complete domain models.

For this reason we recommend the following rule

Make component type models

- *Make a type model showing the context and content of each actor, helper and subject entity.*

In order to get a better overview, one should try to put the components together in complete or partial domain models where possible. A useful compromise may be to develop several viewpoint models.

Consequently, domain object models will normally contain two main parts:

- *Domain structures* which give a structural overview of the domain representing all objects, connections and associations. It may be a single model, or it may be composed from several viewpoint models. In the case of a single model, the entire domain is viewed from the same angle (somewhere “above” the domain), see Figure 6-17 (p.6-57). In the case of viewpoint models the domain is seen from different angles (from some “side”). Viewpoint models may be simpler to make since they may dis-

card aspects that are not relevant from that angle. At the same they allow us to make the differences between different viewpoints more clear. UML supports viewpoints by using Packages.

- *Component type models* which define each component type (class) separately with context and associated properties. These models serve to give the relative views that each component type has on the domain, see Figure 6-18 (p.6-57).

Figure 6-17: The access control domain

Open figure

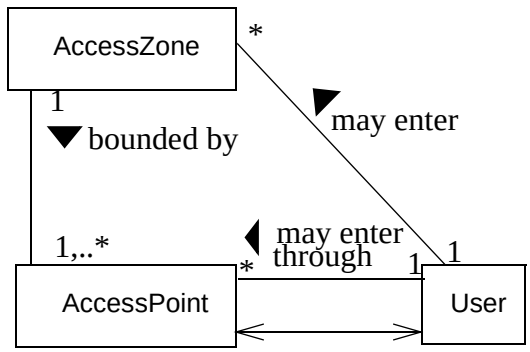
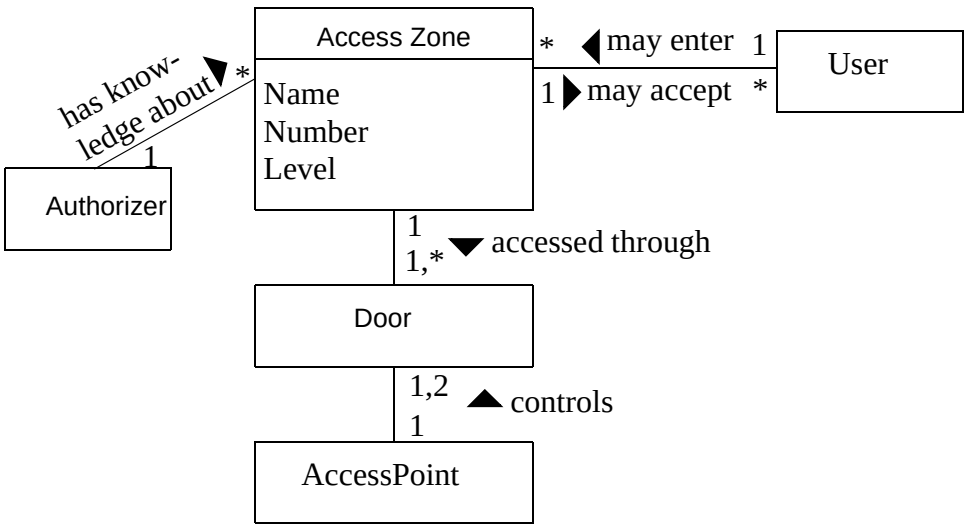


Figure 6-18: The class definition of Access Zone

Open figure



Domain object modelling belongs to a subset of more general object modelling. Object models can be described using either UML or SDL, depending on whether they are domain or design object models and depending on the formality needed.

Domain object model languages and notations

Object models are expressed using UML or SDL. It should be noted that UML is designed primarily for passive objects, while SDL is designed for active objects.

Domain object notations

- *Use SDL for abstract component models where (concurrent) state transition behaviour is central. If associations and passive objects are important as well, make supplementary models in UML. Use:*
  - *Block types where there are concurrent behaviours;*
  - *Process types where there is only sequential behaviour.*
- *Use UML for component models where associations are central and state transition behaviour less central.*
- *Use UML for structure models where the nature of components is association heavy or unknown.*
- *Use UML for concrete models.*

Make separate object models for active and passive objects?

Abstract domain object models

Domain object models are abstract models unless otherwise stated. They are abstract in two senses:

- they model the world without considering the physical details of entities;
- they model parts that will be realised in systems without going into system specific detail.

The abstract models will normally concentrate on the application abstraction, and only consider infrastructure in special cases where the infrastructure is general for the domain. Frameworks are normally not considered as they belong to the family area.

applica-tion:

At this level we find the bulk of domain object models. Every actor, helper and subject entity should be represented in a component model.

In addition the domain structure may be represented in a number of viewpoint models or in a single model.

Infrastruc-ture

This level will consist of component models for infrastructure components that are common to the entire domain. System and family specific components should not be included.

Make component type models

- *Make a type model showing the context and content of each actor, helper and subject entity.*

Concrete domain object models

Implemen-tation architecture

At this level we find models for platform components that are general for the entire domain.

models at this level are only made if the domain is about platforms, or if the platform is standard for the domain.

Domain object model relationships

Within domain

- Every object type shall be mentioned in the dictionary under the same name.
- Every object type shall have associated property models.
- Object models refer to properties through service names and role names.
- The association between objects and properties are maintained through service names and role names.
- The functional properties specified for an object shall be satisfied by the object behaviour, i.e. be a proper and valid projection.
- The behaviour of every object type shall be strongly input consistent in every role.
- Every object instance shall be valid in all assigned roles.

With Design

- Domain given design objects shall be related to domain object through inheritance or implementation relations.

Harmonising domain object models

In this case the domain object model is harmonised within the domain and with design.

Within domain

Ensure that the rules for domain internal relationships are correctly maintained:

- Every type name shall be defined in the dictionary.
- Every object type shall have associated property models.

With Design

Ensure that the rules for Design relationships are correctly maintained:

- Every domain given object in design shall be related to an object type in the domain object models. The relationship shall either be pure inheritance or an implementation relation.

Summary of static domain object model rules

Use the general rules for object modelling and the following special rules:

Make component type models

- *Make a type model showing the context and content of each actor, helper and subject entity.*

Domain object notations

- *Use SDL for abstract component models where (concurrent) state transition behaviour is central. If associations and passive objects are important as well, make supplementary models in UML. Use:*
  - *Block types where there are concurrent behaviours;*
  - *Process types where there is only sequential behaviour.*

- *Use UML for component models where associations are central and state transition behaviour less central.*
- *Use UML for structure models where the nature of components are association heavy or unknown.*
- *Use UML or SOON.hs, SOON.ss for concrete models.*
- *Make separate object models for active and passive objects?*

Domain property models

Objectives To improve understanding and communication about the domain by rigorously describing the properties applying to domain objects, services and interfaces.

What Domain property models are 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 described using text, role structures and MSC, see Figure 6-14 (p.6-53).

Text is used to give a textual explanation of a service or interface. Role structures are UMLinstance diagrams that represents 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 specific) system objects. Properties belonging to the domain will be candidates for properties of several system in the domain.

It is an implicit assumption that models of the domain will not cover design dependent properties. Such properties will be added in design models. However, the domain may well cover concrete properties as long as they are general for the domain and not specific to particular systems or families. Domain properties shall be related to domain objects if they can, but it is allowed to describe properties without reference to objects. One example is roles where the actor objects are unknown or irrelevant in the domain. Another example is general properties applying to the domain at large. property descriptions will typically be quite fragmented.

Abstract domain properties

Application: Services: Text, role structure diagram and UseCases in MSC.

Interfaces: Text, Role structure diagram and Use Cases in MSC.

General properties: safety, liveness, user friendliness etc.

Infrastructure Services: as above.

Interfaces: as above.

General properties: as above.

Property modularity

- *Make property models that are as self contained and independent of other properties as possible.*
  Motivation: to enable property (service) flexibility through modular property composition.

Concrete domain properties

Application implementation

General constraints and requirements that are common to the domain:

reliability, safety, modularity, performance, size, technology.

Platform

As above.

Other

About general properties of the environment, materials, processes, etc.

Domain property model relationships

Within domain

- Every service shall be mentioned in the dictionary.
- The MSC instances shall represent either service roles or interface roles.
- Object models shall refer to properties through service names and role names.

With Design

- Interfaces of domain given design objects shall be an implementation of the corresponding domain interfaces.
- Services of domain given design objects shall be implementations of the corresponding domain services.

Harmonising domain property models

Within domain

Ensure that the rules for property-object model relationships within the domain are satisfied, see Domain property model relationships (p.6-61).

Ensure that every service mentioned in the dictionary and statement has an entry in the service lists and corresponding MSCs.

With design

Ensure that the property-property relationships with design are satisfied, see Domain property model relationships (p.6-61).

Summary of static domain property model rules

Property modularity

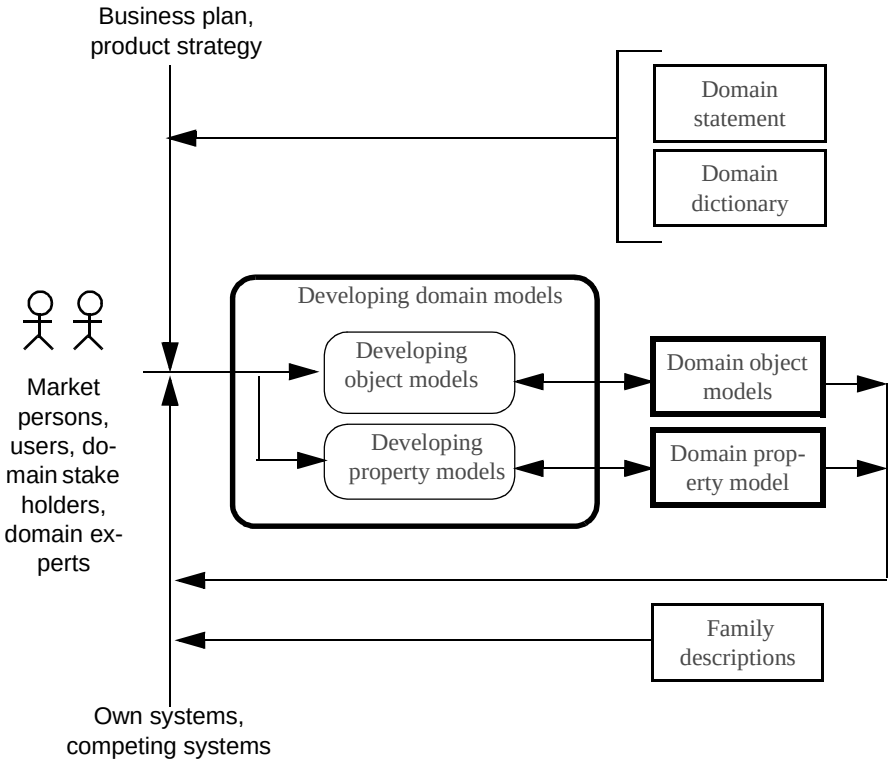
- *Make property models that are as self contained and independent of other properties as possible.*
  Motivation: to enable property (service) flexibility through modular property composition.

Developing domain models

Objective The goal is to make domain object models and domain property models and to keep them aligned and in harmony with other descriptions.

Figure 6-19: Developing domain models

Open figure



The main inputs are domain knowledge that come from people, and general domain knowledge that can be found in the literature. The domain statement and the domain dictionary give important input too. When a system family has been defined, it may influence the domain models. The outputs are the domain object models and the domain property models.

What This activity is composed from two sub-activities:

- Developing domain object models (p.6-63);
- Developing domain property models (p.6-66).

Summary of static domain model rules

- *Switch between object modelling and property modelling, as each of these activities may contribute to the other. Object modelling and property modelling go hand in hand.*

Developing domain object models

Objectives

- to make and evolve Domain object models (p.6-54);
- to analyse the models looking for improvements;
- to keep the models updated and in harmony with other descriptions.

What

This activity produces a (set of) Developing domain object models (p.6-63)s, some of which may be adaptations of existing object models (which will be the normal case during evolution and reengineering).

The main inputs to this activity are existing domain object models (if any) the Domain statement (p.6-44) and the Domain dictionary (p.6-49) supplemented by domain knowledge and the Domain property models (p.6-60). It is also influenced by existing systems and the design of new ones.

As explained in Domain object models (p.6-54), it is possible to model the domain as a single structure, but we recommend to develop viewpoint models and component type models as well. In a single domain structure it is not so easy to distinguish between active and passive objects. Therefore we may start to develop the domain structure without considering whether objects are active or passive. In viewpoint models we are more likely to see differences, and in the component models we must be clear about it. The following guidelines help to identify active and passive objects:

Active and passive domain objects

- *Use connections primarily to link active objects. Therefore objects with connections attached will be active.*
- *Use relations/associations primarily for constructive relations. Therefore objects participating in relations/associations are likely to be represented by passive objects.*
- *It follows that objects with both connections and relations will have both an active and a passive representations in the models*

As an example consider the user of the Access control system, see Figure 6-17 "The access control domain" (p.6-57). The model indicates that there will be two user concepts: active users seeking services, and passive users representing those that are valid users in the domain.

What to do?

Making domain object models

In this case we have no existing object model or property model, only a first domain statement and a domain dictionary. The task is to develop the first domain object model.

Make Domain Object Model (new Domain)

1. Active objects.
   - Look through the dictionary and represent every active object mentioned. Consider every stake holder and be sure to include every actor.
   - Make an object structure showing the interconnection between objects.
   - Identify component types and define each type separately in a diagram where its environment is described, i.e. its interaction links with other object roles.
2. Passive objects:
   - Look through the dictionary and represent every type of passive object mentioned.Consider every subject entity and transaction.
   - Make an (information) object structure showing the classes and associations.
   - Identify component types and define each type separately in a diagram where its environment is described, i.e. its associations with other object roles.

In all steps above, variability, generalisation and specialisation should be considered.

Using this strategy it is natural to develop two models, an active object model and a passive (information) object model. Here the active model will map to active (domain given) objects in the systems while the passive will map to known entities represented by passive (domain given) objects in the systems.

An alternative is to combine active and passive objects in the same models.

Evolving domain object models

In this case an existing domain object model is evolved to take new domain knowledge or requirements into account. Inputs are the existing models and knowledge about the changes to be made. The reasons for change are either:

- an extension of the domain;
- some new services to be added;
- a restructuring due to better insight;

- some major changes in the system technology (such as replacing mechanical keys by magnetic cards).

Note that updates due to system design normally is handled in the harmonising activities.

Evolve Domain Object Model (existing Domain)

1. Express the new features required and analyse the impact on the existing models. Modify the models as required.
2. Active objects.
   - Identify new object types and develop new component type models.
   - Change the component types that must be changed.
   - Update the object structure using the set of component types now available.
3. Passive objects:
   - Identify new object types and develop new component type models.
   - Change the component types that must be changed.
   - Update the object structure using the set of component types now available.

Summary of dynamic domain object model rules

Domain object modelling approach

- *Take the entities that a system must know about and possibly control as starting points for classes of objects.*
- *If working with the domain alone is too abstract, consider in stead systems in the domain and try to generalises from them.*
- *Classes of objects may very well be identified in a process where the required properties of a system are analysed. These may just be lists of requirements, or expressed in terms of Use Cases (MSC), preferably involving people representing the users and people being responsible for the required properties.*
- *Switch between identifying attributes and relations between classes of objects, as it may be a matter of choice if a property is a relation or just an attribute.*

Make component models

- *Take each of the classes and make a model that includes the most important classes of objects in the environment of this class. Identify the constraints on the classes stemming from this model.*

Using domain object notation

- *For parts where the system Design object model is to be specified in SDL, the properties of SDL should be taken into consideration.*
  Examples: As it is known that SDL has single inheritance, it would be a bad idea to make a domain object model in UML with extensive use of multiple inheritance. SDL has a strong notion of real aggregation that depends upon which kind of objects to aggregate (services as part of processes, blocks/processes as part of blocks) - if aggregation is part of the domain object model it should take this into consideration.

Domain object modelling is a special kind of object modelling. In addition to the general guidelines for object modelling; 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 on communication links, unless they are given from the domain statement.
  Communication connections between classes indicates that there will be Interaction models between instances of these. For each of the communication connections check if this is important enough to call for Interaction models.
- Relations
  Do not be afraid to use illustrative relations, but be aware that they may have to be "implemented" during design.
- 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.
- Classes with constraints on the environment
- 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 modelling it will produce a set of packages of type definitions. These will mostly be independent of actual context. If domain modelling go so far as defining system and block types, then apply the general rules of localisation.

Developing domain property models

Objectives

- To make domain property models.

- To analyse the property models seeking to identify problems and find possible improvements in the domain (and thus in systems).
- To keep the property models updated and in harmony with other descriptions.

What

This activity produces the Domain property models (p.6-60), including only properties of entities visible in the Domain object models (p.6-54). Functional Properties that can be expressed as Use Cases by means of function lists and MSC are considered the most important, but Non-Functional Properties are also considered when they are part of the domain. A speciality of TIMe is that objects in the domain object model are not just passive “data objects” - they may also be active objects. If such active objects are identified in the analysis of the domain, then an analysis of their properties also belong here.

Who to involve

The people to involve are the same as for domain analysis in general. However, in order to get the service properties right, it is important to consult actors in the domain, and in particular those that may become users of the systems.

What to do

Consider each of the active object types in all their different roles and describe, by means of role structures, text and MSC, what they need to do or need others to do for them. For each service, identify the roles or actors involved in role structures, and describe the collaborations using MSC.

Be sure to cover all general tasks where systems in the domain are involved. Consider also work situations and tasks to be achieved by means of the systems, i.e. consider a wider context of purposes than the systems in question.

Making domain property models

As part of the Developing domain models (p.6-62) activity this activity runs in parallel with Making domain object models (p.6-64).

The main inputs are the domain statement and the domain dictionary together with the object model developed so far and general domain and system knowledge.

Make Domain Property Models

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. See language specific methodology for details (MSC-92, MSC-96, SDL).
   - *Formalizing (2)*: Those aspects which do not lend themselves easily to descriptions in MSC or SDL should be described in semi-formal prose (see The dialectics of refinement) 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

Evolving domain property models

In this case the starting point is existing domain models and some requirements for new or modified properties. TIMe seeks to support property flexibility by promoting modular property models. However, undesirable interactions may occur between properties and they should be detected and resolved.

Evolve Domain Property Models

1. Make new Property models to represent the new or modified properties.
2. Analyse the impact of the new property models on the existing property models. Consider the property interaction problem.
3. Add the new property models to the total set of domain property models and remove those that are no longer valid. Modify properties that are affected, i.e. where an undesired property interaction may occur.
4. For properties associated with objects, make sure the

Summary of dynamic domain property model rules

Domain property modelling approach

- *Use the domain statement as starting point and consider the needs of all stakeholders.*
- *Use the domain object model as starting point, and consider especially objects that are connected by communication links.*
- *Make a Service List stating all the services (or functions) that needs to be performed and possibly may be supported by systems in the domain. Explain what each service does.*
- *For each service, identify the object roles involved and as far as possible, which objects are the actors of the roles.*
- *For each service, describe the most important Use Cases using MSC. Apply the guidelines of the MSC methodology.*
- *Follow the MSC guidelines for the MSC part. See How to use MSC-92 effectively.*

Follow the rules for MSC usage:

Rule for MSC usage

- *Identify all initiatives. For every initiative; describe the sequences that may follow until the next initiative. Use one MSC for each alternative*.

Domain behaviour properties

- *Describe abstract (behaviour) properties in a way that eases composition into object behaviours and comparison with projections of object behaviour.*
  Motivation: ease of synthesis, quality by construction, ease of validation and verification.

Domain non-functional properties

- *Identify concrete (non-functional) properties that are relevant for the domain models, i.e. properties that any concrete system must abide.*
- *If the system design is considered, look especially for properties that are inherent in the domain and describe them in a general way.*
- *If the system is considered, then it is preferably considered as one object, and it is only decomposed if this helps in identifying properties of the domain objects.*

System family statement

What is a system family statement?

The system family statement is a concise description of the system family with emphasis on specifications, i.e. the external properties. A system family statement is normally expressed in prose, but may well be supplemented by illustrations.

Objective The system family statement is the first introduction to the system family. It serves two main purposes:

- To concisely express the goals for the system development.
- To serve as a top level introduction to the system family.

System family statement outline

Executive summary

This should be a very brief summary with focus on the key issues. It may well be written in a style directly suitable for market purposes. Stake holders in the domain should immediately recognise and accept the description.

How it relates to the domain

Briefly about the domain (with reference to domain descriptions) and what needs in the domain the system family addresses. Which actors in the domain will be supported, and what other stake holders may benefit.

How it relates to the environment

Describe the system context with emphasis on the system boundary and the system environment Explain who are the users, operators and other systems that may connect to the system.

What services it provides

A summary of the main functionality with a list of all the main services, their purpose, and how they are performed.

Interfaces

A brief description of the interfaces with emphasis on user interfaces.

Other properties

Give a brief account of the general properties and the non-functional properties provided. Emphasise positioning properties. If maintenance properties are important they should be mentioned.

Variability and evolution

Consider flexibility for change, variability and other issues related to market adaptation in space and time.

Technology issues

If technological principles used or other design aspects are important at this stage, they should be mentioned.

System family statement notations

The family statement will be expressed in prose, supplemented by figures.

Family statement relationships

With domain The statement is likely to refer to the domain descriptions, but it may also use terminology from the domain. Therefore the domain given terminology used shall have entries in the domain dictionary.

Within family The statement is likely to use some system family specific terminology. Such terminology shall be defined in the system family dictionary.

Harmonising system family statement

With domain Be sure to use a terminology which is consistent with the domain terminology.

Every domain term used should be defined in the domain dictionary.

With family Every family specific term should be defined in the family dictionary.

For every service mentioned there shall be a corresponding entry in the Specifications.

Summary of static family statement rules

t.b.d.

Developing system family statements

What It is assumed that the creative work which actually determines what the system is going to be is performed in other activities, mainly in the system Study activities and Specification activities. The task here is simply to express what has been decided. The form shall be suitable for a broad audience within the company (not only developers) and possibly for external use. It will be used for internal decision making and control.

Actors The system family statement should be written jointly by market persons and developers. Management should be involved in approving it.

Making system family statement

The activity may be subdivided according to the topics of a system family statement:

Make system family statement

1. Describe How it relates to the domain (p.6-70).
2. Describe How it relates to the environment (p.6-70).
3. Describe What services it provides (p.6-70).
4. Describe Interfaces (p.6-70).
5. Describe Other properties (p.6-70).
6. Describe Variability and evolution (p.6-71).
7. Describe Technology issues (p.6-71).

Evolving system family statement

The system family Statement should be written in such a way that it can be kept as stable as possible. Nevertheless, some evolution is likely to occur either because the Statement needs improvement or because new features are added to the systems in the family.

How to proceed depends on the nature of the change.

System family dictionary

What is a system family dictionary?

The system family dictionary is organised in the same way as the Domain dictionary (p.-59). The two dictionaries are used together to give full coverage of the relevant terms. They may even be organised as two parts of the same dictionary.

Objective To define system family specific terminology in order to:

- improve precision and efficiency in the process;
- facilitate training of new people;
- facilitate reading system documentation.

System family dictionary content

The general format is:

<term><explanation>[example][synonyms]

The family dictionary is related to the domain dictionary. The two dictionaries may physically be different parts of the same dictionary.

System family dictionary relationships

Relationships system family dictionary - domain

The domain given terminology used in families shall be the same as in the domain dictionary.

Harmonising system family dictionary - domain

Ensure that the general domain terminology is applied in the families. family dictionaries should refer to the domain dictionary for all domain given terminology and avoid redundant definitions (which may develop into inconsistent definitions).

Relationships system family dictionary - rest of family

For entries in the dictionary that correspond to concepts that will be represented directly by types (classes), it may be a good idea (if this is known) to use the same name on the type as the designation of the concepts.

Harmonising system family dictionary - rest of family

Ensure that the relationships with the other family descriptions are maintained. See System family dictionary relationships (p.6-73).

Summary of static system family dictionary rules

- *Represent each essential system family phenomena and concept by an entry in the dictionary.*
- *Keep the dictionary updated throughout the development. If desired classify the entries as coming from analysis or design, domain, environment or system. This may help in updating the dictionary and also to answer questions like "Is this phenomenon covered by the domain of the system?" or "Is this type of entity handled by the system?"*

Developing system family dictionary

What

In general, the dictionary comes from the other family descriptions, as its purpose is to define the terminology used there. Thus, the dictionary is developed more as a spin-off from making the other descriptions, than as an independent activity.

When object models and property models are developed, the dictionary should cover all the objects/types, associations and properties represented in those models.

Actors

The dictionary is most likely made by the developers.

Making system family dictionary

The family dictionary is made for the first time as part of the first requirements analysis, see: Making requirements (p.-40).

The first family dictionary comes from the System family statement (p.6-70), the System studies (p.-20) and the System family statement (p.6-70). By studying the nouns in the System family statement (p.6-70) we can make the initial dictionary. But the dictionary shall contain more than the nouns, it shall also include services or functions and associations which may be visible in the domain Statement as verbs. It shall include every term used in the system specifications. Note that domain terminology is not to be covered.

editorial

1. should the dictionary be organised in some way? e.g. according to the domain statement categories, according to the kind of entity (concept, property, entity, relationship, service)

2. should the dictionary contain references to object models and property models?

Evolving system family dictionary

The dictionary is evolved as part of the evolution of family descriptions. Whenever existing terms are modified or new terms are introduced in any of the other family descriptions, the family dictionary should follow up.

Summary of dynamic system family dictionary rules

- *Develop the dictionary in parallel with other family descriptions.*
- *Use the dictionary actively as a source of terminology when making the other descriptions.*
- *Do not add new terms to any model without checking that there is no appropriate term in the dictionary already.*
- *Analyse each entry checking that it is precisely and unambiguously defined. Look for similarities among entries. Avoid to use the same term for different things, and to use different terms for the same thing. If necessary define synonyms explicitly. Clarify relationships between related entries, e.g. subtype relationships.*

Family implementations

Implementations

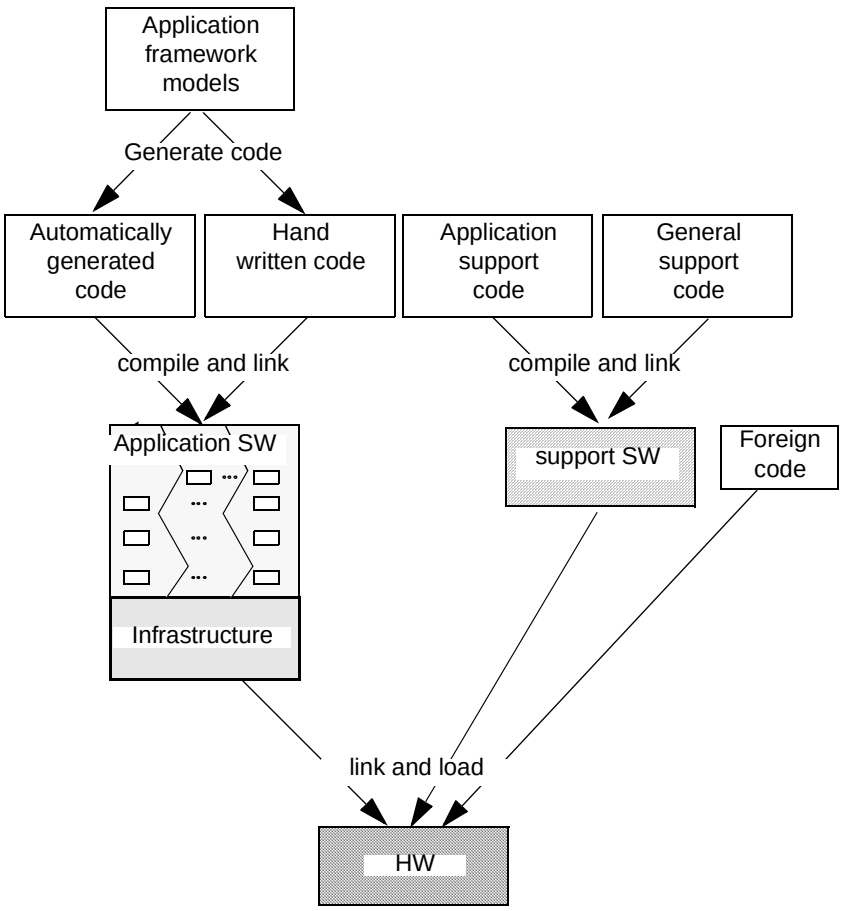
What

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.

Figure 6-13 (p.6-42) illustrates some aspects of implementations. Note that the code generated from application frameworks will interface with code coming from different sources. To produce a concrete system these various parts of code must be linked together and loaded on the hardware.

Figure 6-20: Software implementation

Open figure



Architecture models contain information that may be used to direct the transformation from application frameworks into implementation code and the building of a concrete system.

Automatic code generation

State-of-the-art tools allow the application framework software to be automatically derived. The code which is generated for the application framework must be adapted somehow to the software environment (operating system, input-output, middleware). 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 those parts of applications and frameworks where SDL is used.

Family auxiliary

Why family auxiliaries?

In addition to the formal models and the less formal statements and dictionaries there is always a need for more informal descriptions.

An important category are made up by procedures, or methods, for handling of families.

Method for evolution

In order to stay ahead of competition today, it is necessary to shorten the lead times needed to introduce new services and service features. To this end TIMe seeks to achieve a requirements oriented mode of systems engineering, where service flexibility and incremental evolution is a key issue. Aspects to consider are:

- Modularity in application property models. How to describe services (and other properties) in a modular way so that new services may be added or existing services modified with minimum impact on other services?
- Application flexibility. How to structure applications in a modular way so that the impact of an new or changed service is limited, and how to incrementally evolve the application.
- Implementation flexibility. How to follow up application increments in the generated implementation code, and how to update existing systems.

For any system family where evolution is an issue, one should try to define guidelines for how to perform evolution.

Method for framework code generation

Objectives Once the architecture is defined, much can be gained if the remaining development effort can concentrate on application evolution. That will be possible only if there is a well defined method, supported by automatic tools, for derivation of implementations.

What The method needs to consider several issues:

- Automatic code generation. How to produce implementation code that fits into the application software environment? How to handle incremental generation?
- Hand generated code. What are the rules that hand generated code shall follow?
- Integration of automatically generated code with hand generated code, foreign code and support code.

How the code shall be divided into modules for ease of generation and handling?

Method for system instantiation

Objectives The purpose is to enable cost effective production of system instances that:

- satisfy customer requirements;
- supports future evolution and maintenance.

What This method should define the procedures and the tools that shall be used to generate system instances. Problems to solve are:

- how to specify configurations at the various abstraction levels: application, framework, platform;
- how to generate or produce the various parts;
- how to compose the parts into a system;

how to load and initialise the software.

Application

Content and scope

The application provides the main services of a system and is therefore the most valuable part of a system from a user point of view. Application models are very central in TIMe:

- they are the soul of service orientation, as they model both the services and the object that provide the services;
- they enable service flexibility and service evolution at a high abstraction level rather than on the (concrete) implementation level;
- they enable quality assurance of services before they are implemented;
- they are used as the starting point for implementation architecture design;
- they are used as source when generating application (implementation) code.

In this Chapter we shall describe:

- The Application reference model (p.6-81): the generic structure of application systems.
- Application models (p.6-86): how application systems are represented in models.
- Developing applications (p.6-106): how we go about developing the application models:
  - Application specification (p.6-93),
  - Application design (p.6-96).

Application reference model

It is abstract

The application is an abstract system which provides the services that users and other systems in the environment need. It determines the quality of the system services.

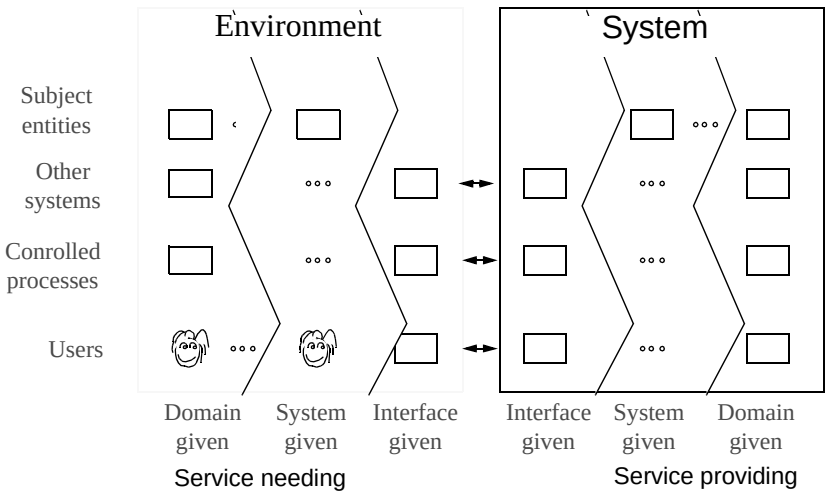
As an open systems it exists in a context where it attain purpose and meaning from an environment. This context may again be part of an even wider context, such as an *enterprise*, where the system and its environment together serve some higher purposes.

The domain is a generalisation of this wider (enterprise) context. In contrast to the domain, the system context is a part of the world being served by a system. The domain is a more general and often wider area of concern.

In the reference model for application systems we consider the system and its environment, see Figure 6-21 (p.6-81). Both the system and the environment are decomposed into Domain given (p.6-82), System given (p.6-83) and Interface given (p.6-83) parts:

Figure 6-21: Application system reference model

Open figure



- Domain given (p.6-82) parts provide functionality which is common to most systems in a domain.
- System given (p.6-83) parts provide functionality which is specific to a particular system or system family.
- Interface given (p.6-83) parts take care of the communication between the *service needing* parts of the environment and the *service providing* parts of the system, see Figure 6-21 (p.6-81).

These parts may have quite different characteristics and contribute to the overall functionality in different ways. Note that the services are related to the domain given and the system given parts, while the interface given parts only serve as communication media. Interfaces have been separated from services in order to achieve protocol transparency

and modularity. It is generally desirable that services can work over a range of communication facilities and that interfaces and services can evolve as independently as possible.

The Environment (p.6-84) is further decomposed into:

- Subject entities (p.6-84): entities known by the system, but not directly interacting with it. The Access Zones handled by the Access Control system is one example. The materials and products handled by a MPS system is another.
- Other systems (p.6-85): other systems communicating with the system. In the case of an ATM machine, this would be the central bank computers.
- Controlled processes (p.6-85): equipment being directly controlled by the system. The Doors controlled by the Access Control system is a simple example. More complex examples are the mechanical parts of a paper mill or a robot.
- Users (p.6-85): human beings communicating with the system. In this category we have end-users as well as operation and maintenance staff.

Since the system shall serve the environment, we will find corresponding entities within the system. In fact, TIMe recommends to "mirror" the environment by objects inside the system. Some of these will be active object and some will be passive objects, see Object Models. The active objects provide *services*, while the passive represent entities and relationships the system needs to *know*.

The application system reference model has been inspired by the OOA&D method *(Mathiassen et al. 1993)*, but adopted and detailed to suit reactive real-time systems. Methods like UML do not have this kind of system reference model.

We believe the reference model will help to structure system models and to give more precise method guideance.

System context

What A system context is a part of the world containing the system itself and its Environment (p.6-84). Thus, Figure 6-21 (p.6-81) illustrates a system context.

Symmetry Note that:

- there is symmetry between the environment and the system: objects in the environment are *served-by* and *represented-by* objects in the system;
- there is symmetry of interfaces around communication links served by protocol stacks. Sometimes the system boundary coincides with such links and sometimes not.

Domain given

What Parts that are modelled in the domain and therefore common to most systems in a domain are classified as domain given parts. The domain given parts of the environment contain domain entities that are served by the system, typically actors and processes, and subject entities known by the system. The domain given parts of the system itself con-

tain domain entities that are realised by the system, typically some domain Helpers. They also contain passive objects representing domain given entities of the environment.

Example The users and the Access Zones of the Access Control system are examples.

Explanation Domain services will be quite general and common to most systems in a given domain. Every PBX, for instance, is able to provide normal two-party telephone calls. Any access control system will know about users and access rights. Such parts are likely to be stable and may potentially be reused in many systems and system families. This is one reason to separate these parts from the more system specific parts.

In methods like UML and OOA&D the emphasis is on passive objects representing knowledge about phenomena in the problem domain. In our method, we also include active objects that perform services.

System given

What The system given parts are specific to a particular system or system family. They will act together with domain given parts to provide the services.

Example In the Access Control system the operators and the objects serving the operators is an example. Another example is the error handling and initialisation functionality needed in most systems.

Explanation These parts provide the services that makes this particular system, or system family, different from other systems in the problem domain. The System given (p.6-83) parts can collaborate with the Domain given (p.6-82) parts of the system to accomplish the services. They are separated from the Domain given (p.6-82) parts because they are less stable, general and reusable. The purpose of these parts may be to:

- Perform solution dependent services, i.e. services that depend on the actual system design. Error handling and maintenance services are typical examples.
- Attend to special needs, not solved by other systems in the domain, i.e. to provide some positioning properties.

Interface given

What The interface given parts take care of the communication between the service needing parts in the environment and the service providing parts in the system.

Explanation Communication require interfaces. In some cases the interface is simple: just hit a button. In other cases it involves complex behaviour, such as protocols and graphic user interfaces. In most cases there is independence between the interfaces and the service provided through the interfaces. It is therefore recommended to isolate what is specific to an interface from the services provided over it. This is particularly true where there is a complex behaviour associated with the interface.

Such separation allows interfaces and services to evolve independently and help to protect investment in either area from changes in the other.

Parts The interface specific parts are subdivided into:

- User Interfaces. Parts taking care of Human Machine Interaction (HMI). These parts are often critical for the user satisfaction, and hence the success of a product. User Interface design is an area that require careful attention and specialist knowledge. It will sometimes involve prototype building and experiments. It will therefore often be a separate development task involving other people than the other parts of the system.
- Process Interfaces. These take care of the interfaces to controlled processes. They will involve sensors and actuators as well as interface electronics and software drivers. This part will often require understanding of the technological principle of the controlled processes, actuators and sensors. It is therefore a multidisciplinary area, involving several technologies. It is an area where real-time constraints are important and need careful consideration.
- Other System Interfaces. These take care of communication with other systems. They involve the necessary hardware, the software drivers, and the protocol stacks. Sometimes they will involve standard networking technology. In other cases special solutions are needed.

Layering Common to all these parts is the *Layering Principle*. All interfaces will be layer structured, starting with the physical medium moving upwards to the services in the System given (p.6-83) and Interface given (p.6-83) parts of the system.

Example In the AC system we have a user interface with a physical layer: the *Panel*, and a protocol layer: *PanelControl*.

Environment

The environment consists of the parts of the surrounding world which are either known to the system or communicate with it.

Subject entities

What Most systems will contain data representing entities, relationships and measures external to themselves. The actual entities, relationships and measures represented by such data are denoted collectively as *known entities*. The known entities can be seen as the meaning (semantics) of the data. Subject entities are known entities that a system does not interact with directly.

Example A salary system, for instance, needs to know about employees, but will normally not interact with them, hence the employees are subject entities.

Subject entities will often be domain given.

Other systems

What Other systems are external technical systems with which the system in question is communicating to accomplish its purpose.

Example A PBX for instance, will communicate with nodes in the public telephone network. An ATM machine will communicate with the central bank computers.

Explanation Many systems will communicate with external systems. In some cases the relationship will be asymmetric, as in a client server architecture. In other cases it will be symmetric as between the switching nodes in a communication network. Anyhow, the relationship with other systems may be central to the purpose of the system in question. Sometimes the system even receives its prime purpose from external systems.

Other systems are sometimes domain given and sometimes System given.

Controlled processes

What The controlled processes are equipment in a surrounding technical system being directly controlled by the system in question.

Example In the Access Control system, the Doors are controlled processes. In an ATM machine, the money-bin is a controlled process. In a lift control system, the physical hoist unit and the lift chair are controlled processes.

Explanation Controlled processes will only be present in embedded systems, where the application system is controlling a larger technical system, e.g. a paper mill, a telephone exchange or a robot. Normally this larger technical system will rely on the application system to behave as required. This is typically the case in so-called mechatronic systems, where software, hardware and mechanical solutions are integrated. A mechatronic system will comprise active parts which exercise control and passive parts being controlled.

System boundary It is a matter of definition where to put the system boundary, and hence what to consider as part of the system in question and what to consider as controlled by the system. If we consider a complete lift system, the physical hoist unit containing motors, brakes, wires, etc. will be considered as part of the system. If we consider only the lift control system, then these parts are considered as controlled by the system.

The controlled processes will often be domain given.

Users

What Users is the term collectively used for human beings interacting with the system in question. They may play a variety of roles such as operators, service personnel or clients. domain actors supported by the system will be domain given, while others will be system given.

Application models

Application model overview

Objectives

First of all, application models serve three basic purposes:

- To describe behaviour at an abstraction level where it can be understood and analyzed independently of a particular implementation.
- To be a firm foundation for designing an optimal implementation.
- To be source for automatic generation of implementation code.

The purpose of the application models in general is to answer *what* systems in the family shall do.

They should support unambiguous communication among project members and in-depth understanding by the individual. An application model is made primarily for human communication and analysis, not for machines. But it should also be processed by machines.

It should be readable for the human being without being vague or ambiguous and it should express the behaviour without unwanted bias towards the physical implementation in order to allow freedom in selecting the implementation.

At the same time it should be implementable so it can be trusted as documentation. In fact, we will generate code automatically from application models.

Once application models serve these purposes and we are able to actually engineer systems on the application level, some secondary purposes emerge. We want to further reduce lead times and to introduce new services more quickly than before. Therefore we want application models to be:

- modular with respect to service description;
- flexible with respect to service introduction.

To achieve all these objectives, application models should be structured with the following criteria in mind:

- Clarity and simplicity of behavior interpretation by the human.
- Modularity supporting incremental service evolution and isolation/encapsulation of change.
- Coverage of the necessary variability.
- Possibility for analysis by the machine.
- Possibility for automatic code generation.

What

As illustrated in Figure 6-22 (p.6-87), the complete application system may be defined as a type. This will be a natural thing to do during the first time development. However, once a framework has been defined, the complete application system may become less

important, because it is the component types and not the complete application system, that will be used in frameworks. Therefore the component types used to construct the application system may be more important than the system itself.

This observation is strengthened by the fact that it may be difficult to express all the variability needed formally in one type model. It may prove more practical to develop several component types that are easy to compose into alternative system types. Thus, the most important part of application models is a library of (component) types that can be used to define application systems using either composition or inheritance.

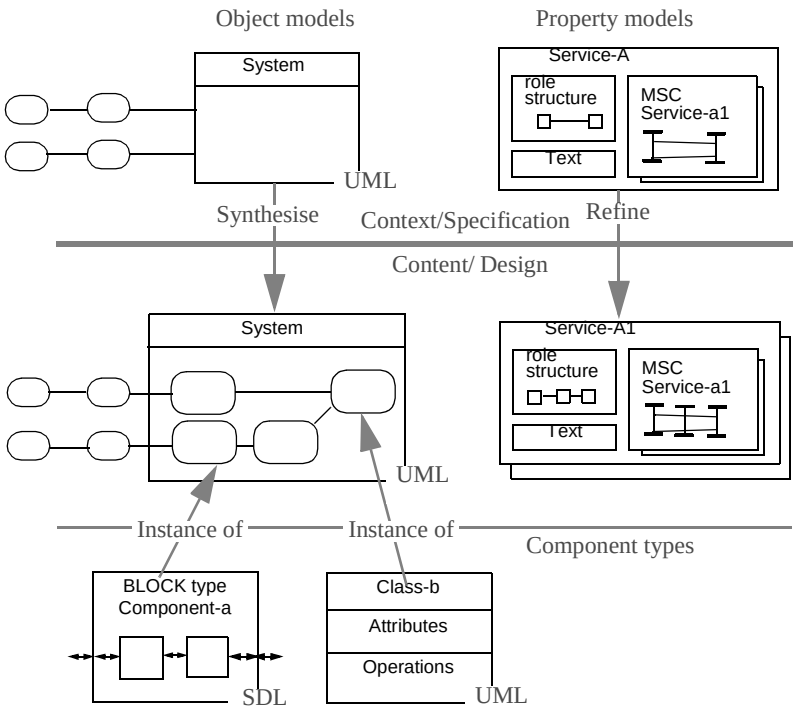
Application model content

Application specification and design parts

As all other family models, application models have two main parts: *the specification part* (see Application specification (p.6-93)) and *the design part* (see Application design (p.6-96)). Each part consists of object models and property models as illustrated in Figure 6-22 (p.6-87):

Figure 6-22: Application specification and design

Open figure



1. The specification focuses on the environment and how it relates to the entity being defined. In the object model, the entity itself is represented as a black box while the environment is detailed. Associated property models will focus on interactions between the entity and the environment. Every service it provides shall be described by means of roles, text and MSC.
2. The design focuses on the content. In the design object models we will see how the application system is composed from parts and also how these parts behave. In the property models we will see how design objects interact. It is of course, essential that the properties actually provided by the design objects correspond to those required in the specification. (A combination of property conserving synthesis and verification techniques is used to ensure that.) The object model design consists of two parts:
   - structure design which define the structure of objects and component aggregates;
   - behaviour design that define the behaviour of objects.

Application system boundary

What should be inside and what should be outside the system is a question that often causes high tempered and endless discussions. We contend that the decision is not very crucial, as the method we propose will allow the boundary to be adjusted later on with little effort.

The important thing is to include either in the environment or in the system itself everything the system ultimately will serve or know. In particular it is important to represent all the various stake-holders needing to be served or represented by the system. These should be placed in the environment. Other objects we are sure never will be produced or delivered as part of a system instance, e.g. people, should be put in the environment. This does not exclude the possibility of including some of these in the system at a later stage, for instance as part of a simulator.

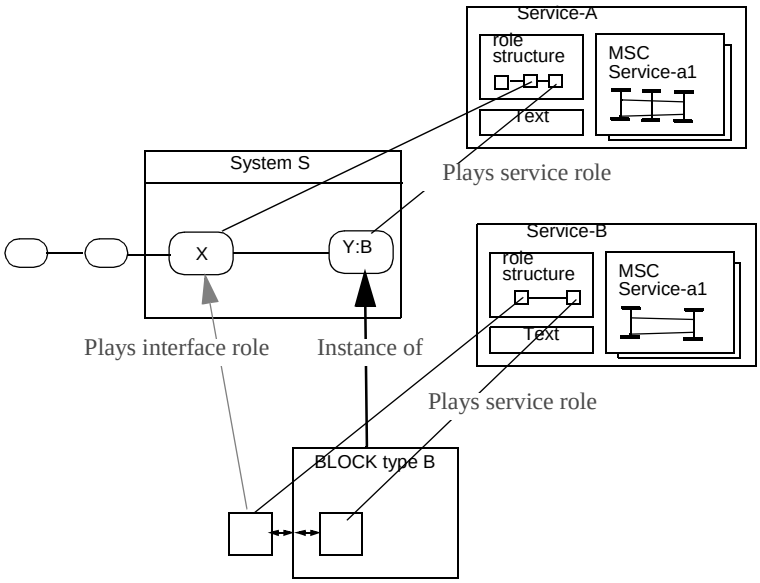
Application services and objects

The services of a system are defined in application property models. As in the domain, the application service models contain a textual explanation, a role structure and an MSC document.

In general, an object is assigned properties in the following three ways, see Figure 6-23 (p.6-89):

Figure 6-23: Ways that services are related to objects

Open figure



1. By explicit assignment of service roles using the *Plays service role* relation. This is because the object is part of an object type to which we explicitly assign property models.
2. By implicit assignment of service roles from the object type it is an *instance of* . This is because objects are instances of types that may have explicitly assigned roles.
3. By implicit assignment of *interface roles* required by objects in its environment. This is because the object appears in an environment where surrounding objects will expect the object to behave according to some interface.

Of course, all the roles that an object receives must somehow be consistent with each other and with the properties that the object actually provides.

Therefore, care must be taken to maintain consistency between the properties specified explicitly for the system as a whole and those received implicitly from its components and the environment.

Role sets

- *The set of roles explicitly assigned to an object should be a subset of those that follow implicitly from its type and its environment.*

The application reference model parts

Domain given

Application models are closely related to the domain and the needs of domain stake holders. We are likely to find some of the domain actors in the system environment (those that the system will support), and some domain helpers inside the system (in the domain given part).

System given

A system family may have many features that were not covered in the domain models, e.g.:

- positioning properties that help this family to be a winner in a competitive marketplace;
- properties related to issues outside the domain such as operation and maintenance functions for the system as such.

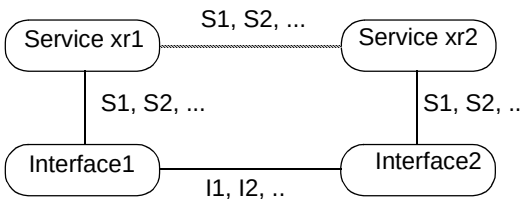
We recommend to model the System given parts in the same general way as the domain given, except that the System given parts originate outside the domain and have no relations to domain models (except in cases where they lead to new understanding of the domain).

Interface given

It is recommended to distinguish between the service behaviour and the interface behaviour by using layering as illustrated in Figure 6-24 (p.6-90).

Figure 6-24: A service layer and an interface layer

Open figure



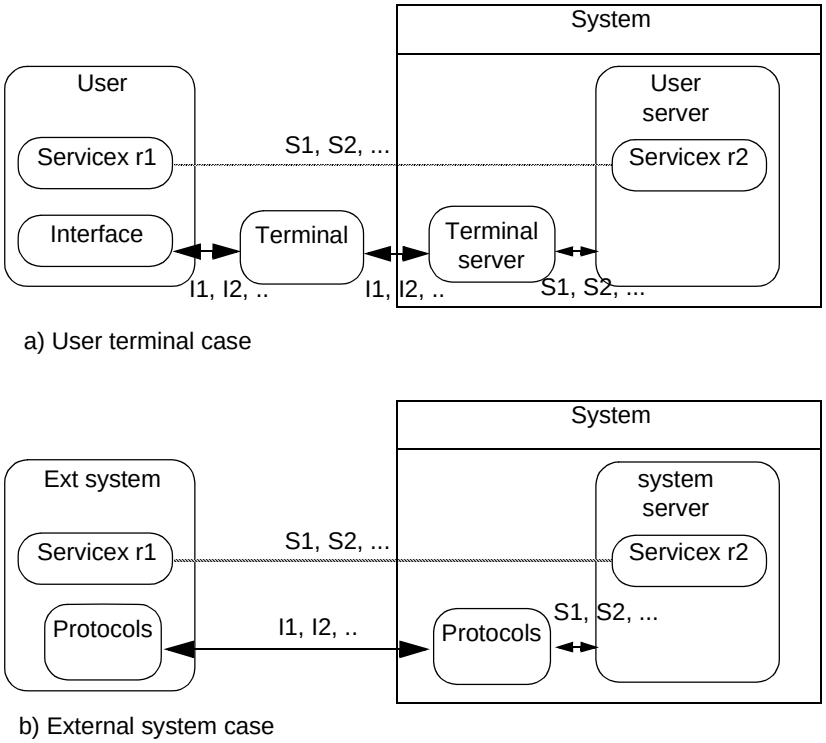
Further we recommend to first define the service behaviour separately without any reference to interface given parts. This means to consider only the two service roles *xr1* and *xr2* and their dashed interconnection in Figure 6-24 (p.6-90). When interfaces are introduced, the services shall ideally behave in the same way even though the interconnections go via the interfaces and the dashed interconnection in Figure 6-24 (p.6-90) disappears.

(It may happen that interfaces interact with the services in ways that were not anticipated in the service models, for instance that messages may be lost. In such cases the service behaviour must be modified to take the interaction into account.)

Figure 6-25 (p.6-91) illustrates how services and service objects may relate to the interface given parts. Where to place interfaces in relation to the system boundary, and how to model them is often a difficult choice.

Figure 6-25: Two cases of interface given parts

Open figure



In the User terminal case, we have placed the *Terminal* in the environment and a *Terminal server* inside the system. Note that the *User* will play both a service role and an interface role in this case. Since these roles are internal to the *User* object, the service interactions *S1, S2,.* are not visible outside the *User*. Inside the system however, the roles are played by two different objects, and therefore the service interactions are visible there. In the *Ext system* case, the interface given parts are *Protocols* belonging to the two systems.

Application languages and notations

As illustrated in Figure 6-22 (p.6-87), the main languages to use are MSC for property models and UML combined with SDL for object models.

SDL is the main language for control behaviour, while UML will be used for other aspects (such as user interfaces and data-bases). Typically UML will be used in the specification part, and possibly for top level design structuring. SDL will take over where the main thing is to model reactive behaviour.

In some cases it will be possible to express the application object models completely in SDL.

For description of interaction properties, we stick to MSC even if UML is used for the Object Model.

Application specification languages and notations

Normally UML will be used for the specification object models:

Application specification notation

- *In general use UML for the specification object model:*
  - *Context: model the application system as a Class. Represent the environment as object sets with specified variability. Describe associations and communication links in the environment and with the system.*
  - Content: model the content (if any) as object sets inside the class.
  - *Component types: define a class for each component in the system environment and content. For each component class, model the context using separate diagrams.*

For property models, text, role structures expressed in UML and MSC will be used.

- *There shall be a separate property model for each service and interface containing:*
  - *a role structure expressed in UML;*
  - *explaining text;*
  - *a MSCs for every initiative and its most important responses.*

Using SDL in application specifications

If SDL is used, we recommend to model the application system as an SDL Block type rather than as a System type. There are two main reasons for this:

1. It is possible to formally describe the environment of Block types but not of System types.
2. Block types can be used as components while System types cannot.

- *Where reactive behaviour is important and general relations (associations in UML) are unimportant, SDL should be used in the specification:*
  - *Context: model the application system as a Block Type in SDL. Represent the environment as gate constraints.*
  - Content: model the content (if any) as Block sets.
  - *Components: define a Block or process type for each component in the system or the environment. For each type, define the type context.*
- *Associate the roles of property models with the objects and types/classes of object models.*

Application design languages and notations

For reactive, state transition behaviour it is recommended to use SDL. This means that active objects in general should be described with SDL. One exception is graphical user interfaces. For passive objects, and especially objects going into a database, UML may be preferred.

This means that the application design may use a combination of SDL and UML.

Application design notation

- *If the specification uses SDL, then continue with pure SDL in design.*
- *If the specification uses UML, then continue with UML for the structure until SDL components are reached.*
- *Use SDL for components and subsystems having reactive, state transition behaviour.*

Of course this division may cause interworking problems between the SDL and UML parts.

- *For property descriptions, we use role structures in UML, text and MSC.*

Application specification

Objectives

An application specification serves to answer fundamental questions like:

1. What services are provided?
2. What is the environment i terms of active and passive objects?
3. What is variable in the environment and the services?

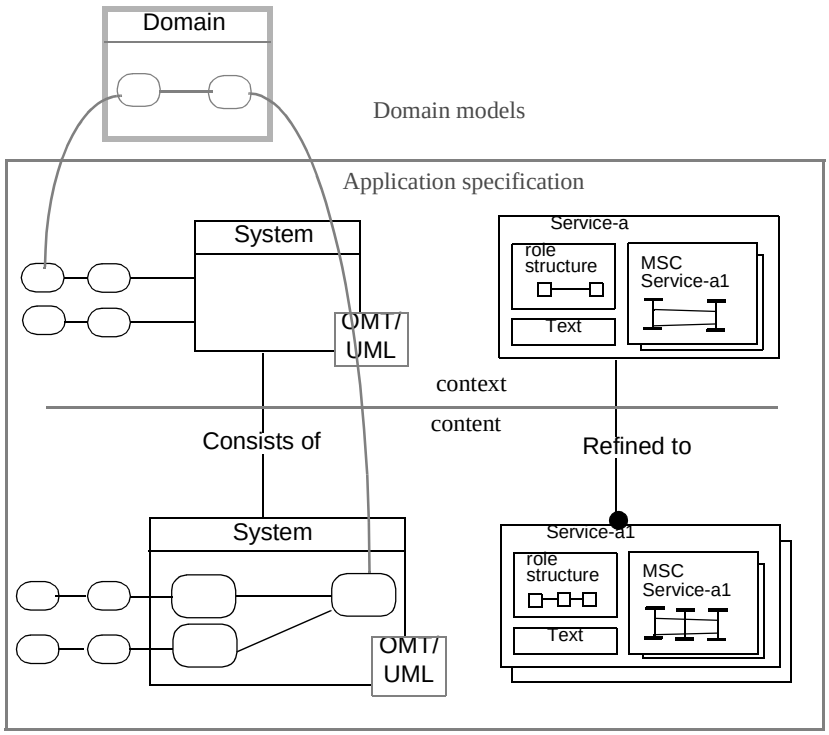
It specify the external interfaces and behaviour properties (services) of an application.

What

The application specification models are illustrated in Figure 6-26 (p.6-93)

Figure 6-26: Application system specifications

Open figure



Specifications are developed first for the application system as a whole and later for each (component) type. We distinguish between domain given, System given and Interface given parts. It is natural to use this division as a guideline in the specification:

- *Domain given parts and services*. These originate in the domain models and may be used more or less directly in the specification. The specification shall describe which parts of the domain that are supported by the system. Some of these will be in the environment, and some inside the system, see Figure 6-26 (p.6-93). Domain given objects inside the system might as well be shown in the specification. For domain services, it may be possible to use the corresponding domain property model directly, but it is likely that some modifications will be needed. In any case the relationship to domain models shall be maintained, see Application model relationships (p.6-97).
- *System given parts and services*. These may be special for the system family or they may be shared with other families. We recommend to focus on the environment, and avoid specifying content. The specification shall describe system given objects in the environment, and the system given services. We do not include System given objects inside the system in the specifications unless there are good reasons for it. But the services may well specify service roles that will be played by (as yet unknown) actors in the system.
- *Interface given parts and interface properties*. These are introduced to provide reliable communication across the system boundary for domain given and system given parts. As the interface given parts often coincide with the external system interfaces they are an important part of specifications. Therefore specifications may include interface objects both in the environment and the system. Layering should be applied to isolate the interfaces from the other parts.

Domain given types will be quite stable, and may be reused between many systems and system families. System given types are less stable as they provide the functionality that is particular for a system or system family. Among the interface given types, user interfaces can be the least stable. Protocols, on the other hand, may be quite stable.

Application specification content

Domain given specification

One key issue in any system development is to decide on what parts of a domain to support. (This is true even if there are no explicit domain models.) Basically this means to decide on which domain actors and which domain services to support. These shall be represented in the specification as follows:

Domain given specification

- *Every domain* actor *and* helper *needing to be served by the system shall be represented (as* active objects*) in the environment.*
- *Every* stake holder, subject entity *and* helper *the system needs to know shall be represented (as* passive objects*) in the environment.*
- *Every domain helper the system shall implement shall be represented in the (domain given) content.*

- *Every domain relation(association) the system needs to know shall be represented in the environment.*
- *Every domain communication link the system needs to handle shall be represented.*

As a result we have a subset of the specification object model representing the domain given parts. Inside the system (in the content) we only find domain objects (Helpers) that are going to be *realized* by the system. In the environment we find the domain objects that are going to be *supported* (if they are active) or *represented* (if they are passive) by the system.

- *Every domain given object shall have a type (class) which is related to the corresponding domain type (class).*
- *Each domain given service shall be described independently of interfaces using service roles.*
- Each domain given type shall have relations to the domain given service roles it participate in.

In this way there will be a separate service description (using role structures, text and MSC diagrams) for each domain given service. Interaction with System given service roles should be shown but interface given roles should be avoided.

System given specification

We do not include System given objects inside the system in the specifications unless there are good reasons for it. We recommend to focus on the environment, and avoid specifying content.

Again we recommend to separate services from interfaces.

In an enterprise view, consider stake holders and other entities that need to *interact with* or be *represented by* the system.

System given specifications

- *Every entity that needs to interact with the system shall be represented in the system environment with a communication link to the system.*
- *Every entity and relation the system needs to know shall be represented in the environment.*
- *Each service shall be described independently of interfaces using service roles.*
- *Each system given object in the environment shall have a type definition with relations to the system given service it takes part in.*

As a result, we have a detailed description of the environment and the communication links with the system. Each service has a separate description, and we know which service roles that are going to be played by the system itself and by the objects in its environment.

Interface given specification

Specifications will often be quite detailed about interface given parts. Interface given parts will be localised somewhere on the communication links between the system and the environment where their role is to convey the service interactions.

Interfaces will often have a layered structure with one or more protocol layers on top of a physical layer.

Interface given specifications

- *For every communication link with the environment the interface given parts shall be specified.*
- *For each type of interface, there shall be a property model containing:*
  - *interface roles using UML;*
  - *a textual explanation;*
  - *the role behaviour expressed using MSC. There should be one MSC for each independent initiative and each main course of behaviour that may follow.*

Application design

Objective

To describe the design part of an application that:

- satisfies the specification;
- has clearly defined behaviour;
- has a modular structure suitable for future evolution;
- fits into the framework when defined.

What

The context part of an application is fully covered in the specification. The design part adds the content which is not covered in the specification. It defines the content *structure* in terms of objects, and the *behaviour* of each object type.

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.

Object behaviours are central to the application, and the design should primarily be structured to give clear and concise behaviour models. We contend that behaviour should be described state oriented, see General application guidelines (p.6-102). To achieve that, the structure should contain one object for each independent (concurrent) thread of behaviour in the system. These objects may be then aggregated in various ways to suit additional purposes.

It is recommended to make a library of component types that can be used in a range of application systems to cover the full variability needed. Some of these will be real aggregates containing a structure of components, and others will be object types with behaviour. The component types designed with SDL will be organised in Packages. (Packages are used only for types that are general enough to qualify for being part of a package. It is, however, possible to start with collecting the types being used in a specific system in a set of packages.)

Application design content

Application design models consists of:

1. The application modeled as a structure type. It is a type definition with context and content that define the overall structure of the application system and may use several levels of real aggregation. It will be constructed using a collection of component types and refer to the definition of these component types for further detail. The content structuring is primarily concerned with active objects and interconnections. However, the active objects will contain passive objects and conceptual relations.
2. The component types which are either:
   - *structure* types that consist of object aggregates, defining the content as a structure of components, or
   - *behaviour types,* i.e. object types having a behaviour of their own, e.g. SDL processes. There will be both active objects and passive objects as well as association between active objects and passive objects, i.e. which active objects that access and may change the passive objects representing known entities in the context. (For SDL, the passive objects will be variables contained in processes)
3. Property models for the overall content. The application content shall satisfy the specified context properties.

Each type is defined with a context and a content with associated properties. An object type definition consists of:

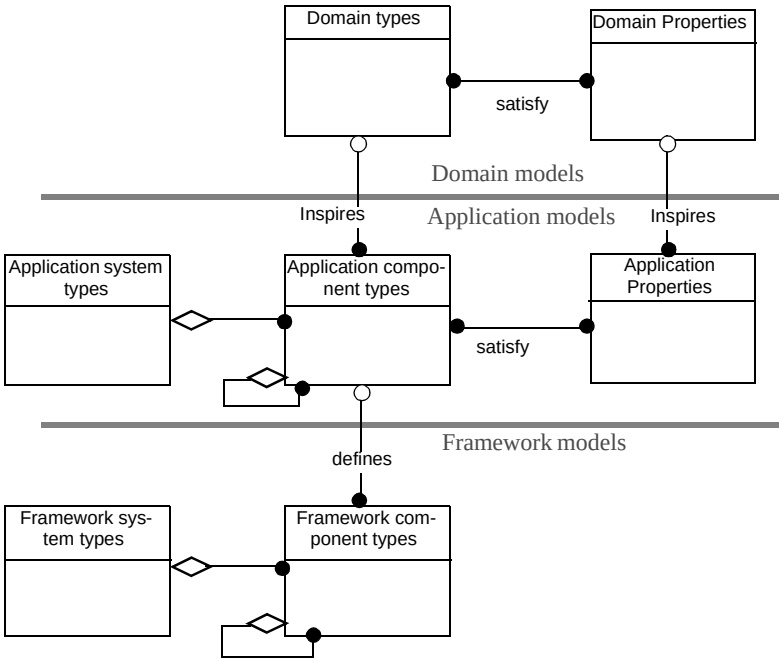
- the context in terms of active objects and passive objects in the environment;
- the interfaces in terms of gates (or equivalent in UML);
- behaviour properties in terms of service roles, interface roles and MSC;
- attributes;
- the behaviour definition in terms of SDL process graphs or UML operations.

Application model relationships

How the application is related to the domain and the Framework is illustrated in Figure 6-27 (p.6-98).

Figure 6-27: External Application relationships

Open figure



Relationships application - domain

We recommend that each domain given object has a type, with a context description. (Each of these types will correspond to one of the component types that we recommended to make in the Domain object models (p.-65)) In this way the relations to the domain go via the types.

The nature of these relationship may vary from plain equality (when domain objects can be used as-is) through inheritance (when the application objects are extensions of the domain objects) to informal similarity (when the domain objects are just used as inspirations for the application objects).

Domain relationships can thus be either *Identity*, *Inheritance* or *Inspires* where *Identity* and *Inheritance* can be expressed in the object modelling languages, while *Inspires* must be added as annotations. In Figure 6-27 (p.6-98) we have only indicated the weakest relationship “inspires”.

Ideally the domain given types should be identical to the domain types. Any difference shall be due to system given or interface given aspects. As far as possible we seek to isolate these aspects, but it cannot be avoided that the domain given objects are influenced:

- Domain given objects will most likely interact with System given and Interface given objects. As this was not accounted for in the domain models, it is likely that the domain given objects will have additional features.

- System given services will most likely interact with domain given services. These interactions must be resolved. This means that the domain given service behaviour must be *refined* to take these interactions into account, see The dialectics of refinement.

Consequently, the domain given behaviour properties will be a refinement of the corresponding domain properties.

Application domain relationship

- *Every domain given type in the application models shall be related to the corresponding domain type by either:*
  - *identity;*
  - *inheritance;*
  - *or an annotation telling that it is inspired-by.*
- *If it turns out that system given properties must be handled by domain given objects, then try to make -within the domain given objects - a separation between domain and system given aspects.*

Domain communication

- *If there are communication connections between objects in the domain object model, then there will be a corresponding communication between the design objects, possibly via some interface given object(s).*

Harmonising application - domain

Only the domain given parts need to be considered. When these parts differ from the corresponding domain models, the difference should be caused by System or Interface given aspects. If they are not, it is likely that the difference is accidental and should be removed, either by changing the domain model or the application model. (It is not uncommon that working with the application helps to improve the understanding of the domain.)

Application domain harmonisation

- *Critically review the application looking for general aspects that are not system or interface specific. Update the domain models to capture these aspects.*

Relationships application - framework

Frameworks are composed from an infrastructure part, that come from the implementation architecture, and a redefinable application part, that come from the application. Objects in the application part (of the framework) are formally related to Objects in the application by using the same types. However, the application part of the framework may be structured somewhat differently from the application. This means that the application types need to be designed so they can be distributed in the framework and communicate transparently with each other and the environment. It also means that they must be adapted to the operation and maintenance services provided in the framework, e.g. the methods for dynamic system configuration.

Application framework relationship

- *Every type used in the application part of frameworks shall be defined in the application models.*
- *Every application type shall be applicable in the framework.*

Harmonising application - framework

Application types are developed and maintained as part of the application, and used in frameworks to produce executable systems. It is not necessary to develop a complete application system each time a new executable system is to be produced. It is sufficient that the necessary (component) types are available for use in the framework. However, to fit into the framework, application types must comply with the framework infrastructure and interfaces.

Application framework harmonisation

- *When the framework is defined, adjust all application types to comply with the framework infrastructure and interfaces.*
- *Maintain the relationship between framework and application by using identical types.*

Relationships application specification - design

We recommend that the properties of a design always shall satisfy the properties of a specification.

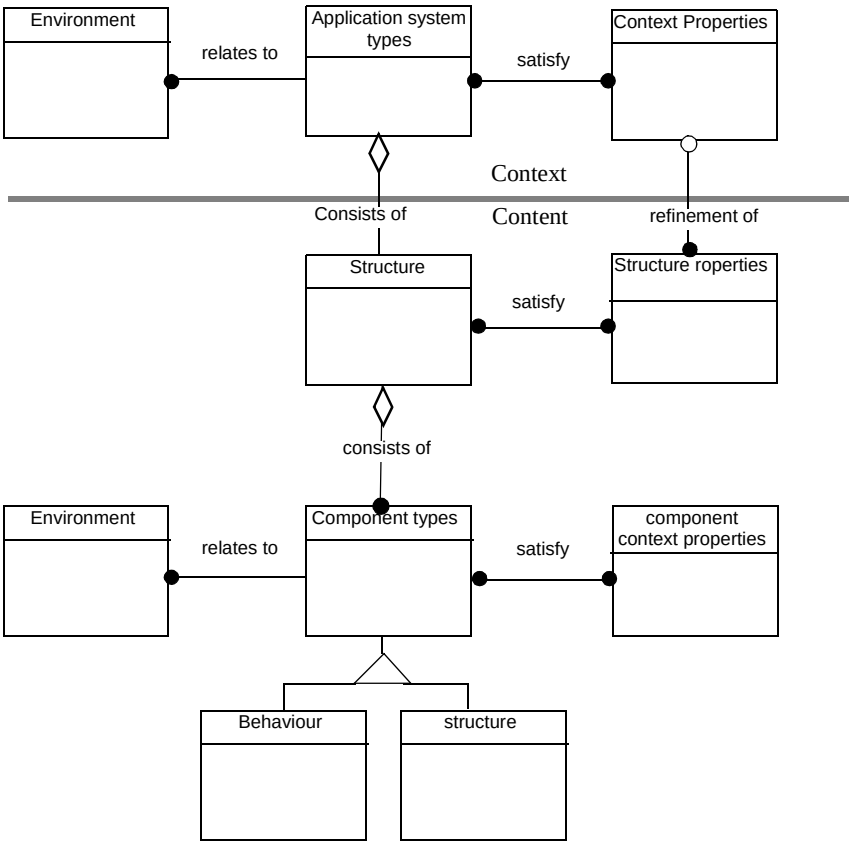
This can be verified to some extent using state-of-the-art tools, e.g. verifying that a MSC specification may be executed by a SDL design.

If possible, the rules for mapping service specifications to design objects shall be described, both for the purpose of traceability and for incremental service creation. Ideally, if the rules are well defined, we may perform service evolution mainly by evolving the service specifications. Although this cannot fully be achieved presently, being clear about the rules is a step in that direction.

The internal relationship of application models are illustrated in Figure 6-28 (p.6-101).

Figure 6-28: Internal application model relationships

Open figure



In general, the content properties shall be a refinement of the context properties such that the application content model satisfies the context properties.

The application content receives properties from the component types that are used. These component properties shall be consistent with other properties that are specified for the content as a whole and with each other.

Harmonisation application specification - design

Top-down we seek to synthesize the content such that the context properties remain satisfied. Bottom-up we compose the content using existing types such that each interface is consistent, and that the context properties are satisfied. As far as possible we use constructive rules.

Application internal harmonisation

- *Ensure that the design properties satisfy the specification properties, either by using property conserving synthesis or by verification.*

General application guidelines

Goals

While we structure domain models mainly to show the concepts and the relationships of the domain, we strive to achieve clarity in the behaviour description when we make application models.

Golden rule

- *The golden rule is to partition along the lines of independence and not across dependencies.*

The main goal is to find an object structure that will enable us to describe behaviour as clearly and concisely as possible in state oriented fashion.

State orientation

State orientation is a general principle for behaviour description that helps to make behaviour descriptions easier to understand and analyse. The basic idea is to unfold the state transition behaviour as explicitly as possible in the description. In that way the description is similar to the real behaviour and therefore easier to follow than an action oriented behaviour.

- *Represent what the environment may distinguish as control states of the objects (Processes in SDL) as states in the behaviour descriptions (process graphs in SDL).*
- *Critically review all decisions in SDL to ensure that they are not symptoms of undesirable state hiding.*
- *Represent what the environment may distinguish as different control signals by different signal types.*
- *Branch on input signals rather than on decisions.*

Data

- *Use data*
  - *when the process graph structure is not dependent on the data values (non-decisive data);*
  - *to keep information about the situation and structure of the environment (context knowledge);*
  - *to control loops that are not terminated by specific signals (loop control data).*
- *Introduce special processes to encapsulate shared data. Encapsulate data needing independent access in separate processes.*

Resource allocation

- *Introduce a special resource allocator process to control the access to each pool of functionally equivalent resources.*

Concurrency

In order to describe behaviour state oriented and clearly, it is necessary to find objects that behave as independently as possible.

- *Model independent and parallel behaviours as separate concurrent objects (processes in SDL).*
- *Do not hide similar behaviours in data. Decompose such that similar independent behaviours are described explicitly as separate instances of a process type.*

Service concurrency

For the domain and System given parts that perform the service behaviour, this rule leads to:

- *The service needing parts of the environment (i.e. the domain and System given parts except subject entities) should be decomposed into objects such that every object has an independent thread of behaviour and takes initiatives independently of other objects.*

This helps to find the objects that behave independently in the environment and therefore need to be served independently by the system.

- *The service providing parts of the system should be decomposed into objects such that there is at least one active object to serve each service needing object in the environment.*
- *In other words: the service needing environment should be mirrored by corresponding service providing objects in the system.*
- *Use one process to play each independent behaviour role required by the environment.*

Communication

- *Use one channel and/or signal route to carry each independent and concurrent interaction dialogue.*
- *Protocols should be decomposed by layering, such that each layer hides the details of the protocols used on that layer from higher layers. Lower layers should provide application independent transfer services for the upper layers.*

Aggregation purposes

- *Where appropriate, objects should be collected in aggregated to provide one or more of the following aggregation purposes:*
  - *gradual approach to detail;*
  - *separation of concerns;*
  - *definition of a type (class) suitable for use in many places (e.g. AccessPoint);*
  - *layering;*
  - *encapsulation;*
  - *similarity with the concrete system. NB this consideration should not be taken before the framework is defined.*

Aggregates

Aggregates are like systems, they have a context and a content. Their environment consists of the entities they know and communicate with, and their content consists of the aggregated parts.

- *Every aggregate shall fulfil one or more of the aggregate purposes.*
- *Let each aggregate have at least:*
  - *one active object for each active object in the aggregate environment it communicates with;*
  - *one active object for each pool of shared resources to be dynamically allocated;*
  - *one active object for each block of shared data accessed and controlled independently;*

  - *one active object for each independent routing function needed.*

- *For each aggregate where there is a choice between local and remote; communication, use a two-level addressing scheme and hide the routing knowledge in a routing process.*
- *The structure imposed by the application system should be expressed as explicitly as possible:*
  - *either each aggregate (SDL Blocks) shall be classified as domain, system or interface given;*
  - *or each aggregate is subdivided into parts that can be classified as domain, system or interface given.*

Similarity and types

In many cases it is best to define aggregates as types. We call such types *structure types*, as opposed to *object types* that define single objects with behaviour. Look for similarities, which will make type concepts. Re-examine components that are partially similar, but partially dissimilar.

- *Aggregates should be defined as types, except when they are singular and only serve the purpose of gradual approach to detail.*

Using library

- *Analyse the system such that it may be possible to recognise similarities with existing components. This should include both conceptual, structural and behavioural descriptions. Return to analysis if library search fails to produce the desired building blocks.*

Designing with SDL

- *When designing with SDL-92 bear in mind there is no problem in the first round just to identify one process type for each role that not obviously is associated with a domain Specific object. If it turns out that this role is more or less independent of other roles, then it may either stay as a process type or it may be turned into a service type in order to be combined with other service type. If it turns out that this role is highly dependent on other roles, then a merge/synthesis of process properties is needed (++ref 1112-5/6 - role modelling)*

Variability

The variability is an important part to consider.

- *Each object set in the environment and the content of a structure type shall have a cardinality range, and a member type with sufficient variability to cover the range of environments where it may be instantiated.*

It may sometimes be difficult to capture all variability required in one formal model of the application, because the languages we use have limitations. The way out of this problem is to model several application systems with different ranges of variability. This may sound like a very expensive approach, but it need not be. Provided that we have defined the various component types we need, to define another application system is a simple matter of defining a new top level structure. For the specification this means to define a new context model using existing component types.

- *Ensure that the variability expressed is sufficient for the range of system instances demanded from the market.*

- *If the full variability cannot be modelled in one system model, make several system models, and focus on the range of components needed.*

Evolution

- *Ensure that the modularity is sufficient that changes in services or interfaces can be confined in few modules.*
- *Ensure that services can be added or changed with minimal impact on (interaction with) other services.*

Framework

The focus on components will be even more important when a framework has been defined.

- *If the framework has been defined, make application types such that they can be used in the framework.*

Object modelling in general

The general guidelines for *Object Modelling* apply with the following additions:

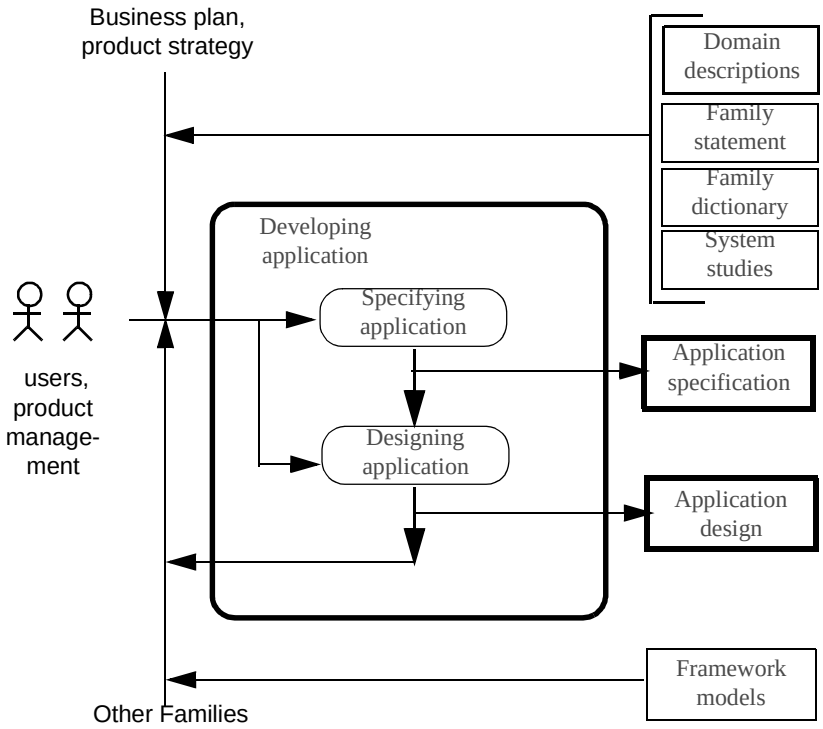
- Object classes with attributes, relations and connections:
  - *Define attributes by attribute types that are either predefined or locally defined.*
  - *Associate signal lists with communication links.*
  - *Turn communication links into signal routes or channels when designing in SDL.*
- Relations
  - *Stick to constructive relations.*
- Aggregation
  - *Always use real aggregation for active objects.*
  - *Use aggregation relations only for passive objects.*

Developing applications

What

Figure 6-29: Developing Application

Open figure



Inputs The main sources of input information are:

- Descriptions:
  - The system family statement will express overall goals for the application system.
  - The system studies will provide some sketches of the application system and also express initial requirements.
  - The domain descriptions will describe objects and properties that will go into the domain given parts of the application (more or less directly).
  - The framework (when it has been developed), will provide structure and interfaces that the application must abide.
  - Other families will provide components and possibly ideas for structuring.
  - Project documents and other documents within the company will provide additional information such as business plans and product strategies.
- People:

- Users and other domain stake holders can help to clarify what is unclear in existing descriptions and can help to make decisions (in particular regarding specifications and behaviour details). They will also come up with new requirements as their understanding and experience develops.
- Project and product management may give additional requirements and guidelines from the project and company point of view.

The quality of the various sources may vary a lot from project to project. It is of course recommended to put as much as possible on paper, but descriptions cannot entirely replace interaction with other people. Two things are certain:

- additional information will be needed from people;
- needs and requirements will not be stable, but evolve during the development.

What to do

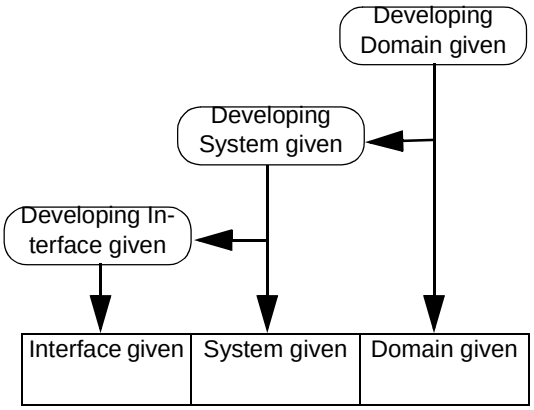
Application types are developed in two main steps: 1. the specification step and 2. the design step. As indicated in Figure 6-29 (p.6-106), this holds both for making the first model and for later evolution.

The specification step develops the application context and the required properties, while the design step synthesizes an application content that provides the properties. The content will be a structure of objects with behaviour. Therefore the design step will have two interrelated activities: structural design and behaviour design.

Both the specification step and the synthesis step are split into sub-activities that develop the three main parts of an application system: the domain given, the System given and the Interface given parts. Here the general approach is to work from the domain given towards the Interface given parts as illustrated in Figure 6-30 (p.6-107).

Figure 6-30: Developing the various parts of an application

Open figure



There are good reasons for this general approach:

- The needs originate in the domain, and domain given parts can be understood and modelled independently of the rest of the system, at least initially.

- The system given parts provide services on top of the domain given parts. Therefore the domain given parts is a precondition for the system given parts.
- The interface given parts are there to support communication with the environment for the domain and System given parts. The communication needs should be known before the Interface given parts are developed.

In terms of services, a good approach is to start with the high level services (required by the domain and the system) and work towards the low level interface protocols. This approach is consistent with the idea of layering and separation of concerns. It will promote a structure where changes in one part may be hidden from other parts.

Of course, it may so happen that some parts are more or less given by the initial requirements. For instance that an existing protocol stack or user interface shall be used. This does not alter the general approach to be taken, but reduces the amount of work required for that part.

Making application models

The following general approach to making application models is recommended:

Make Application

1. Making application specifications (p.6-110). This activity results in a context model (i.e. an object diagram, where the system environment is detailed while the system itself is depicted as a single entity, accompanied by property models specifying services and roles) and a partial content showing the domain given parts.
2. Making application structure (p.6-114). This activity develops the content structure (i.e. object diagrams that detail the structure of the content, supported by property models.) The leaf components in the structure are objects that will have a behaviour. For each object type continue by:
3. Making application behaviour (p.6-115) to define object behaviours that will satisfy the behaviour properties specified for each object type.

The approach is further detailed in Specifying applications (p.6-109) and Designing applications (p.6-113).

Evolving application models

The reason for evolution is that some new requirements shall be satisfied.

The same overall approach is recommended for making (the first) and evolving (the existing) application. The difference is that evolution is incremental and based on existing models. Our ideal goal is to be able to specify services independently and to add new services to the system with minimal impact on existing services and objects. We will strive to achieve solutions that are as flexible as possible, but realise that the ideal goal is not fully achievable today. Some redesign of existing parts will often be required. Normally new services, or service features, will interact with existing services. Feature interaction analysis will be needed to ensure that the new properties combine with the existing properties without any undesirable effects.

This activity is both simpler and more difficult than making (the initial) application model. It is simpler because so much is already in place and we only need to do an incremental addition or change. It is more difficult because even small changes or additions may have profound impact on the existing parts, if not prepared for.

Evolve application

Evolution is performed by:

1. Evolving application specifications (p.6-113) in order to update the specification and to analyse the impact on the design.
2. Performing an incremental synthesis of the application so that the new services are satisfied without undesirable interference with existing services: do this by:
   - Evolving application structure (p.6-115)
   - Evolving application behaviour (p.6-116)

Specifying applications

Inputs

Inputs to this activity are the same as for Developing application models as a whole, see Figure 6-29 (p.6-106). Be aware that system studies normally are performed before we start specifying the application. During System Studies, the main decision about system boundary, services and environment is made. As a result we have partial answers to questions like:

- what parts of the domain that shall be supported by the system;
- what shall be the system environment and interfaces;
- what shall be the main services;
- what are the requirements to evolution and instantiation?

If these questions have not been answered before, perform that part of System Studies first.

What to do

The objectives of application specifications are:

- to specify the external behaviour at an abstraction level where it can be understood and analyzed independently of a particular design;
- to specify the environment and the interfaces sufficiently to make the applicability (of instances) clear.

Therefore, a central issue is to identify objects in the environment that demand services independently of each other, and to describe these services.

The overall approach is as follows:

- *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 of.*
- *Describe the domain specific services in terms of text, 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 roles. Specify the corresponding service behaviour using roles and MSC.*
- *Add system specific entities to the active and the passive environment*
- *Only show parts of the environment that are related to the system.*
- *Identify the content parts that are subject to requirements.*
- *Sketch or outline the system structure using UML.*
- *Use real aggregation to illustrate how entities in the environment relates and are connected with parts of the system.*
- *If possible or relevant, define the interface behaviour of each role in the system and in the environment.*

Making application specifications

As a first step consider what language to use, see Language choice (p.6-116).

Then start to make a type model for the entire application system by representing the system type as one entity type (Class in UML).

Specifying the domain given objects

1. *Represent all the domain Actors and Helpers needing to be served by the system as (active) objects in the environment.*
2. *Represent domain Helpers to be implemented by the system as active objects inside the system.*

Make Application specification

1. Make a context diagram where the system is represented as one unit. The environment and the content are then detailed by:
2. Specifying the domain given objects (p.6-110). This means to specify domain given objects in the environment and in the content.
3. Specifying the system given objects (p.6-111). This means to analyse the need for system given services on top of the domain given services. Specify additional system given objects needed in the environment.
4. Specifying the services (p.6-112). Specify domain given and system given services using service property models.
5. Specifying the interface given objects (p.6-112).

3. *Represent stake holders, subject entities and helpers the system need to know as (passive) objects in the environment.*
4. *Start to define object types for all the objects identified so far. For each type, make a context diagram showing the type environment.*
5. *Relate domain given types to the corresponding domain types.*

As a result we have identified the domain given parts of the environment and the system, and we have started to define object types.

Specifying the system given objects

Consider the system from an enterprise viewpoint. Think of the system in the wider context of an enterprise where it operates and provide services together with users and other systems. What are the combined services and what additional requirements to the system family can be identified in this perspective?

1. *Add (active) objects representing system given* actors *and* helpers *needing to be served by the system; place them in the environment and connect them to the system by communication links.*
2. *Add (passive) objects representing system given objects and relationships the system need to know; place them in the environment.*
3. *Add the context part of type models for the system given types.*

As a result we have made a context object model for the system as a whole where the complete system environment is represented, and context models of the component types used in the system context.

We always consult the library of existing component types and pick existing types whenever possible.

Specifying the services

From the domain models we already have domain services. But the domain services will now interact with the System given services. Our first step is to consider each domain service in the light of System given objects and services.

1. *Refine each domain service so it takes into account interaction with System given objects (or roles) and services.*
2. *Define each System given service.*
3. *Make new component type models more complete by specifying relations to service roles they play.*

This activity shall take existing service models into account. For instance, if an existing component type is used, the services it provides may be taken for granted, and the new services may build on it. It may also happen that a new version of existing services are needed. It is only in rare cases one is free to specify all services from the bottom.

Once the services are specified, we have a (complete) context model for the system given and domain given parts of the application system. We also have context models for each of the object types used in the environment, and we have service models. We may now analyse the model for completeness and consistency, and we should involve users and other stake holders in a validation.

Specifying the interface given objects

The Interface given parts should be kept separately from the rest by layering. Consider the physical interfaces and add interface specific parts to the environment, or to the system as appropriate.

1. *Make a type definition for each kind of interface object.*
2. *Specify the interface behaviour for each layer of interface protocols:*
   - *define the interface roles using UML;*
   - *make a textual explanation;*
   - *define the role behaviour using MSC. Make one MSC for each independent initiative and describe the main course of behaviour that may follow.*

It is recommended to describe the behaviour of each interface using an interface role diagram and at least a number of MSCs. Wherever practical, develop a more complete interface behaviour using SDL.

Evolving application specifications

Evolution may take place in all parts: domain given, system given and interface given. The big difference from making a specification is that we now shall add something new or change something existing. We must therefore consider the impact on the existing.

Evolve Application specification

1. Specify the new services using property models.
2. Analyse the impact on existing services and object models.
3. Specify new objects required in (the various parts of) the environment (if any).
4. Change existing services and object models as

Note that application specifications are made for the first time before any framework is defined (normally). However, when they are subject to evolution, it is more likely that a framework is in place. In that case we may have abandoned the complete view on application systems in favour of a more component oriented view. This means that our task is to find what component types are affected, and to modify each of them separately.

Designing applications

This activity takes the application specification and synthesizes a design that will provide the required properties. It consists of two parts: Designing application structure (p.6-113) and Designing application behaviour (p.6-115).

Designing application structure

What

In SDL terms this activity is concerned with the identification of blocks and block types in order to divide the system into convenient “components”. Blocks may be used to make a layered system. One may chose the extreme that the interface-, system- and domain given parts are represented by one block each, but this may be too inflexible. Blocks are aggregates and should be selected to fulfil the aggregation purposes, see General application guidelines (p.6-102).

Some of the object types and properties developed in this activity may turn out to be quite general and common to many system families in the domain. In such cases the domain object models and domain property models should be harmonised, i.e. updated, to include these object types and properties.

Property descriptions will in this activity be associated with the identified objects of the system, while property descriptions in the domain analysis and requirement specification activities more often are property descriptions of the whole system.

"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 context is specified with service roles and the interface roles, the task is simpler: just "find" actors for all the roles. With a slight adaptation of an old saying we may say: *Tell us who is in your environment and we will tell you who you are (what your content is).*

What to do

Making application structure

The following strategy assumes that we know the active environment and the roles that it requires from the system.

1. *Mirror the active, service needing environment. Identify the objects and interfaces in the service needing part of the environment that behave independently and need to be served concurrently by the application. Mirror each object by a serving agent inside the application.*
2. *Find actors for all service roles. First analyse the service roles defined in the property models and find actor objects for those that shall be played by the application. If possible, assign the roles to objects already defined, otherwise introduce new objects.*
3. *Introduce actor allocators when roles are dynamically assigned and there may be contention for the actors. Services invoked by different, concurrent users, may contend for access to the same actor and will need to be coordinated. Each set of equivalent actors should have an actor allocator object.*
4. *Find types. Look for existing types that can play the roles. Use existing types if possible, otherwise identify new types to be defined in application type design.*
5. *Consider the interface given part of the environment and introduce correspondingly layered interface objects inside the application.*
6. *Mirror the passive environment. The passive objects in the context represent context knowledge the system need to have. This knowledge must be mirrored by data or active objects inside the system. Therefore, allocate the passive objects in the context to active objects in the body. This may result in some new objects being defined. The same principle applies recursively for every object inside the system too. If necessary introduce new active objects to hold the data.*
7. *Analyse the behaviour. Make MSCs detailing the internal interactions and check that the structure will give effective behaviour definitions.*
8. *Analyse the block structure. Find a suitable block structure that will satisfy the block purposes.*
9. *Analyse variability and adaptations required.*
10. *Verify and validate. Validate the functionality against requirements using simulation and MSC verification. Validate external and internal interfaces. Ensure that every role is properly played in the system using the role analysis principles of the method. (During type design apply the rules constructively during behaviour composition.)*
11. *Iterate and adjust the structure until the general rules and guidelines for application models are satisfied (see* General application guidelines (p.6-102)*).*

For each aggregate (SDL block); the same rules apply as if it was a system. The decomposition process continues recursively until objects with behaviour (SDL processes) are reached. The structuring of blocks results in a tree of blocks. The leaves of this tree will contain the real actors of our system, the processes.

The principles behind this process are elaborated in synthesis.

Evolving application structure

t.b.d.

Designing application behaviour

What

The task at hand is to define object types where the content is a behaviour definition and possibly some attributes. The goal is to describe the behaviour clearly and concisely in state oriented form.

Object design tries to synthesize object behaviours that will satisfy the desired behaviour properties (services). It will, however, often be so that several objects must be involved in satisfying a single property (service) of the system.

What to do

Making application behaviour

The central issue is to synthesize the behaviour. This can be done in three ways:

- Synthesizing behaviour from properties (p.6-115);
- Synthesizing behaviour from environment behaviours (p.6-115);
- Synthesizing behaviour from informal requirements (p.6-115)

Synthesizing behaviour from properties

The behaviour properties express fragments of behaviour that somehow shall be part of the final object behaviour. Behaviour design will compose these behaviour fragments into complete behaviours. This may be done in two steps, see MSC to object behaviour:

1. *composition of behaviour fragments from MSCs to more complete role behaviours;*
2. *composition (with a certain amount of adaptation) of object type behaviour from role behaviours.*

Synthesizing behaviour from environment behaviours

In some cases the behaviours of objects in the environment are defined, and the task is to design a behaviour that will serve these. The resulting behaviour shall be a kind of “mirror” image of the environment behaviours.

Synthesizing behaviour from informal requirements

Sometimes no behaviour properties are formally expressed, only some informal description of functionality exists. In this case, the preferred approach is to first formalise the requirements in MSCs and then synthesize the behaviour as described above. Alternatively, produce state transition diagrams (SDL process graphs) directly:

1. *Identify the main control states that can be distinguished form outside. At least identify an initial state.*
2. *Link the main states together by transitions.*
3. *Consider each service that shall be performed and define its external state transition behaviour. Add states if necessary.*
4. *When all the services are defined, go through the diagram state by state and check that all possible inputs are properly treated. Apply the rules for input consistent behaviour, see ++*
5. *Add the necessary data declarations and parameter handling.*

Evolving application behaviour

t.b.d.

Summary of static application rules

Language choice

- *In general use UML for the context object model:*
  - *Model the application system as a Class. Represent the environment as object sets with specified variability. Describe associations and communication links in the environment and with the system.*
  - Model the content (if any) as object sets.
  - *Model the Class context for objects in the system environment and content using separate diagrams.*
- *Where reactive behaviour is important and general relations (associations in UML) are unimportant, SDL should be used:*
  - *Model the application system as a Block Type in SDL if SDL is used for the complete application system. Represent the environment as gate constraints.*
  - Model the content (if any) as Block sets.
  - *Model the types context for Block types in the environment ant the content.*
- *There shall be a separate property model for each service and interface containing:*
  - *a role structure expressed in UML;*
  - *explaining text;*
  - *a MSCs for every initiative and its most important responses.*
- *Associate the roles of property models with the objects and types/classes of object models.*

Domain given parts

- *Every domain Actor and Helper needing to be served by the system shall be represented as an (active) object in the environment.*
- *Every stake holder, subject entity and helper the system need to know shall be represented (as a passive object) in the environment.*
- *Every domain helper the system shall implement shall be represented in the (domain given) content.*
- *Every domain relation(association) the system need to know shall be represented in the environment.*
- *Every domain communication link the system need to handle shall be represented.*
- *Every domain given object shall have a type (class) which is related to the corresponding domain type (class).*
- *Each domain given service shall be described independently of interfaces using service roles.*
- Each domain given type shall have relations to the domain given service roles it participate in.

System given parts

- *Every entity that need to interact with the system shall be represented in the system environment with a communication link to the system.*
- *Every entity and relation the system need to know shall be represented in the environment.*
- *Each service shall be described independently of interfaces using service roles.*
- *Each system given objects in the environment shall have a type definition with relations to each system given service role it participate in.*

Interface given parts

- *For every communication link with the environment the interface given parts shall be specified.*
- *For each type of interface, there shall be a property model containing:*
  - *interface roles using UML;*
  - *a textual explanation;*
  - *the role behaviour expressed using MSC. There should be one MSC for each independent initiative and each main course of behaviour that may follow.*

General considerations

- *Every object in the environment shall have an independent threads of behaviour and take initiatives independently of other objects.*
- *Every aggregate in the environment shall fulfil some of the desired aggregate purposes.*

- *Each object set in the environment shall have a specified cardinality range, and member types with sufficient variability to cover the range of environments to be supported by members of the family.*
- *If the full variability cannot be modelled in one system model, make several system models, and focus on the range of components needed.*

System Content

- *Avoid to reveal more content in a specification than necessary for the external use.*
- *If more content need to be specified, then split the specification into an external part and an internal part. The external shall be sufficient for assessment and use.*
- *If the system body shall be described in some detail, then use the guidelines for* Object Modelling. *Use the same notation for content structure as for context structure (most likely UML) in specifications.*

Summary of dynamic application rules

Domain mapping

- *Relate domain given types to the corresponding domain types.*

Domain relationships can be either Inheritance or Inspired-by. Inheritance can be expressed in the object modelling languages, while Inspired-by must be added outside the languages as annotations.

Domain stake holders

- *Consider all domain stake holders:*
  - *Represent all the domain Actors and Helpers needing to be served by the system as active objects in the environment.*
  - *Represent domain Helpers to be implemented by the system as active objects inside the system.*
  - Represent stake holders, subject entities and helpers the system need to know as passive objects in the environment.

Interface layering

- *Use a layered approach to behaviour properties. Separate the interface given parts from the domain and System given.*
  - *make property models that abstract from the interface given parts and focus on the domain and system given parts;*
  - make additional property models where the interface given parts are included.

Environment

- *Decompose the environment into objects that have independent threads of behaviour and take initiative independently of each other.*

- *Organise the environment in aggregates that fulfils the desired aggregate purposes.*

SDL system and environment

For the elements at the periphery of your concern, place them inside the system if you wish to describe their behaviour in detail. If you are merely interested in their signal interface, place them in the environment which means they will not be identified explicitly in SDL.

Framework

Content and scope

A framework is an abstract system or a collection of (large) system component with two parts:

- a redefinable application;
- a configurable infrastructure that takes distribution into account, and contains all additional behaviour and supporting functionality needed to support the application in the concrete system.

The application part is normally rudimentary in the framework. To make a complete abstract system from a framework the application is redefined and the infrastructure is configurated.

In this chapter we shall describe:

- What is a framework (p.6-121): an introduction to the notion of a framework.
- Framework reference model (p.6-122): the system model assumed in frameworks.
- Framework models (p.6-126): how to describe frameworks.
- Developing framework (p.6-143): how to develop framework models.

Framework models are important because:

- they model implementation dependent behaviour and distribution common to many applications (the infrastructure);
- they support reuse of high level designs and not only single classes;
- they contain the common infrastructure in a way that can be easily reused with a range of applications;
- they allow application development to be separated from infrastructure development (a kind of layering);
- they are instantiated into complete abstract systems that:
  - document the complete system behaviour as it will be (or has been) implemented;
  - are complete sources for automatic code generation.

Once a framework is defined, application development may proceed as an independent activity concentrating on application issues.

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 ...

What is a framework

The Free On-line Dictionary of Computing defines a “framework” as:

“In object-oriented systems, a set of classes that embodies an abstract design for solutions to a number of related problems.”

A tutorial on designing frameworks says:

“Frameworks are reusable designs for an application or a subsystem expressed as a set of classes and the way that instances of these classes collaborate.

A framework describes not only how to partition the responsibilities of a system among its components, but also how to think about a problem. It is therefore not only a way to reuse code, but a way to reuse design and analysis information, as well.

Frameworks are difficult to design because they are abstract. Framework designers must look at many concrete applications to ensure that the abstractions that they are designing make sense. Frameworks are difficult to learn because the user of a framework must adopt the collaborative model of the framework. It is usually not possible to learn a framework one class at a time, but instead several classes have to be learned together.

Nevertheless, there is a great advantage to learning a well-designed framework, and mature frameworks (like some of the user interface frameworks) can provide order of magnitude increases in programmer productivity.“

The main property that distinguishes a library from a framework is that a framework embodies a design of a type of systems, while a library just is a collection of related classes. An elaboration of this distinction is given in Table 6-1 (p.6-121).

Table 6-1: Differences between libraries and frameworks

Library	Framework
The application uses library classes, but the library knows nothing about the application	The framework knows about the application and uses application classes.
No predefined system structure. The system is entirely defined in the application.	Provides structure. The system is (partially) defined in the framework
No predefined interaction	Defines object interaction
No default behaviour	Provides default behaviour

Examples on these differences will be given below.

Classical examples of frameworks like window systems are defined as a set of related classes, and are normally implemented in C++. The structure of these frameworks results from dynamically created objects that are kept together by object references. The event loop that dispatches mouse and keyboard events to window objects will use a list of currently active window objects. This list will typically contain different window objects, with the common property that they react on these events. By calling virtual procedures, it is then up to the user of the framework to redefine these procedures to what is special for the application.

The notion of framework is not special for TIMe. What is special for TIMe, however, is that this idea is adapted to SDL. TIMe advocates the structuring of systems into frameworks and gives guidelines for how frameworks can be defined in SDL. 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.

In TIMe a framework is a system type/class or a collection of (large) component types/classes, 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 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 and again for each new 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.

The structuring of a framework may have one or more of the following goals:

- simply to give the system a generic structure, allowing *all* parts of the structure to have specific contents provided in a specific system;
- to structure the system so that the *application specific* and the *implementation specific* parts are separated allowing special applications to be "plugged" into the framework by providing the contents of the application parts.

TIMe emphasises the second goal. How this may be done for designs made in SDL, is explained in How to define a framework using SDL (p.6-132).

One reason for designing frameworks is that this has turned out to be the most effective way of reuse. Another is that the framework helps to simplify application evolution and system instantiation.

Framework reference model

Application and infrastructure

In the Application reference model (p.-99) the "pure" application is in focus without considering how it will be implemented. When the system is implemented, the application system may be distributed over several physical nodes and needs to be supported by an infrastructure with functionality for:

- distribution;
- communication;
- error handling;

- etc.

Complete abstract system

This additional functionality may be modelled in a complete abstract system which can be seen as having two layers: an infrastructure layer and an application layer. In the complete abstract system the application has been distributed in the same way as in the concrete system. In the Infrastructure part we find objects that support distribution, system administration and other facilities not directly related to services provided by the application. The infrastructure part contains additional behaviour needed to fully understand, simulate and analyse what the system does (the complete system behaviour).

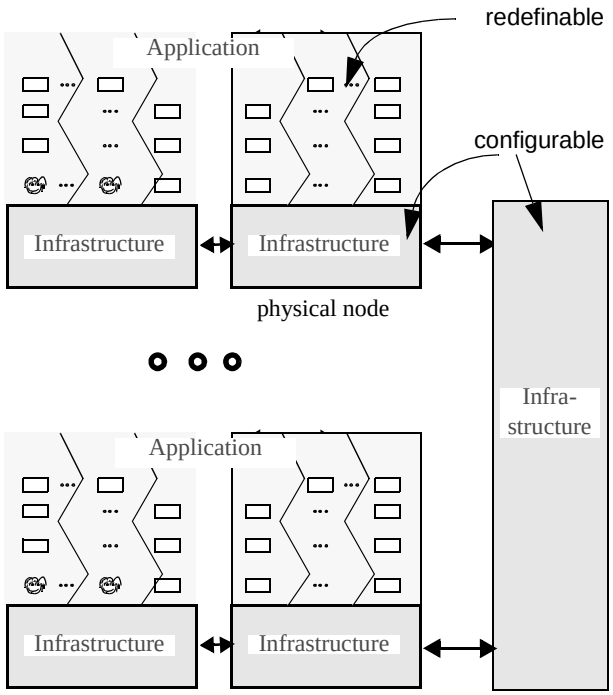
Model

A reference model that takes these aspects into consideration is shown in

Figure 6-31 (p.6-123).

Figure 6-31: Application framework reference model

Open figure



Here the Application reference model (p.-99), described in Figure 21 (p.-99) [openfig] is distributed on top of an 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 systems with different applications it is desirable not to change or even consider the Infrastructure part (besides what it offers). Adding or changing services should mainly be performed in the application, leaving other parts unchanged.

On the other hand it is sometimes desirable to change the implementation platform without needing to modify the application. It is desirable that an application may survive several platform generations and thereby provide better return of investment.

A framework defines the composition of the Infrastructure parts and application parts in such a way that they can be changed independently.

Framework infrastructure

The Infrastructure is structured similarly to the concrete system. If the concrete system is distributed, the infrastructure will be distributed too. If the concrete system is centralised, then the Infrastructure will be centralised too. (Therefore, the Infrastructure cannot be designed before the Architecture (of the concrete system) has been designed.) There are several reasons for this similarity:

- The underlying architecture has a strong impact on the infrastructure functionality and must therefore be reflected in its description.
- By identifying physical nodes, it will be easier to see how the application is distributed, and to generate complete code for the framework (infrastructure and application) running on each physical node.
- The underlying architecture will have some impact on the application too. Physical nodes may for instance, fail independently. They are linked by unreliable channels, and the message routing in the system must take distribution into account.

In general an Infrastructure will contain abstractions of:

- The structure of physical nodes.
- Networks for communication between physical nodes.
- Protocols for communication between physical nodes.
- Routing facilities that enable application objects to communicate transparently across the physical nodes.
- Facilities for initial configuration and later reconfiguration of the system.
- Error handling facilities.

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.

For the Access Control system the fact that validation shall be performed by a central computer is an infrastructure issue, like the need to support remote communication, with additional protocols.

Framework application

The framework will contain redefinable application objects. In a framework instance these are redefined using application types that shall be defined in the Application models (p.-106).

When the application is developed for the first time, the Infrastructure is normally not known, and therefore not taken into account. As soon as the infrastructure is known, application development shall take the infrastructure into account. This means that:

- signal sending must use the addressing scheme and routing facilities of the infrastructure. To achieve distribution transparency the application objects must communicate via the infrastructure and not directly with each other, even when they are localised on the same physical node;
- the error possibilities of the infrastructure is taken into account;
- error handling uses the facilities of the infrastructure, if they exists;
- That configuration is solved using infrastructure facilities, if they exists.
- That the application types have a size and content suitable for instantiation within physical nodes in the framework. They should not be too large for that.

Framework models

Objectives

The purposes of framework models are to:

- describe the architectural context for applications:
  - how the application is distributed;
  - what the communication interfaces are;
  - how routing and addressing of application messages (SDL signals) are done;
  - additional facilities available to the application, e.g. error handling, dynamic configuration;
  - how application behaviour must be adapted;
- describe the infrastructure:
  - show the overall communication links and routing;
  - describe protocols;
  - describe additional facilities like error handling, configuration support, etc.
- simplify application evolution;
- simplify definition of complete abstract systems with the same infrastructure, but different applications.

Framework instances serve to:

- be a source for complete code generation (of the application and the infrastructure);
- be a documentation of the complete behaviour and to enable analysis and simulation of the complete behaviour.

Framework models serve to define complete behaviour when taking implementation specific features into account. Framework models are made once for a family, and applied many times (one for each different application).

Framework rule

- *Whenever an abstract system can be split into an application layer and an infrastructure layer, a framework model should be developed that serves to simplify the definition of new systems.*

What

Framework models contain different kinds of object type models:

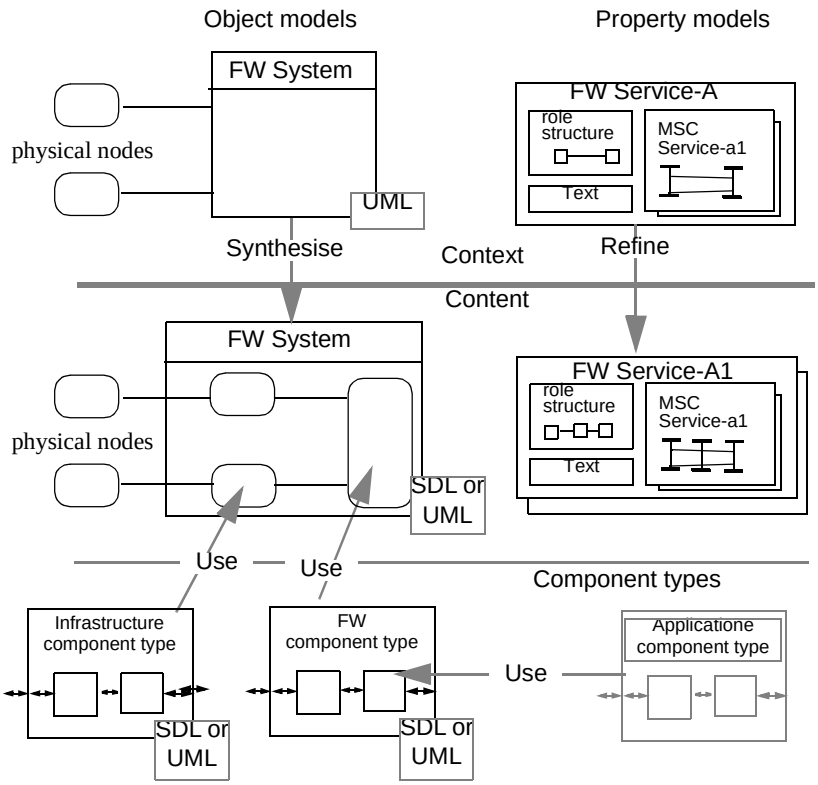
- Framework types:
  - System types. These are structure types defining the overall framework structure. They are formalised whenever it is possible and practical to do so and will contain both infrastructure and application parts.
  - Component types. These are structure types for components that are used within framework systems. They may contain both infrastructure and application parts.
- Infrastructure types. These are structure and behaviour types for “pure” infrastructure components, i.e. components without application parts.

The application part of the framework types will be virtual application components. All the application types are defined in the Application models (p.-106).

Framework models, as other models, have a specification part and a design part, see Figure 6-32 (p.6-127).

Figure 6-32: Framework models

Open figure



It may sometimes be difficult to express all the variability needed formally in one type model. It may also happen that some components are defined using SDL and other components are defined using UML. In such cases it may prove more practical to emphasise component types that are easy to compose rather than complete framework system types.

When necessary, rules for mapping applications to the framework shall also be defined.

Framework model content

Framework specification

The specification part shall concentrate on the context of the framework and the external framework properties, see Framework specification (p.6-128).

Framework design

The design part shall concentrate on the content of the framework, see Framework design (p.6-129).

Framework languages and notations

As illustrated in Figure 6-32 (p.6-127), the main languages to use are MSC for property models and UML combined with SDL for object models.

SDL is the main language for control behaviour, while UML will be used for other aspects (such as user interfaces and data-bases). Typically UML will be used in the specification part, and possibly for top level design structuring. SDL will take over where the main thing is to model reactive behaviour.

In some cases it will be possible to express the framework object models completely in SDL.

For description of interaction properties, we stick to MSC even if UML is used for the Object Model.

Framework property models will focus on the infrastructure properties. In addition to specifying the infrastructure services and interfaces using text, role diagrams and MSC, they will specify requirements to physical distribution, routing, configuration and similar issues that are better expressed in text and illustrative figures than in MSC and role diagrams.

Framework specification

Objectives A framework Specification serves to answer questions like:

1. What infrastructure services are provided?
2. How is the environment physically distributed?
3. How is the content physically distributed?
4. What is variable?
5. How are instances defined.

What Specifications are developed first for the framework system as a whole and later for each framework- and infrastructure component type.

For the framework as a whole the following issues shall be considered:

- the physical distribution of the environment, if any;
- requirements to physical distribution of the application system;
- requirements to the infrastructure part:
  - network requirements;
  - protocol requirements;

  - communication and routing requirements;
  - configuration requirements;
  - error handling requirements;
- the variability in the system family, and how system instances are to be configurated and built. This includes variability in the application as well as in the infrastructure;
- requirements to dynamic change.

Framework specification content

Framework object model

The object model will describe:

1. the system context detailing the environment and the interfaces using physical grouping and distribution as main structuring criteria.
2. the parts of the content that are externally visible and relevant to the specification, also structured according to the physical world.
3. component type specifications for component types that are identified.

Framework property model

The framework property models will contain:

1. service models for infrastructure services;
2. interface models for infrastructure interfaces;
3. textual specification of other properties:
   - network requirements;
   - protocol requirements;
   - communication and routing requirements;
   - configuration requirements;
   - error handling requirements;
   - the variability in the system family, and how system instances are to be configurated and built.

Framework design

Objectives To describe the design part of framework models that:

- satisfies the specification;
- has structural similarity with the architecture;
- supports a (layered) separation between infrastructure and application;
- can be easily instantiated to perform different applications.

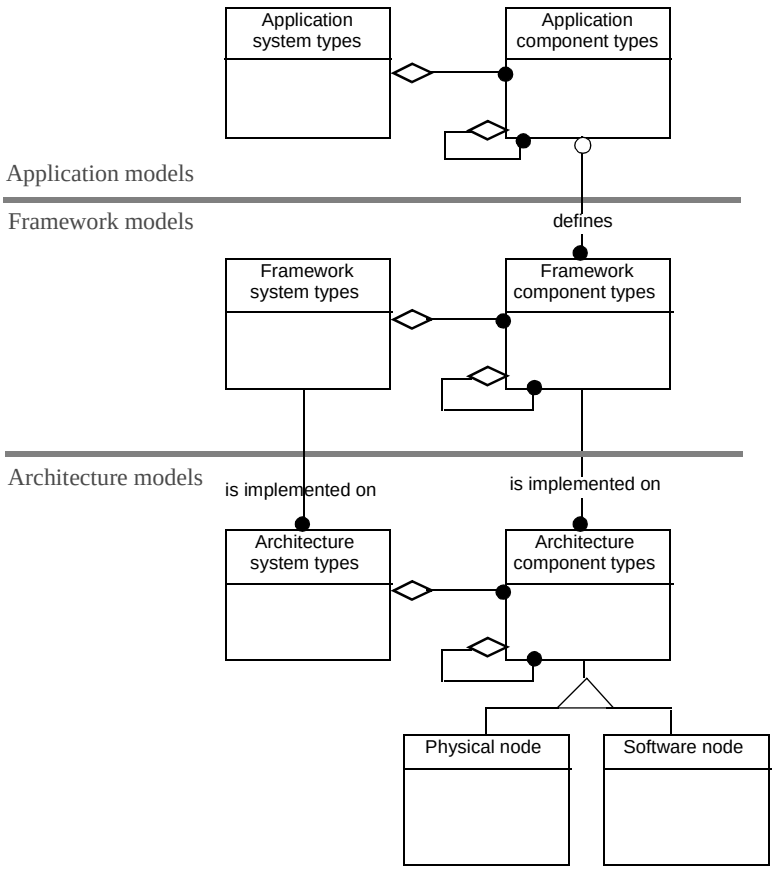
What The context part is fully covered in the specification. The design part define the content structure and the types of framework component and infrastructure components that are specific to the framework.

Framework relationships

How the application is related to the domain and the framework is illustrated in Figure 6-33 (p.6-130).

Figure 6-33: External Framework relationships

Open figure



Relationships framework - application

The application part of the framework should use types that are defined in the Application models (p.-106).

Framework application relationship

- *Use only types defined in application models in the application part of a framework.*

Harmonising framework - application

The application will not be visible as a system in the framework, but its component types will be used. However, if the application types shall be applicable in the framework they must obey the rules of the infrastructure. This may require some adjustment compared to the types initially developed (before the infrastructure was known).

Framework application harmonisation

- *Make sure all application parts of the framework are defined using "pure" application types.*
- Make sure the infrastructure is transparent to the application parts.
- *Make sure every application type:*
  - *complies to the infrastructure requirements;*
  - *is a suitable distribution unit in the infrastructure.*

Relationships framework - architecture

Framework architecture relationship

- *There shall be a one to one relationship between physical nodes in the Architecture and corresponding components in the framework.*

Harmonising framework - architecture

Framework architecture harmonisation

- *Whenever there is any change in the Architecture (due to platform changes or structural changes) update the framework structure accordingly.*

Relationships framework specification - design

The considerations here are the same as for Relationships application specification - design (p.-124).

Harmonising framework specification - design

The considerations here are the same as for Harmonisation application specification - design (p.-126).

General framework guidelines

While the application models are structured primarily to achieve readability for the human, framework models are structured primarily to:

- reflect the Architecture;
- describe the complete behaviour;
- simplify evolution and production after the initial development.

The General application guidelines (p.-126) hold also for framework models. But there are additional rules.

Framework system type

- *It will often be difficult to define a complete framework for the entire system family which will be easy to configurate in SDL. In those cases, do not consider the entire system except for simulation or analysis. In stead define a block type for each kind of configuration unit (e.g. a physical node) and configure these separately. Perform configuration and building of complete systems at the implementation (design) level.*
- Use SDL system types only when
  - *context parameters are sufficient for instantiation;*
  - *the block structure is static;*
  - *and the need for dynamic configuration and change can be fully handled by process behaviours.*
- *Otherwise focus on Block types, and use these either to compose a set of different system types using SDL or to compose systems outside SDL (using either UML, a configuration language, or implementation level mechanisms like UNIX make.)*

Dynamic change

- *Dynamic changes are not supported by SDL (apart from dynamic creation of predefined process types within predefined process sets). Therefore, dynamic changes must be handled outside SDL, in the implementation. For systems where dynamic changes are important, the framework will consist mainly of a library of types on a granularity suitable as change units.*
- *Every substructure that may be changed or moved dynamically should be defined as a structure type and be addressable as a unit.*

Dynamic configuration

- *If members of a component set (block set or process set in SDL) need to be configurated differently, then either split the set into subsets with identical members or use dynamic configuration.*

Distribution transparency

- *Introduce routing objects and addressing that hide the physical location from application objects.*
- *Ensure that application objects communicate using this scheme.*

How to define a framework using SDL

The distinction between a library and a framework is in SDL directly reflected by the concepts of *package* and *system type*. In SDL terms a library can be expressed by a Package, while a framework must be defined by a System type. The essential difference is that a library only defines types and not instances (and this holds for a Package in SDL), while a System type can have both.

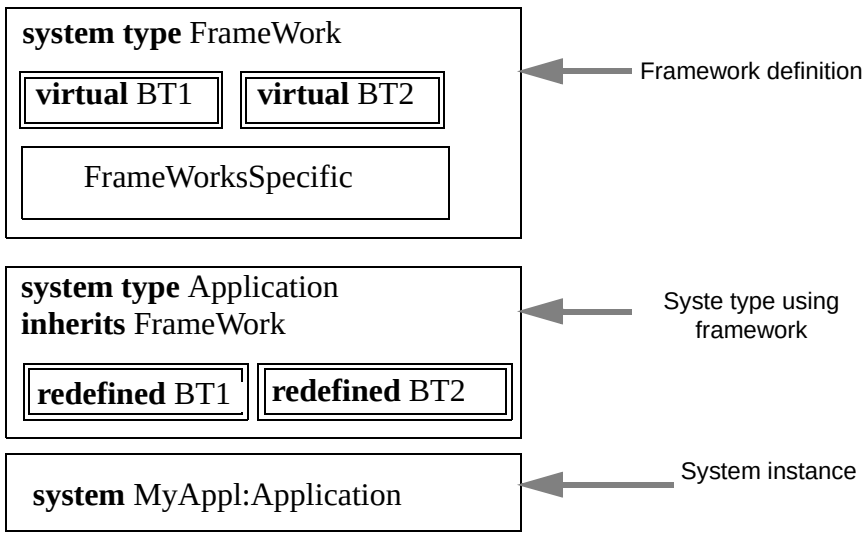
A System type defining a framework will typically have instances that cover parts of the structure, and it will assume that the actual application of the framework will provide application specific instances. A framework can be based upon libraries, and the system type will be part of a Package.

System types and Block types have so much in common that if the desired approach is to develop Block types and use these for composing systems, then many of the framework techniques explained below also apply to Block types.

A framework system type is defined by a system type or block type in SDL, see Figure 6-34 (p.6-133). This type will have virtual types for the application components, and these will be redefined when defining a complete abstract system.

Figure 6-34: Framework definition and use

Open figure



Normally the framework system type will have framework components that are blocks with some framework specific structure. The block type BT1 may e.g. consist of an infrastructure part and a virtual type that represent the application specific part of BT1.

How then are application specific parts included in the framework? In general this is done by redefining virtual types, e.g. virtual block types, virtual process type and virtual procedures. There are two different ways to treat application specific instances:

- Application specific instances are specified as part of the framework (p.6-133), as fixed instances/instance sets of the virtuals;
- Application specific instances not specified as part of the framework (p.6-136), but their generation is anticipated.

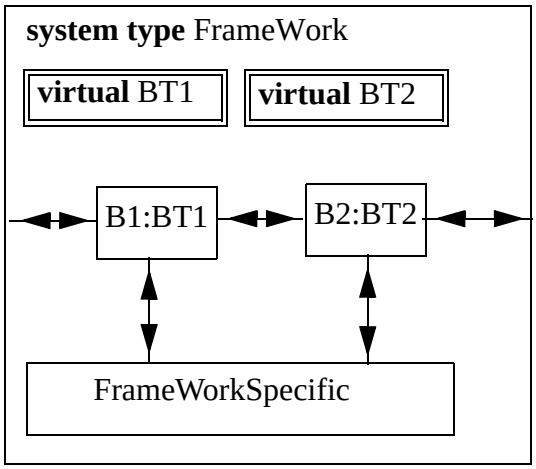
Application specific instances are specified as part of the framework

In this case it is only necessary to redefine the virtual application types - the instances/ set of instances and their position in the system structure is already part of the framework definition.

The framework system type in this case has the form illustrated in Figure 6-35 (p.6-134),

Figure 6-35: Framework definition with predefined instances

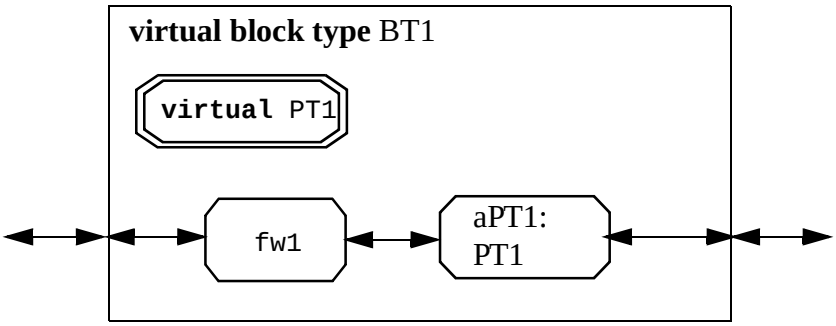
Open figure



and it is repeated for the block type BT1 (Figure 6-36 (p.6-134)) in order to show predefined process (sets).

Figure 6-36: Virtual block type as part of framework, with application specific instance specificed

Open figure



In this case the frameworks defines the structure of instances and instance sets with their connections in terms of channels and signal routes. In a complete application system the properties of the instances are provided by redefining the virtual types (using application types).

In order to define the structure of instances and their connections in the framework, some minimal interface definition for the instances must exist that cannot be redefined when redefining the virtual types. The rule of SDL is that redefinitions of the virtual types (here BT1, BT2 and PT1) must be subtypes of the constraints of the virtual types. The default is that the constraints of the virtual types are the type definitions themselves. This is, however, not enough to assure that the interfaces are the same, as it would then be possible to add new gates and new signals to the signal list of existing gates. Addi-

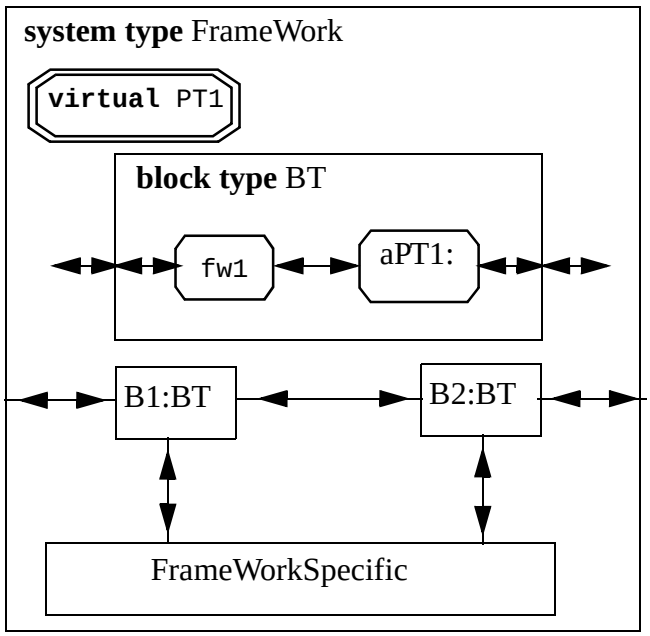
tional rules state that a redefinition of a virtual type must have exactly the same interface as the constraint of the virtual type. Before using this approach you should be sure that one can live with this rule.

A typical example on this kind of framework is the handling of protocols. The fw1 process will be the interface to the protocol, and it will present itself to the application specific PT1 process independently of the protocol. This scheme may be generalised: if e.g. both block type BT1 and BT2 will have the same protocol handling part, then this may be expressed in a common super block type of BT1 and BT2.

If the structure is so that the same virtual process type is used to make process sets in many parts of the system structure, then the virtual process type should be moved to the system type, see Figure 6-37 (p.6-135). In this way it can be redefined at *one* place as part of the system subtype and have implication on many parts of the system.

Figure 6-37: Application specific virtual type at system type level

Open figure



An alternative structuring of the system is provided in Figure 6-37 (p.6-135).

A framework example for the Access Control system is given in Figure 6-50 (p.6-150) and Figure 6-51 (p.6-151).

We know that AccessPoint is used both in LocalUnit and ClusterUnit (see Figure 6-51 (p.6-151)). By defining it at system type level, a redefinition in a system subtype will imply that (as desired) AccessPoint in both LocalUnit and ClusterUnit will get the same redefinition. With the AccessPoint as a virtual type at system level and Cluster as a non-virtual type we express that the AccessPoint of Cluster objects can be redefined, but not other parts of Cluster.

Application specific instances not specified as part of the framework

In this case the framework only consists of general types that may be used for the construction of the application part. The framework specific parts may either be instances or also just represented by types. Framework components will then need to create objects (in this case processes) according to application specific classes.

SDL is special in the sense that processes are part of process sets and that creation of processes is done by referring to the name of the set and not to the name of the process type. It is therefore not enough that the framework specific process types are defined in the same scope (e.g. a package or a system) as the application specific types for them to create instances application specific instances - they must have means for referring to the process sets to come.

This may be done in two different ways: by using context parameters or by using virtual procedures.

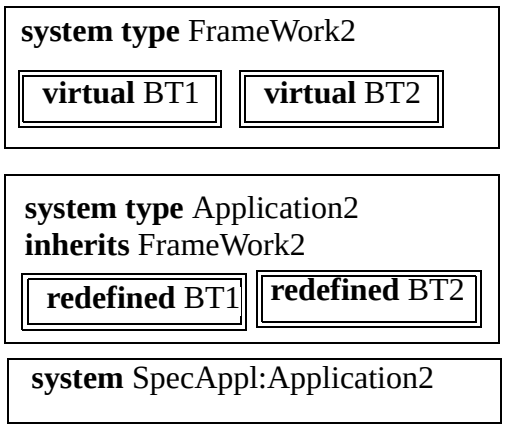
Context parameters

The general types of the framework that have to create process instances according to application specific types do this through process context parameters. This works in SDL because an actual process context parameter is a process set and not a process type.

The scheme is illustrated in Figure 6-38 (p.6-136), Figure 6-39 (p.6-137) and Figure 6-40 (p.6-137).

Figure 6-38: Framework with no instances

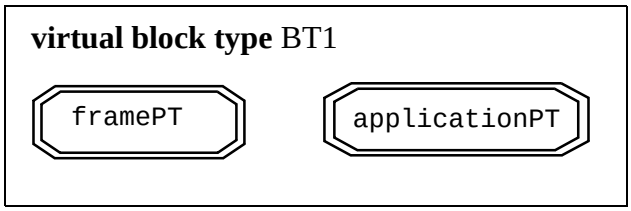
Open figure



Within the block types, there will be general process types (*framePT* and *applicationPT*) that are used as supertypes in specific systems inheriting from *FrameWork2*. In addition the block types may specify instances of other process types, and these are the only predefined instances of this kind of framework.

Figure 6-39: Virtual block type

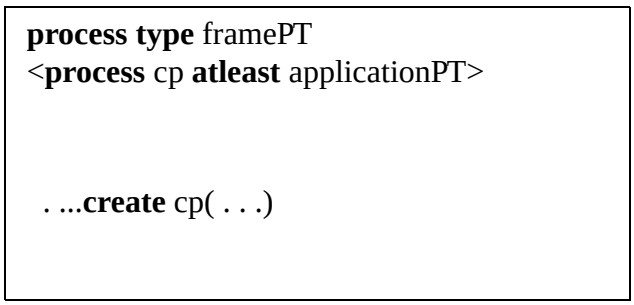
Open figure



The *framePT* process type will have a process context parameter that is constrained by applicationPT, see Figure 6-40 (p.6-137). The idea is that a particular system will provide its specialisation of *applicationPT* and its process set, and by the context parameter the *framePT* processes will create instances of the specialisation of applicationPT.

Figure 6-40: Creation of application specific processes

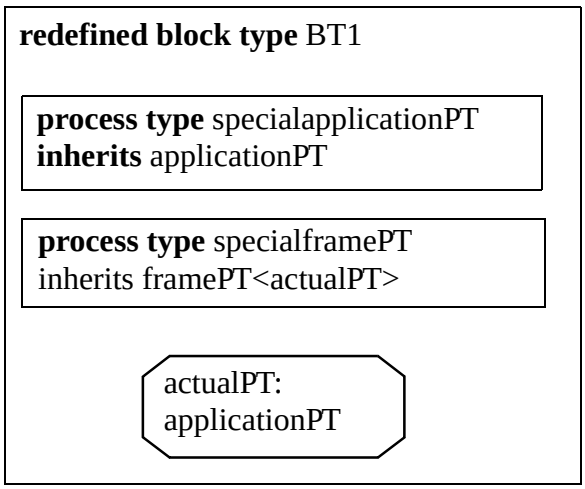
Open figure



The final system type will introduce the redefined block types with appropriate process sets, see Figure 6-41 (p.6-138) (both for framePT and applicationPT processes) and provide these as the actual context parameters.

Figure 6-41: Redefined block type with process set and actual context parameter

Open figure



The difference from the case with application specific instances as part of the framework is not so big: the process set must be foreseen (the context parameter must be defined - and that corresponds to a process set), but the name of it and its position in the application part is not determined. Its position will, however, be constrained by the fact that it shall be visible from the place where the actual context parameter is provided. This means that everything will take place with the virtual block types immediately enclosed by the system type.

Another constraint with this approach is that it is not possible to specify instance sets of the framePT. The reason is that this process type has context parameters. In total this means that the approach with context parameters works only partially: you cannot specify the actual instance sets before the last system subtype.

Process type as context parameter

This approach should only be used in cases where it is important that the framework specific and application specific process types can be defined within the same enclosing block type and where the framework specific types must specify the creation of application specific processes.

A final constraint with this approach is that it is not supported by tools (yet).

Virtual creation procedures

This approach is even more general in the sense that it does not have to be decided if there is one or several process sets in the final system type. In the general types where there is a need to create application specific processes, this is represented by corresponding virtual procedures. In the final system type these are redefined to create processes in the right process sets.

This approach is also constrained by the fact that all processes must be within the same block, so the framework will be defined by a system type with one or more block sets of virtual block types, the extreme case being just one block of one virtual block type.

This approach also has the property that instance sets of the application specific types are first introduced in the final system subtype.

In order to provide an example on this way of making frameworks, we have to change the structure of the Access Control System somewhat.

As the structure of the system is specified above, the number of clusters and thereby of access points is fixed at the time of specification. With clusters defined as above, each cluster will have AccessPoints with the same properties (of the same type).

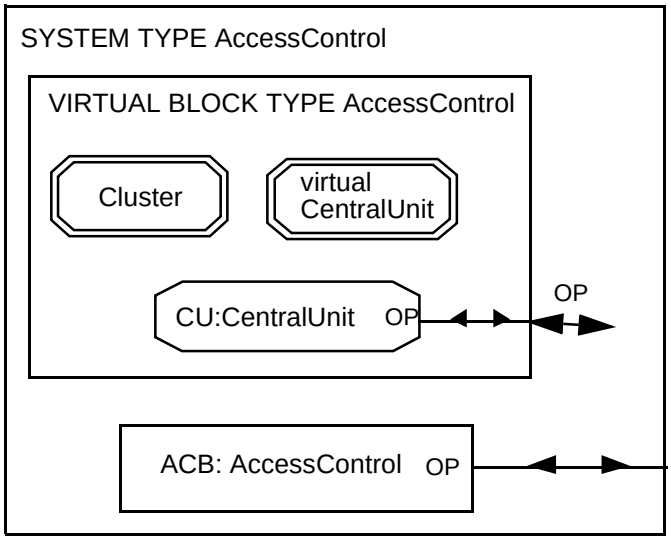
Suppose that it is a requirement that the system shall start by creating processes for each of the actual access points and that changes to the number of access points shall be reflected while the system is running. Still we would like to define the system as a framework in the sense that it will consist of a central unit and a number of clusters. We assume that the division of responsibility between the two are determined, the communication is fixed and that the functionality of both clusters and central unit is specified - the only thing that is not specified is the types and numbers of clusters. The configuration of the system is initiated by a new signal (setUp, with appropriate parameters) coming to the central unit.

For the purpose of this example we assume that it is possible to define both the CentralUnit and Cluster as processes.

The framework system type then is defined as in Figure 6-42 (p.6-139).

Figure 6-42: Framework with virtual creation procedures

Open figure

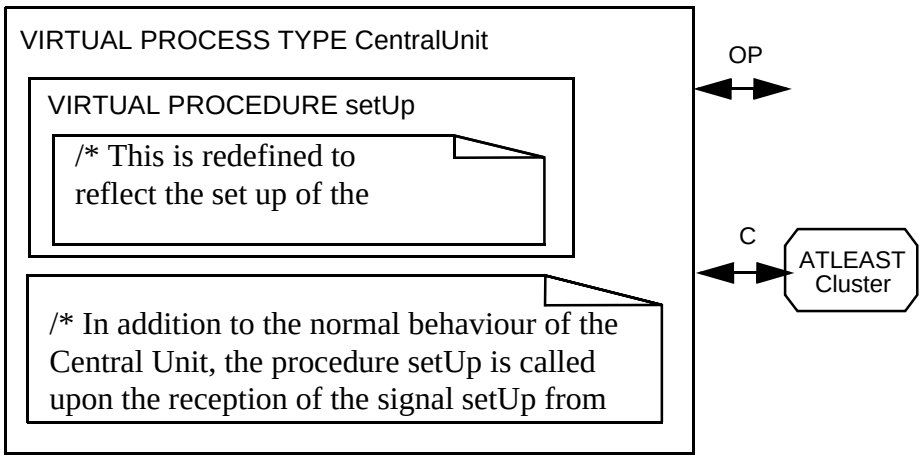


The setUp signal is supposed to come to the CentralUnit and imply the creation of Cluster processes in the right process sets. Depending on the desired number and types of Cluster processes, the signal will carry enough parameters for the CentralUnit to create the right instances.

The Cluster and CentralUnit types can now be defined as before, the only difference being that CentralUnit will be virtual, that it will have a virtual procedure setUp and that it will communicate with possible Cluster processes via a gate that is constrained by Cluster - that is only process sets of Cluster or subtypes of Cluster can be connected to the gate, see Figure 6-43 (p.6-140).

Figure 6-43: Setup as a virtual procedure of CentralUnit

Open figure

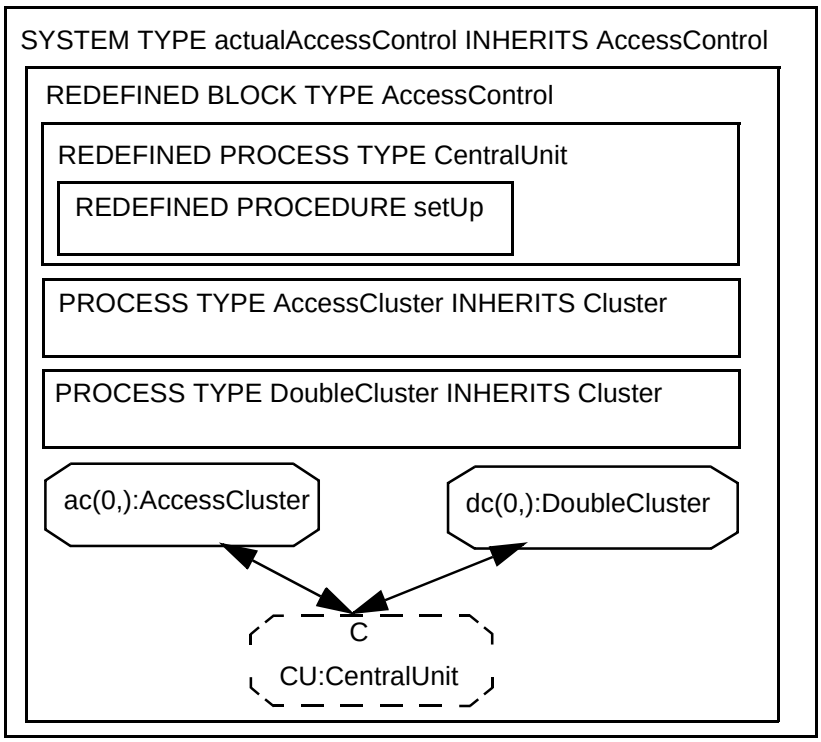


In addition to the normal behaviour and the creation of Cluster processes, the Central-Unit may have behaviour that contributes to the definition of the framework. As an example there may be a limit on the total number of Cluster processes, independent of type of Cluster. The behaviour that ensures this will either be part of the CentralUnit, e.g. some action executed each time setUp is executed, or it may be a constraint on setUp which all redefinitions will inherit.

An actual system consisting of two types of Cluster processes is specified as a subtype of the system type AccessControl, redefining the setUp procedure to cater for this and introduce the two process sets, see Figure 6-44 (p.6-141).

Figure 6-44: Actual Access Control System

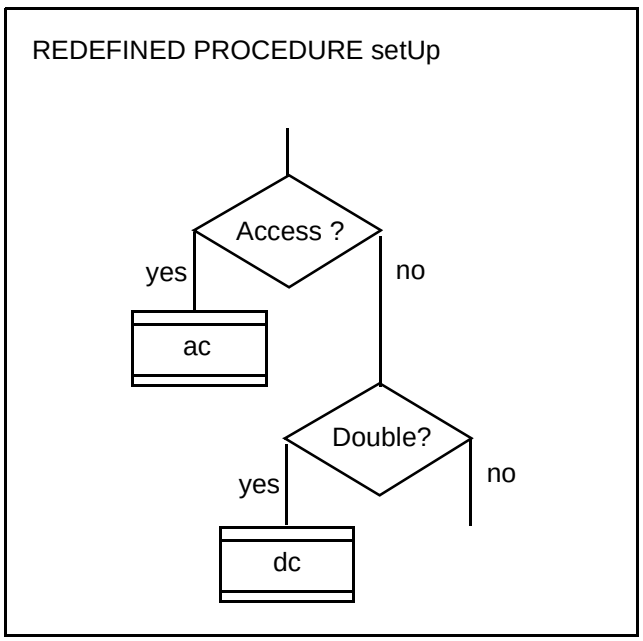
Open figure



The names of the process sets are used in the redefined setUp procedure for the specification of the creation of process instances. A fragment of the redefinition is illustrated in Figure 6-45 "Fragment of redefined setUp" (p.6-141).

Figure 6-45: Fragment of redefined setUp

Open figure

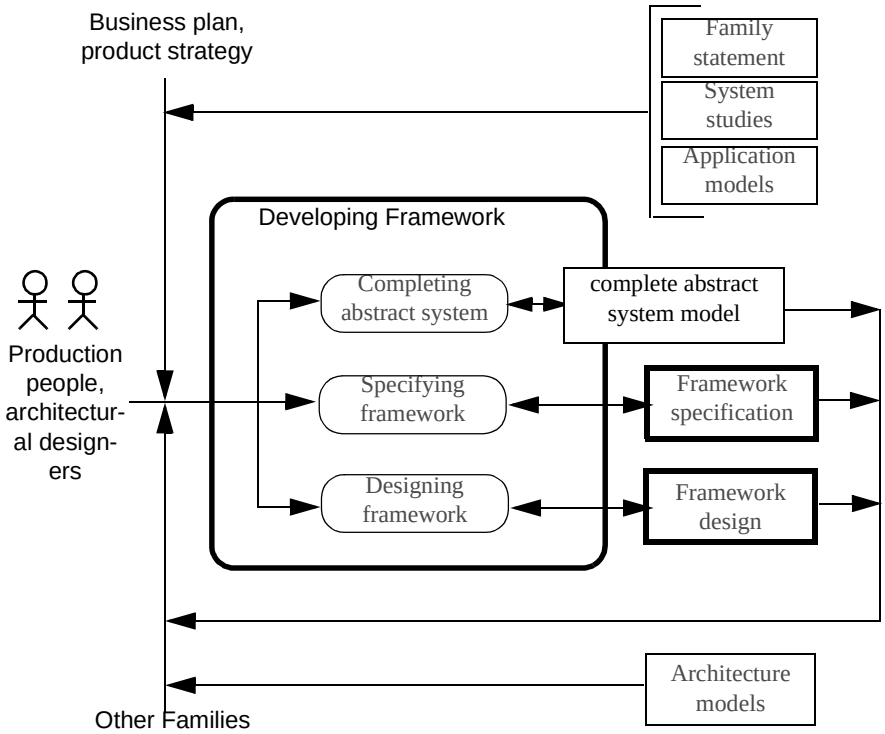


Developing framework

What This activity develops the Framework models (p.6-126) which consists of object models and property models organised as a Framework specification (p.6-128) and a Framework design (p.6-129).

Figure 6-46: Developing Framework

Open figure



Inputs The main sources of input information are:

- Descriptions:
  - Application models describe the application system that the framework shall support.
  - Architecture models describe the underlying platform, which gives structure to the complete abstract system and requirements to the infrastructure.
  - Business plans and product strategy will give high level goals concerning production volumes, service flexibility, evolution and anticipated lifetimes.
  - Other families and especially their frameworks may provide valuable input. It may even happen that the infrastructure and the overall framework structure may be reused from another family even if the application is different.
  - System Studies may provide additional information about possible solutions, future evolution, etc.

  - The family statement should be consulted, although we should expect that the other sources mentioned above contain the same information (but more precise and detailed).

- People:
  - Product management can give additional requirements and guide-lines concerning the needs for variability and flexibility in production and evolution.

Who to involve

This is primarily a task for abstract system designers, but they need to work closely with architectural designers and production people. Frameworks are developed primarily for the purpose of simplifying evolution and production after the initial development. It is therefore essential that people with such responsibility are engaged so that their requirements become clear. It is particularly important to clarify if dynamic changes to a running system shall be supported or not.

It is not always obvious what implementation dependent functionality to include in the framework and what to “hide” in the implementation. This must be agreed with the Architectural designers. It may also happen that framework considerations have an impact on the architecture.

When to do it

Framework design depends on the underlying architecture (implementation) design and is therefore made after the architecture.

This activity is only performed when the framework is undefined or needs to be changed. This occurs during the initial development of a system family and during maintenance when changes in the framework are needed e.g. because of changes to the infrastructure.

During normal application evolution the framework will stay the same. The idea is that new services of an existing system can be designed in terms of the application without considering details of the framework.

Interfaces are sometimes elaborated in this activity. The reason being that the interfaces depend on the architectural design decisions:

- User interfaces depends on the basic technology (e.g. visual interface or mechanical) as well as the support software (e.g. Mac OS or DOS).
- Internal interfaces and external interfaces depend on the interconnecting network and the protocols.

What to do

Making frameworks

The following strategy may be used when making the first framework model for a new system family:

Make Framework

1. Start by Making framework specification (p.6-148). (This may be done before the architecture is designed, during requirements analysis).
2. When the architecture is designed, then the physical structure implied by the implementation platform and additional functionality needed to support the application is considered when Completing the abstract system model (p.6-145).
3. Then analyse the complete abstract system model, trying to isolate the infrastructure. Transform the complete abstract system model into a framework and a framework instantiation. Do this by Making framework design (p.6-149).
4. Harmonising application - framework (p.-124) is then performed (described under Developing Application Models).
5. An so is Harmonising framework specification - design (p.6-131) to make a complete and precise Framework specification for the purpose of later

Completing the abstract system model

This activity assumes that the architecture has been defined. The goal is to restructure the application and to add the necessary infrastructure functionality. This is a typical intermediate step on the way towards a framework. Once the framework has been designed, this model is replaced by a framework instance. However, if it is decided not to develop a framework the complete abstract system must be maintained.

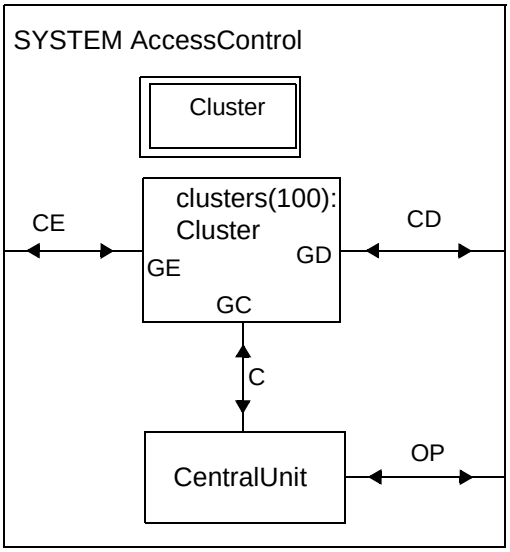
As an example the complete abstract system model for the Access Control system is presented in Figure 6-47 (p.6-146), Figure 6-48 (p.6-146) and Figure 6-48 (p.6-146).

Strategy:

1. *Consider the structure of computational nodes in the architecture, and make a corresponding abstract system structure.*
2. *Localise all the application components (instances of structure types and behaviour types (objects)) in this structure.*
3. *Add infrastructure components needed to support communication within the application.*

Figure 6-47: Redesigned Access Control system

Open figure



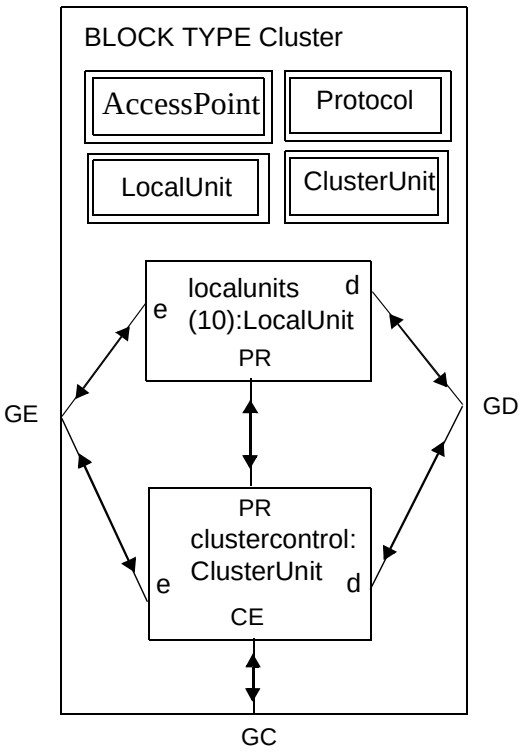
In the access control system the channels between the AccessPoints and the CentralUnit are candidates for distribution. We therefore decide to let these channels be the ones that cover distances.

There will be at least one of 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.

Figure 6-48: Cluster with LocalUnits and ClusterUnits

pen figure

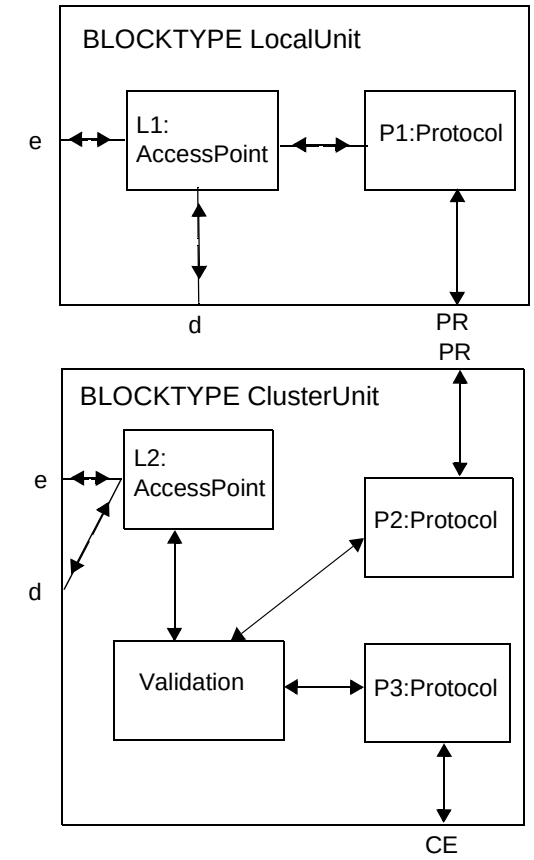


In this solution the validation database will be distributed. There will be a copy of the central Validation process (and its database) in each cluster. This means that the CentralUnit must handle updates in a distributed database. This introduces a new problem to solve in the functional design, but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

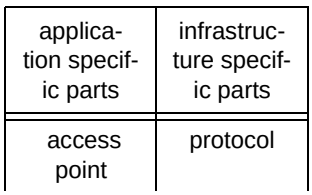
application specific parts	infrastructure specific parts
access point validation	protocol cluster unit local unit

Figure 6-49: 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), but the signals to and from the AccessPoint blocks will be the same. In order to achieve distribution transparency the way of access to Validation should be hidden from the AccessPoints. Therefore a routing function must be added to the structure.



4. *Add additional infrastructure components needed to support services stated in the framework specification (Use the same approach as for application design synthesis here):*
   - *support for distribution transparency, object relocation , dynamic configuration and change;*
   - *support for error handling.*
5. *Check that the complete application functionality can be provided in the new structure.*
6. Define the infrastructure component types that have been identified.

Define the method for instantiating frameworks with an application.

This is a textual description telling how to instantiate the framework both statically and dynamically.

Evolving frameworks

Use the following strategy when evolving a framework:

Evolve Framework

1. First add new properties and change existing properties as required by Evolving framework specification (p.6-149).
2. Then analyse the impact on the framework design and perform the necessary changes.
3. If the changes have any consequences on the existing applications then ensure that these applications are updated by Harmonising framework - application (p.6-131).

Specifying frameworks

Inputs

Inputs to this activity are the same as for Developing framework models as a whole, see Figure 6-32 (p.6-127). However, the initial specification is made before there any architecture.

Making framework specification

The initial framework specification is normally developed before the architecture during requirements specification. What can be said at that stage is often limited, and therefore the initial framework specification is usually quite open. Typical items are:

- Requirements to distribution imposed by the physical environment, e.g. that services must be provided at different locations. In the Access Control example, services must be provided in the vicinity of doors which are physically scattered around in buildings.
- Requirements to distribution and communication infrastructure. It may for instance be a requirement that distribution shall be supported using CORBA and that TCP/IP shall be the protocols.
- Requirements to error handling services. Should there be alarm and diagnostic functions? Should it be possible to block units for repair?
- Requirements to dynamic configuration services and dynamic change. Should it be possible to change or relocate application objects? Should it be possible to extend the system during operation?

Guidelines:

Specify framework context

- *Consider the physical distribution of the environment. If the distribution is known, make a context model where the environment is structured to show physical distribution of the environment and the interfaces.*

- *Consider framework specific services. Specify each service using text, role structures and MSC.*

Specify framework content

- *Consider the physical distribution of the content. If it is known, make a corresponding content structure.*
- *Consider requirements to the infrastructure and add corresponding objects to the framework object model.*
- *Specify the behaviour of infrastructure services and interfaces.*
- *describe in text requirements to configuration, evolution and dynamic changes.*

Evolving framework specification

Framework evolution is not due to application services, but to changes in the architecture or the infrastructure services. It may also be due to a better understanding of how the framework should be. Finally the specification is influenced by the framework design since it should always be harmonised with that.

Infrastructure services are evolved in the same general way as application services:

1. *Specify the new services using property models.*
2. *Analyse the impact on existing services and object models.*
3. *Specify new objects required in (the various parts of) the environment (if any).*
4. *Change existing services and object models as required.*

Changes in the architecture are more likely structural changes that will affect the object model.

Designing framework structure

Making framework design

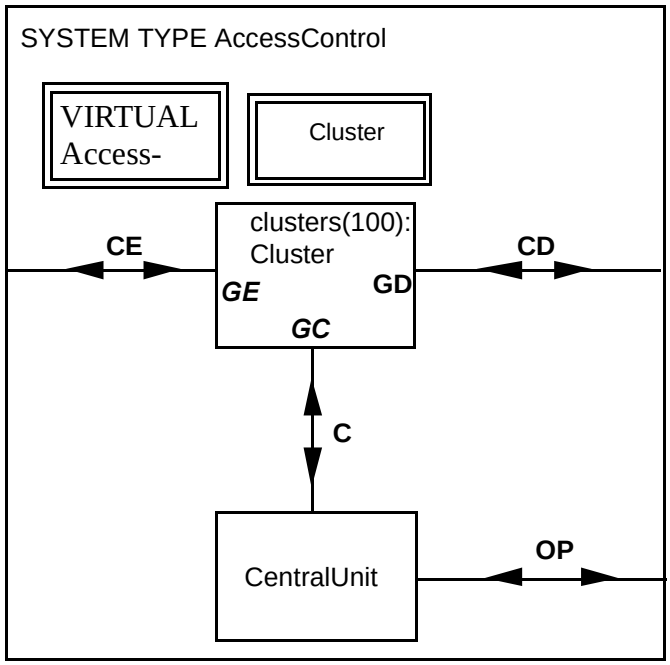
It is an advantage to make a complete abstract system model before starting to make a framework, because then we have something tangible to start from. Our goal is to transform the complete model into:

1. a framework design that is easy to use with different applications;
2. an instantiation of the framework that replaces the original complete abstract system model.

As an example we consider the Access Control system again. The system description of Figure 6-47 (p.6-146) 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 6-50 (p.6-150).

Figure 6-50: Access Control System type as a framework

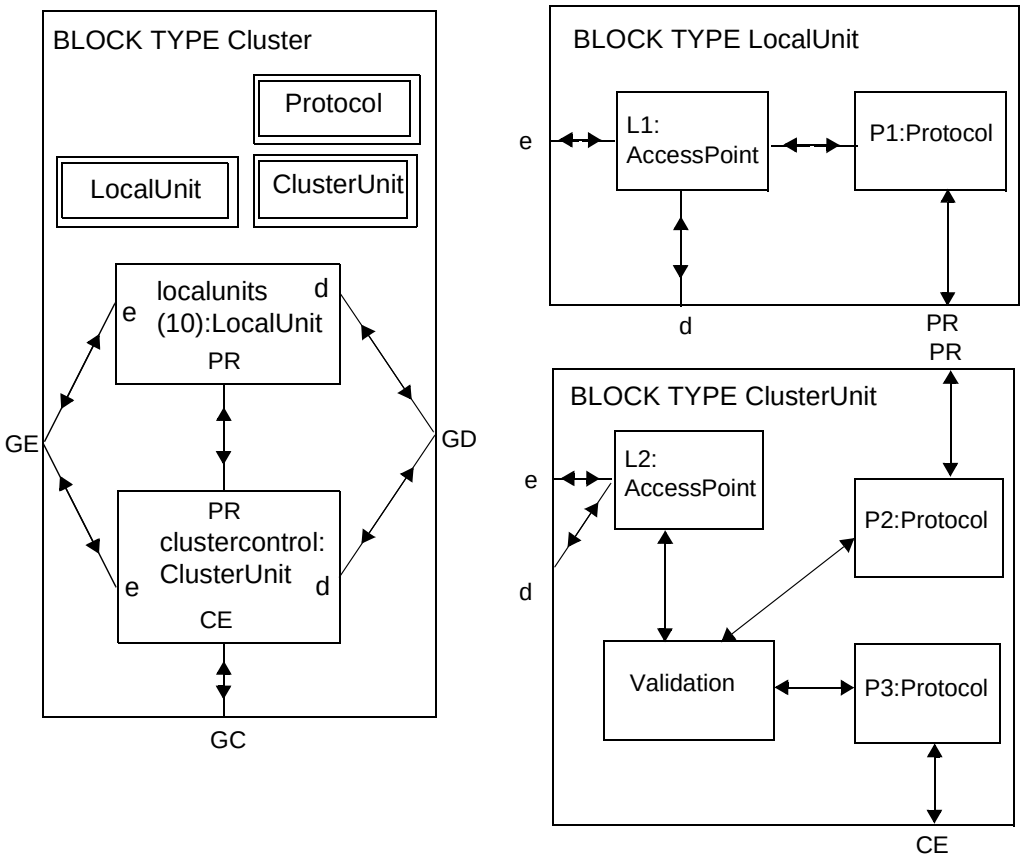
Open figure



The Cluster block is almost as before: it uses the virtual block type Access Point (but it does not contain its definition), and it embodies the infrastructure parts needed for distribution (ClusterUnit, LocalUnit and Protocol), see Figure 6-50 (p.6-150).

Figure 6-51: Block type Cluster as part of framework for Access Control Systems

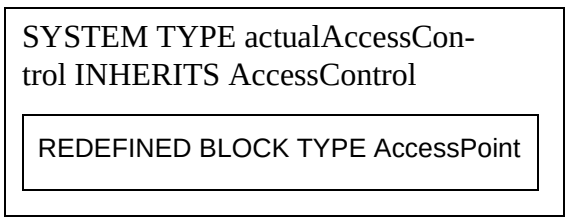
Open figure



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 6-52 (p.6-151). 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 6-52: An actual system based upon a framework

Open figure



In the complete abstract system we need not worry too much about variability. The goal is to understand the overall structure and to identify the infrastructure components. In the Framework design, however, we must seriously consider the requirements to dynamic configuration and change. Is each system Instance to be statically configured for its lifetime, or shall it be possible to make dynamic changes? If the answer is yes, then we must design the framework so it contains suitable dynamic change units.

Is the configuration support we get by using system types with context parameters in SDL sufficient or do we need more flexibility? If the answer to the latter is yes, then we must turn to what can be supported by SDL behaviour, i.e. dynamic process creation and value assignments. If we need even more flexibility, for instance to create blocks dynamically, then this must be handled outside SDL. In such cases, it will not be very useful to cover the entire system formally as a single SDL system type - or block type (except for simulation purposes). In stead we should develop framework components that can be used to compose and configurate systems more freely, possibly using means that we provide outside SDL. In the Access Control example, it will be difficult to achieve full flexibility using the system type defined in Figure 6-50 (p.6-150) as each LocalUnit and ClusterUnit may need to be configurated differently. What is possible, however, is to use these components to compose many different SDL systems. Therefore framework components will sometimes be more useful than complete framework systems.

There are some trends in system development that should be considered:

- Service flexibility. The ability to provide new services fast and safely, even in existing systems, is increasingly important.
- Distribution flexibility and transparency. It is increasingly common to allow objects to be distributed freely and even to be relocated in the system.
- Standard platforms. There is a strong drive towards using standard computation platforms and communication infrastructure. New standards for distribution support, such as CORBA are emerging.

These trends point towards:

1. that much infrastructure functionality will be standardised;
2. that the system structures will change dynamically so that components are more stable than systems.

Guidelines:

- *Consider the components of the application part alone. Make a “pure” application type for each kind of application components.*
- *Consider the infrastructure part alone. Make a “pure” infrastructure type for each kind of infrastructure component.*
- *Consider the overall structure of “mixed” application and infrastructure components. Find framework components where the application part will vary while the infrastructure part is stable. Define corresponding types where the application is redefinable (using virtuals) and the infrastructure (sufficiently) configurable.*
- *Note that the component types found above should not be too large, they should be:*

- *running entirely within one computation node;*
- *be a configuration unit;*
- *be a change unit;*

• *If feasible and useful, define the entire system framework as a structure type (a block type or system type in SDL).*

Designing framework behaviour

This is carried out in the same general way as Designing application behaviour (p.-142).

Architecture

Content and scope

An architecture is an abstraction of a concrete system representing:

- the overall structure of hardware identifying at least all physical nodes and interconnections needed to implement an abstract system;
- the overall structure of software identifying at least all software nodes, software communications and relations needed to implement an abstract system (in terms of processes, procedures and data).

In this chapter we shall describe:

- Architecture reference model (p.6-154): the system model assumed in the architecture.
- Architecture models (p.6-164): how to describe architectures.
- Developing architecture (p.6-173): how to develop architecture models.

Architecture models are important because:

- they help to design the physical implementation in the first place;
- they serve to document how the system is implemented.

Documentation

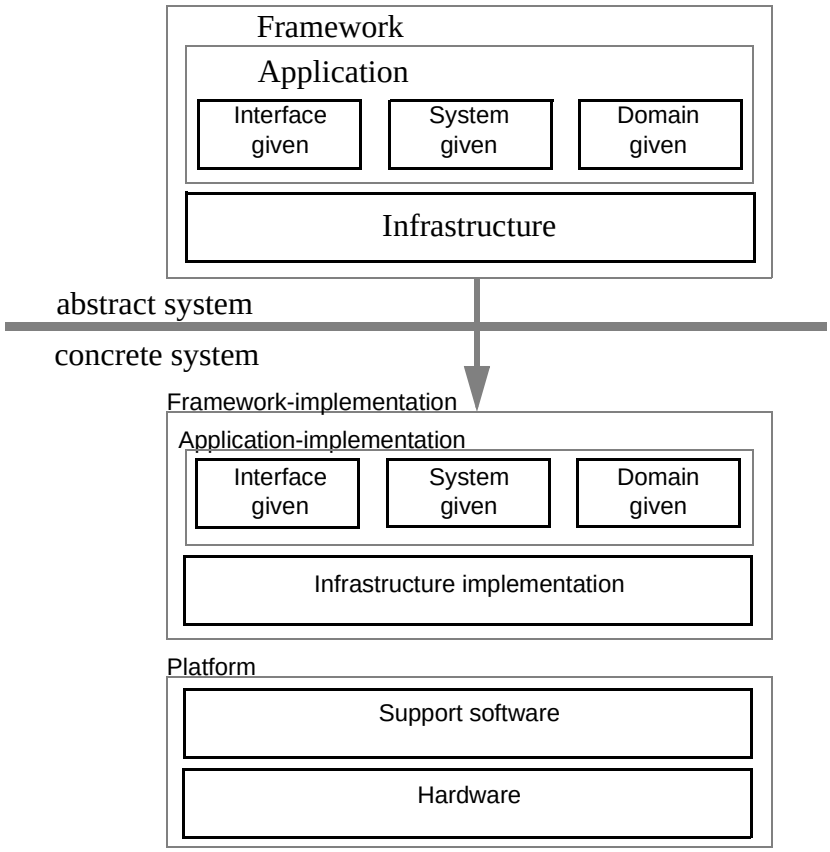
Together the application, the framework and the architecture complement each other to form a complete documentation.

Architecture reference model

In the system reference models we distinguish between abstract systems and concrete systems as illustrated in Figure 6-53 (p.6-155).

Figure 6-53: Reference models for abstract and concrete system

Open figure



Concrete system

The architecture is a simplified model of the concrete system. Concrete systems are composed from real hardware and executable software that provide services to real users.

While abstract systems, described in application and framework models, are composed from abstract components, concrete systems are composed from physical components and software. It is the concrete system that really matters to the users, but it is the abstract systems that enable us to understand what it does and to ensure that its functionality has the desired quality.

As illustrated in Figure 6-53 (p.6-155), concrete systems consist of:

- Framework implementations. Here we find implementations of the abstract systems in hardware and software:
  - the application implementations.
  - the infrastructure implementation.
- The Platform, which consists of:

- the support software, which normally is a layered structure containing operating systems, middelware for distribution support, runtime systems for the languages used (SDL), DBMS and interface support;
- the platform hardware, which typically is an network of computers.

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.

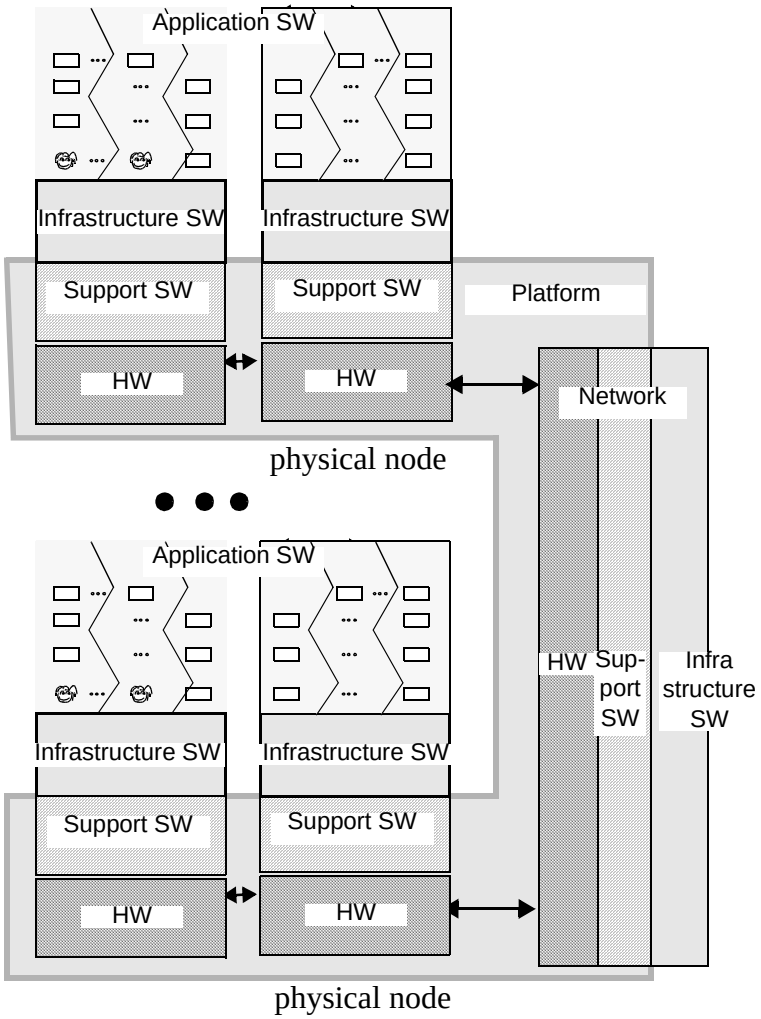
What is an architecture?

What

The architecture is an abstraction of the concrete system that emphasises high level hardware structure and software structure. It will not go into the implementation details but focus on the implementation principles and how functional and non-functional properties are satisfied. It will often be distributed and have additional support for internal communication, as illustrated in Figure 6-54 (p.6-156).

Figure 6-54: Architecture reference model

Open figure



Objectives

By identifying key architectural elements, the reference model helps to structure architecture models and to guide their development. Although the model in Figure 6-54 (p.6-156) is quite coarsely grained, it identifies elements that come from different domains where the driving forces behind change and evolution are quite different. Application evolution, for instance, may be due to new service requirements (market pull), while the reason for platform changes may be performance and price (technology push). Keeping such parts separate and as independent as possible, we believe, is a key to flexibility and profitability.

The architecture reference model contains the following main parts:

1. Hardware. It contains the underlying structure of physical nodes and interconnections that may be classifies as:
   - General platform hardware. This is hardware that is not application framework specific. It will typically consist of general purpose computers in a network.
   - Application framework specific hardware. This is hardware needed for the particular application or infrastructure that the system implements. Here we find peripheral equipment, such as the panels in the AccessControl system, and other special purpose hardware.
2. Support software, which contains such elements as:
   - the operating system;
   - middleware (e.g. CORBA) that supports distribution;
   - support for application framework programming, for instance: user interface tools (GUI), database management tools (DBMS), language runtime support (for SDL);
   - support for application framework input/output (I/O) (that is for interaction with hardware).
3. Infrastructure software, which implements the necessary interface between the application and the support software.
4. Application framework software. This is the software part of the framework implementation. It contains implementations of the:
   - Framework application (p.-154).
   - Framework infrastructure (p.-153).

   The application framework software can be further classified as:
   - automatically generated code;
   - manually generated code (handmade code).
5. Foreign code, code that is imported from other sources, legacy code, etc.

In real-time systems a central software design issue is how to handle concurrency, timing and communication. To this end it is important that the architecture manages to model the organisation of concurrent software processes (also called tasks in some operating systems), how they communicate and how they are scheduled.

Other important issues are:

- the practical organisation in software modules that can be developed by separate teams;
- how to generate framework implementations;
- how to configurate and build concrete (executable) systems.

Generality

In the family area, architectures are supposed to be quite general and contain the variability needed in different system instances. As a minimum they must support the variability needed in applications.

It is generally a good idea to use the same basic principle we used to make frameworks from applications: to define a layered framework structure.

Applied to the hardware part, the architecture may be seen as a hardware framework made up by some (stable) general hardware and some (redefinable) application framework hardware.

In the software part, we find the basic operating system at the bottom, with the other parts as successive applications on top of that. We may organise this as several software frameworks on top of each other. Ideally the result is a high level software framework where all we need to do to generate a particular application is to supply the application specific code.

Differences between abstract and concrete system

A good designer must be well aware of the differences between the concrete and the abstract world. There are two main categories of such differences:

1. Fundamental differences (p.6-158) in the nature of components. Physical components are rather imperfect compared to the more ideal components of abstract systems. They develop errors over time, they are subject to noise, and they need time to perform their processing tasks.
2. Accidental differences (p.6-160) in the functioning of components. In both worlds there are concepts for concurrency, communication, sequential behaviour and data, but they are not necessarily the same.

Fundamental differences

The fundamental differences are related to: Processing time (p.6-158), Errors and noise (p.6-159), Physical distribution (p.6-160) and Finite resources (p.6-160).

Processing time

An abstract system is not limited by processing resources. Consequently the balance between the traffic load offered to the system and its processing capacity need not be considered. One simply assumes that the system is fast enough to process the load it is offered.

The real world is vastly different on this point. Each signal transfer, and each transition of a process, will take some time and require some processing resources. Due to these differences, the focal point of implementation design is quite different from that of the

functional specification. The challenge is to find implementations for the abstract (SDL) concepts that are sufficiently fast to meet the traffic load and response time requirements without destroying the validity of the abstract system models.

One major issue is to balance the processing capacity of the implementation against the offered traffic load.

In the Access Control system, for instance, the average peak load is 600 validations a minute. This means that a central computer must process each validation in less than one tenth of a second. How much less will depend on the other tasks the computer has to do and the margin one wants against overloading the system.

A related issue is to balance the processing capacity against the requirements to response times. Again the abstract (SDL) system has no problems, but the implementation may be highly pressed to meet response time requirements. One must be able to perform time critical processing, e.g. fetching input samples, process the sampled information, and respond in a feedback loop, all within a maximum time frame. Such requirements may increase the demands on processing speed beyond the speed required to handle the traffic load.

The hardware software interfaces need special consideration. It is not unusual that the larger part of a computers capacity is spent doing input-output. Much can therefore be achieved by carefully designing the input-output interface.

A special class of time constraints originates from channels assuming time dependent synchronization. This means that the receiver has to be fast enough to catch all relevant signal information at the speed it is passed over the channel.

Since SDL descriptions clearly specify the external and internal interactions needed to perform given functions, they provide an excellent basis for estimating the processing capacity needed to meet load and timing requirements.

Errors and noise

Abstract systems may suffer from specification errors, but the abstract world does not suffer from physical errors. It is simply assumed that processes and channels always operate according to their specification. It is not assumed that processes will stop from time to time, or that channels will distort the content of signals. But in the real world such things happen. From time to time errors will manifest themselves as faults in the operation of channels and processes.

In addition to the logical errors introduced in the implementation, we will have to cope with physical errors. Hardware errors, physical damage, and noise are caused by physical phenomena entirely outside the realm of abstract systems.

The effect of errors and noise will often need to be handled explicitly in the abstract system models, however. One must consider what may happen, how it can be detected and how the damages may be limited. If one active object fails, for instance, how should the environment react? One must consider what an object should do if it never gets a response to a request, or if it gets an erroneous response. What should be the reaction to a channel going down? What if an object starts to produce crazy signals? What if a signal is sent to a non-existing receiver?

To some extent the answers depend on the physical distribution of abstract objects in the concrete system, and the physical distances that abstract channels must cover.

Physical distribution

Physically separate processes and channels may fail independently of each other. Channels covering long physical distances are subject to more noise and errors than channels implemented in software within one computer.

An SDL description does not tell anything about the physical distance covered by a channel. In reality, however, there may (or may not) be large physical distances. This means that transmission equipment and protocols are needed to implement the channel reliably. Thus physical distance may introduce new functions needed to support the implementation of channels.

In the AccessControl system we can expect AccessPoints to be distributed physically and to be far away from the central unit. Thus there will be a need for communication protocols on the channels between them.

A positive effect of physical separation is that errors are isolated. Errors in one unit need not affect the other units in the system, provided that erroneous information is not propagated into them. Thus, physical separation may improve the error handling. But there is no free lunch. Errors need to be detected and isolated to allow the operational parts to continue operation with the error present. Proper handling of this aspect can be quite complex, and will normally require additional functionality in the abstract models. This is one of the issues that we seek to isolate in the infrastructure.

Finite resources

All resources in a real system are finite. There may be a maximum number of processes the operating system can handle, or a maximum number of buffers for sending messages. The word length is restricted, and the memory space too. Even primitive data like integers are finite.

SDL, on the other hand, has an unbounded queue in the input port of each process, and allows infinite data to be specified. Hence the designer must find ways to implement potentially infinite SDL systems using finite resources. One way is to restrict the use of SDL such that all values are certain to be bounded. Another is to deal with resource limitations in the implementation, preferably in a manner transparent to the SDL level. In cases where transparency cannot be achieved, one must either accept deviation from the SDL semantics, or explicitly handle the limitations in the SDL system.

Accidental differences

The accidental differences has to do with: Concurrency (p.6-160), Communication (p.6-161), Synchronization (p.6-162) and Data (p.6-163).

Concurrency

The model of concurrency used in SDL assumes that processes behave independently and asynchronously. There is no relative ordering of operations in different processes except the ordering implied by the sending and reception of signals.

This permits SDL processes to be implemented either truly in parallel on separate hardware units, or in quasi-parallel on shared hardware.

Physical objects in the real world behave truly in parallel. This means that operations within different objects proceed in parallel to each other at the speed of the performing hardware.

A “natural” implementation is therefore to map each SDL process to a separate physical object. This is not always cost-effective. An alternative approach is to implement many processes in software sharing the same computer hardware. The implications of this are twofold:

1. The processes and channels will not operate truly in parallel, but in quasi-parallel, meaning that they will operate one at the time, according to some scheduling strategy.
2. Additional support will be needed to perform scheduling and multiplexing on top of the sequential machine.

Normally, scheduling and multiplexing are handled by an operating system. The operating system can be seen as a layer that implements a quasi-parallel virtual machine on top of the physical machine.

Communication

Very basically there are two different classes of information one needs to communicate:

1. sequences of symbols, or values, in a given order;
2. symbols, or values, continuously for the time they are valid.

In the first case, the sequential ordering is important. In the second case, the sequence does not matter, only the current value at each instant in time.

The two communication forms are dual in the sense that one form may be used to implement the other. Consider the need to communicate a continuous value: the most direct implementation is to use a communication medium that will transmit the value continuously, such as a shared variable in software, or an electrical connection in hardware. But one may alternatively use a sequential medium, such as a message queue, to transmit a sequence of symbols representing the sequence of changes (events) in the continuous value. This will introduce overhead to reconstruct the continuous value.

Both forms may be used in abstract as well as in concrete systems, but it is not always the case that the same form is used. Therefore it may be necessary to make the necessary adaptations in the implementation.

Input from a keypad may serve as an example. The output signal from each button is basically a continuous “1” when pushed, and a continuous “0” when not pushed. But the system needs to know the sequence of key strokes, and not the instant values. Thus the value changes (events) need to be detected and converted to symbols representing complete key strokes. Event detection like this is often needed at the interfaces of a real-time system. It may either be performed in software or in hardware.

Visual signals on a display screen are another example. The user wants information presented as continuous values, and not as messages flickering across the screen. Hence the event oriented SDL signal has to be converted to a continuous value on the screen.

Channels crossing the hardware-software boundary need special attention. An atomic channel, represented by a line in SDL/GR, may turn out to be a mixture of physical lines, electronic equipment and software in the real system. The communication and synchronization primitives used in hardware will often differ from those used in software.

To sum up: we cannot expect to find SDL-type signals at all interfaces, and must therefore be prepared to adapt and convert. Conversion from one form to another will be necessary. This is often a time critical task needing careful optimization.

Synchronization

The act of aligning the operations of different concurrent processes in relation to each other is generally called synchronization. Synchronization is necessary not only to achieve correctness in communication, but also to control the access to shared resources in the physical system.

In SDL, synchronization is achieved by means of the signal queues of processes and channels.

Consider two SDL processes that communicate. The sending process may send a signal at any time because it will be buffered in the input port of the receiving process. The receiving process may then consume the signal at a later time.

This is a buffered communication scheme in which the sender may produce infinitely many signals without waiting for the receiver to consume them. It is often referred to as *asynchronous* communication.

Asynchronous communication may be contrasted with so-called *synchronous* communication, in which the sending operation and the consuming operation occur at the same time. In this case there is no buffer between the processes.

Synchronisation may be further classified into *time dependent* synchronization, in which the operations are not explicitly synchronized, and *time independent* synchronization, which depends on an explicit synchronization of operations (some kind of semaphore). In time dependent synchronisation the correctness of an interaction depends on the relative timing of operations. This is a frequent source of so-called hard real-time requirements. It is quite common in communication channels (e.g. Asynchronous Transfer Mode, ATM).

One will often find mechanisms that differ from the SDL mechanisms at the physical interfaces to the system. It is quite typical to find time dependent synchronization on physical channels. This implies that time critical event monitoring and event generation will be necessary.

A designer will be faced on one hand by the synchronization primitives available in the real system, and on the other hand by the synchronization implied by the SDL specification. Additional functionality will often be needed to glue the various forms together.

Data

SDL data is based on the notion of abstract data types where operations may be defined by means of axioms. An implementation will normally need concrete data types where the operations are defined operationally. Therefore, the designer may need to transform the abstract data types of SDL into more concrete data types suitable for implementation.

Architecture models

Objectives

Architecture models are intended to answer *how* a system is (going to be/has been) realised. Their focus is on the concrete system construction in terms of hardware and software components, and how it implements the abstract systems defined in the application framework. While the application framework has focus on functional properties and behaviour, the architecture has focus on non-functional properties and physical structures. The purposes of architecture models are:

- to support the high level implementation design activity through models that facilitate communication and analysis;
- to serve as precise high level documentation of the implementation (using a unified notation);
- to document how the application framework is implemented;
- to specify the implementation (non-functional) properties;
- to facilitate the generation of implementations and the production of system instances;
- to serve as an entry to complete system documentation. This is possible since all aspects of a concrete system come together in the Architecture.

The concrete system modelled by the architecture shall behave as defined in the framework and application models and satisfy the non-functional properties.

What

Architecture models describe the:

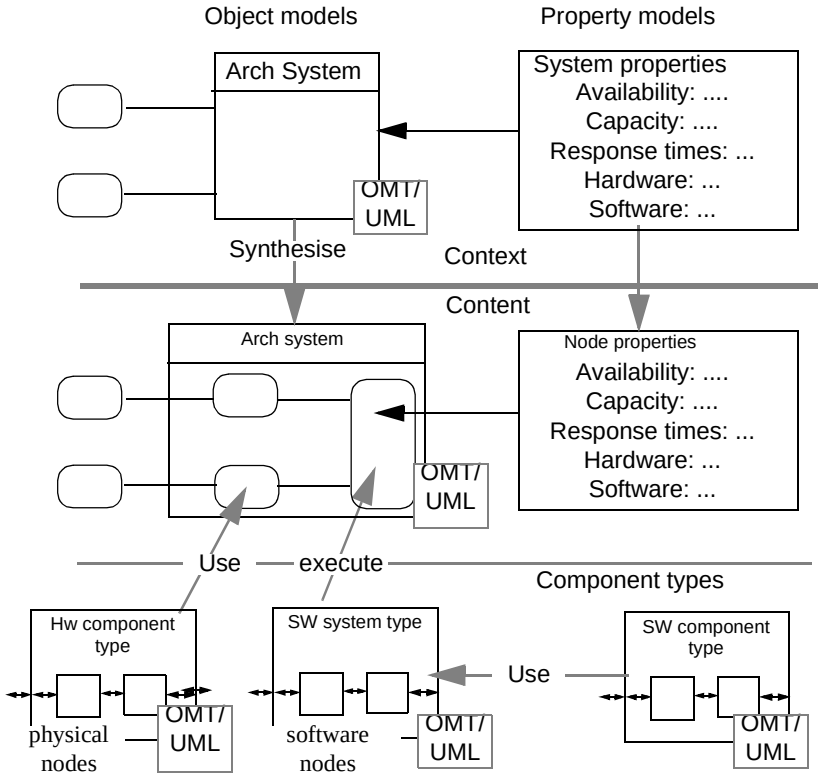
- overall architecture of physical nodes and software nodes that actually perform system behaviour and satisfies user needs;
- relationships to the application framework components (and thus application components) that each software node and physical node implements, see Figure 6-33 (p.6-130);
- non-functional properties.

In the architecture object models, the hardware and software architectures will be defined to a level of detail from which implementation is well defined. They should be organised into a generic platform specific part and a framework specific part which is adaptable to different applications.

The architecture contains variability that needs to be bound in system instances. In fact, configuration of system instances may take place at the architecture level, if suitable tool support is available.

Figure 6-55: Architecture models

Open figure



In the architecture property models, the non-functional properties are expressed.

The architecture models are organised into a specification part, and a design part, as illustrated in Figure 6-32 (p.6-127).

Qualities

The following qualities are sought:

- Functional equivalence between the abstract and concrete system.
- A mapping that is simple and easy to understand.
- Satisfaction of non-functional requirements.
- Automation of the realization process.

Documen-tation

Together the application, the framework and the architecture complement each other to form a complete documentation. As long as the framework and the architecture is stable, only the application need to be supplied for each new system.

Note that the architecture serves as our (only) entry point to a complete description of a given concrete system. By following the relationships “upwards” to the framework we find models of the abstract functionality it provides, and by following the relations “down” towards the Implementation we find all the implementation detail.

Derivation of concrete systems

It is recommended to define a Method for framework code generation (p.-96) (automatic and manual), and a Method for system instantiation (p.-97) which define the procedures and tools for configuration and building of system instances. (The configuration and building of system instances shall be as easy as possible.)

Architecture model content

The architecture property models describe non-functional properties and the architecture object models describe hardware structure, software structure and relationships.

Architecture property models (non-functional properties)

Non-functional properties are more physical in nature than the functional properties. They express features of a concrete system that are not modelled by a corresponding abstract system, typically features related to performance, error handling, power consumption and physical construction. See Architecture non-functional properties (p.6-168) for details.

Architecture object models

Object models should be layered according to the Architecture reference model (p.6-154), see Figure 6-54 (p.6-156) and consist of Hardware models (p.6-166), Software models (p.6-166) and Implementation relations (p.6-167).

Hardware models

Hardware models describe the hardware on a high level. Here we find a description of the overall hardware structure as well as the types of hardware components used to compose the structure. Hardware models shall identify at least all physical nodes needed to implement the abstract system. Whenever possible and practical one should make a hardware framework that separates between:

- general platform hardware;
- application framework specific hardware.

Software models

These are models of the software. Here we describe the overall structure in terms of processes, procedures and data as well as the component types used in the structure. Whenever possible and practical one should make software frameworks that separates between:

1. Support software:
   - the operating system;
   - middleware (e.g. CORBA) that supports distribution;
   - support for application framework programming, for instance: user interface tools (GUI), database management tools (DBMS) , language runtime support (for SDL);

- support for application framework input/output (I/O) (that is for interaction with hardware).

2. Application framework software. It contains the application and the infrastructure (software) implementations, and may be classified as: automatic and hand written.
3. Foreign code, code that is imported from other sources, legacy code, etc.

Software models will often be organised with one (framework) type model for each type of computer (address space).

Implementation relations

Implementation relations are used to describe which parts of the application frameworks that are *implemented-by* the various physical nodes and software nodes, and where detailed hardware descriptions and source code can be found.

Architecture model languages and notations

Object models will be used to express the hardware and the software models. In the old SISU methodology two special diagram types were used for this. As it seems, the forthcoming UML will contain notations intended for similar purposes. If they prove adequate, they will be adopted by TIMe.

For property models a combination of text, mathematics and figures will be used.

Architecture specification

Objectives While the application specifications describe abstract system properties related to the system behaviour (functional properties) the architecture specification describe concrete system properties (non-functional properties) related to the physical construction. Application specifications serve several purposes:

1. Before the architecture is designed it serves to specify required properties that the concrete system shall satisfy. Such requirements are sometimes called design constraints, because they constrain the possible design space.
2. After the architecture has been designed, to document its provided properties in a way suitable for assessment, retrieval and (re)use.

What Specifications apply to the physical architecture as a whole as well as to each type of component in it. Hardware components and software components are normally characterised by different properties. (Power consumption, for instance, is a hardware property, while code size is a software property.) As for other specifications, the main thing is to describe properties that are important for the external use of the system. Consequently the context model is most important, but internal aspects may also be relevant.

Architecture specification object model

Specification object models will not always be needed, and may be omitted when they follow from the non-functional requirements in an obvious way. However, object models are good at describing objects and relationships, and should be included whenever that kind of information is important.

1. Hardware specification. This may possibly be split into a specification of the general hardware and specification of the application framework hardware:

- Context models that describe the physical structure of the environment:
  - (distributed) hardware units;
  - connections and interfaces in hardware.
- Content models that describe:
  - (distributed) hardware units directly linked to the environment, and therefore visible by the environment;
  - other hardware units that needs to be mentioned in the specification;
  - connections and interfaces in hardware (note that the physical distribution of a system may be important, and the interfaces between these distributed units may be of interest).

2. Software specifications; that may possibly be split into a layered structure according to the Architecture reference model (p.6-154).

- Context models that describe:
  - software in the environment and external software interfaces.
- Content models that describe :
  - software units directly linked to the environment, and therefore visible to the environment,
  - other software units that need to be mentioned in the specification. This may typically be support software of various kinds and existing application software that shall be used.
  - internal software interfaces if appropriate.

For hardware as well as software there may be general policies and engineering practices. These must be mentioned in the property models.

Note that variability is important. Try to express variability both in the number of components and in their types. One should for instance specify what range of platforms that shall be supported, and the maximum number of computers the system may contain.

Architecture non-functional properties

User and customer related properties (p.6-169) are important for the user (or the customer/owner) while Company internal properties (p.6-170) are more important for the company itself (see also Dimensions of the property concept):

Having specified the services of the Access Control system, the Sesam-Sesam people began wondering if that was all a customer would be interested in? Wouldn't the number of users that could be handled at the same time matter? What about the response times? And not the least - what if anything go amiss? They also started to consider their own problems. How to produce the system? What about maintenance and service? Did they have any components ready made? After some thinking they came up with an initial list of non-functional requirements:

1. Physical distribution. The system shall be able to serve a building complex where the distance between doors is up to 1km measured as the cables lie.
2. Processing capacity. The system shall be able to serve 6 users a minute at each Access Point, and up to a total continuous peak load of 600 users a minute for the total system. Higher input rates shall not lead to loss or corruption of data, only to longer delays.
3. Error handling. A single error shall not affect the (normal) operation of more than 10 Access Points.
4. Security. The authentication and authorization information shall be secured against unintended access.
5. Hardware. Standard plastic cards shall be the means for identification. Standard card readers shall be used.

Some properties applies generally to the architecture while other properties can be linked with specific parts of the object models.

User and customer related properties

These are:

1. General properties.
2. Properties related to the hardware context:

- The physical environment:
  - temperature;
  - humidity;
  - vibration;
  - physical distribution and distances, etc.;
- interfaces;
- performance:
  - response times;
  - load capacity;
  - overload handling;
- reliability and error handling;

- security.

3. Properties related to the hardware content:

- use of platform hardware;
- use of ready-made components, etc.

4. Properties related to the software context:

- software environment;
- software interfaces.

5. Properties related to the software content:

- support software;
- application framework software.

Company internal properties

These are such as:

- production aspects;
- service and maintenance aspects;
- hardware requirements;
- software requirements.

Non-functional properties are expressed in a mixture of text, figures and mathematics. The precise form varies between different kinds of properties. Non-functional properties are to be associated with the object models.

Architecture design

Objectives To describe the design parts of architecture models that:

- satisfy the specification;
- show every physical node and software node where application and framework objects are implemented;
- define hardware and software architectures to a level of detail from which implementation is well defined and sufficient as high level documentation.

What Architecture design determines critical architectural issues such as physical distribution, communication schemes, support software and physical interfaces. Some of these may subsequently be reflected in the Framework model in order to describe the complete system behaviour.

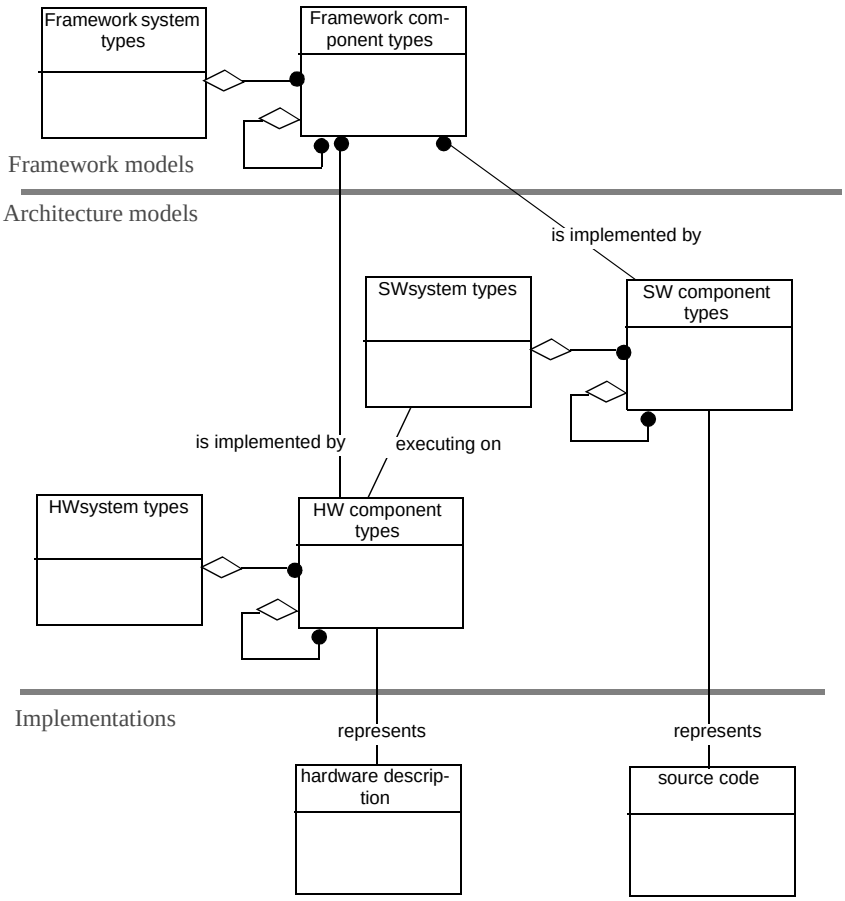
The design will cover the overall structure as well as each type of component in it.

Architecture relationships

The external relationships are presented in Figure 6-33 (p.6-130).

Figure 6-56: Architecture relationships

Open figure



Relationship architecture - framework

Nodes in the architecture will *implement* components of the framework. It is recommended that the framework structure is similar to the structure of the architecture so that there is a one-to-one correspondence between the two structures (at a suitable aggregation level).

Harmonising architecture - framework

Normally the framework is harmonised to reflect the architecture, while the architecture need not take the framework into account, see Relationships framework - architecture (p.-162).

Relationships architecture - implementation

The architecture is an abstraction of the concrete system intended to give overview and promote understanding of its construction. It should cover all parts of the implementation so it can be trusted as a faithful representation of it. At the same time it should not go into details that are better described using hardware description languages and programming languages. Such detail shall be found in implementation descriptions that are referenced from the architecture.

Architecture models give added value to the low level implementation descriptions through overview and by providing a single entry point to the entire documentation.

Harmonising architecture - implementation

Architecture implementation

- *Ensure that the architecture covers the implementation down to a level of detail where the remaining detail is well taken care of by reference to implementation descriptions.*
- Ensure that names and references are updated when changes are made.

Harmonising architecture specification - design

When the architecture has been designed, the architecture specification shall be harmonised. It shall be correct with respect to the actually provided properties, and it shall cover every property of importance for external assessment and use. Properties not needed for that purpose should be removed even if they were included in the initial specification.

Architecture specification

- *The architecture specification surviving after the architecture design is made should provide exactly the properties needed for external assessment and use.*

Summary of static architecture rules

The architecture models serve several purposes, see Architecture models (p.6-164), and should be structured to fulfil as many of those as possible. In addition the Architecture reference model (p.6-154) should be used to make a layered structure that facilitates evolution.

Documentation

- *From the architecture entities there shall be a clear relationship to detailed implementation descriptions either through naming conventions or explicit references.*
- *All detailed implementation descriptions shall be covered by the architecture.*

Hardware models

- *Hardware models shall identify at least all physical nodes needed to implement the abstract system.*
- *Whenever possible and practical one should make a hardware framework that separates between:*

  - *General platform hardware. This is hardware that is not application framework specific. It will typically consist of general purpose computers in a network.*
  - *Application framework specific hardware.*
- *Hardware models should define all hardware interfaces used to compose hardware systems, and in particular those used by application frameworks and software.*
- *Hardware models shall be sufficiently detailed and complete to provide the high level hardware documentation needed in addition to other hardware descriptions.*

Software models

- *Software models shall identify at least all software nodes needed to implement the abstract system.*

Software framework

- *Whenever possible and practical one should make software frameworks that separate between:*
  - *Support software, possibly with sub-layers: operating system, middleware; application programming support, application I/O support.*
  - *Application framework software. It contains the application and the infrastructure (software) implementations, and may be classified as automatically generated and hand written code.*
  - *Foreign code, code that is imported from other sources, legacy code, etc.*
- *Software models should identify at least every concurrent thread of behaviour, called a software process, and its communication links.*
- *Software models should define all software interfaces used by each layer of software.*
- *Software models shall be sufficiently detailed and complete to provide the high level software documentation needed in addition to source code.*

Developing architecture

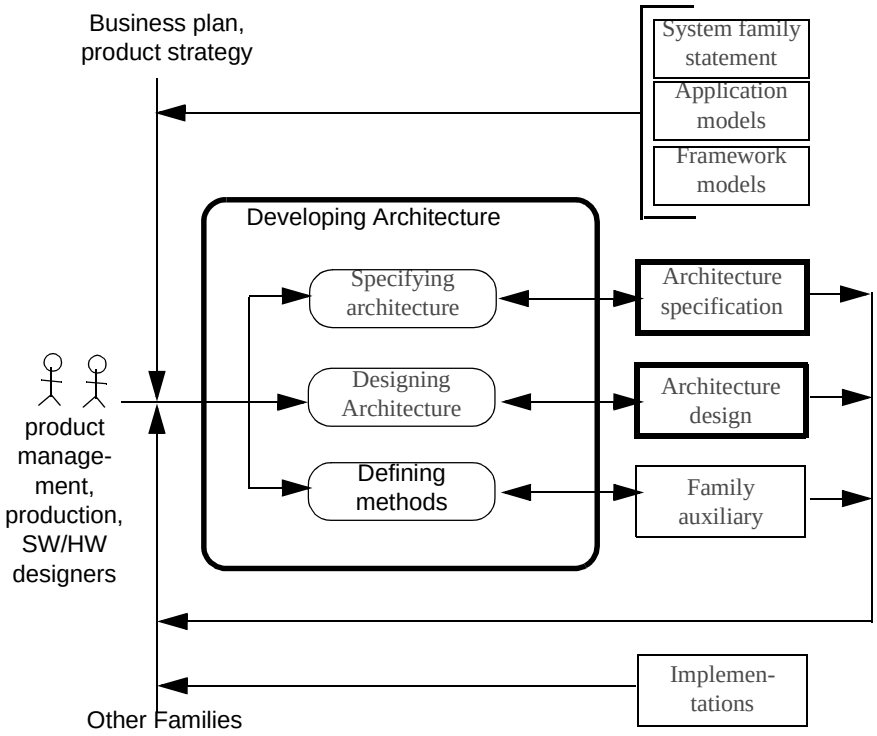
What

This activity develops implementation Architecture models (p.6-164) which consists of object- and property models organised as an Architecture specification (p.6-167) and an Designing architecture (p.6-177).

It also develops a Method for framework code generation (p.-96), and a Method for system instantiation (p.-97), see Figure 6-29 (p.6-106).

Figure 6-57: Developing Architecture

Open figure



Inputs The main sources of input information are:

- Descriptions:
  - Application models describe the application systems that the architecture shall implement.
  - Business plans and product strategy will give high level goals for the concrete system.
  - Other families and especially their architectures may provide valuable input. It may even happen that an architecture can be (re)used even if the applications are different.
  - The family statement should be consulted.
- People:
  - Product management can give additional requirements and guide-lines concerning the concrete system.

Outputs Architecture development results in an implementation architecture that will behave as defined in the Application models (p.-106) and satisfy the non-functional properties. It will also define a method for (automatic) generation of application implementation code and for configuration and building of system instances.

Who to involve

Concrete system design involves specialists in electronics, mechanics and software design. Such specialists tend to have a limited ability to perform high level system design. As this is very important in architectural design, an interdisciplinary team is required unless people with sufficiently broad system design experience are available. The team should not be biased towards a particular technology, but be able to perform trade offs wherever the solution is open. Such openness in design solutions is often essential in order to achieve innovation.

Users are normally not much concerned about architecture. But for the customer and the manufacturer it matters quite a deal. Not the least because it determines lifetime cost, the ability to evolve and the ability to interoperate with other systems.

Architectural designers need to work closely with production people, and specialists in software, electronics and mechanics.

When to do it

This activity 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 needed e.g. because of changes in the support software.

In an initial development, architecture design will come before framework/infrastructure design. During normal application evolution, the generic architecture will stay the same, and system evolution can take place mainly at the application level.

What to do

Making architectures

The following strategy may be used when making the first architecture model for a new system family:

Make Architecture

1. First the initial specification is created by Making architecture specifications (p.6-176). This will normally take place during requirements analysis.
2. When the application is designed, time is ready for Making architecture design (p.6-177). Use the application design and the architecture specification as input, and design the hardware and the software parts of the architecture.
3. Perform Harmonising architecture specification - design (p.6-172) to make a complete and precise architecture specification for the purpose of later (re)use.
4. Define a Method for framework code generation (p.-96) in order to simplify and streamline the generation of application system implementations.

Evolving architectures

The following strategy may be used when evolving an architecture:

Evolve Architecture

1. Add new properties or change existing properties in the architecture specification as required by Evolving architecture specifications (p.6-176).
2. Analyse the impact that the new or changed properties have on the design and perform the necessary changes by Evolving architecture design (p.6-177).
3. Evolve the Method for framework code generation (p.-96) if affected.
4. Evolve the Method for system instantiation (p.-97) if

Specifying architectures

Inputs

Inputs to this activity are the same as for Developing architecture models as a whole, see Figure 6-29 (p.6-106). There are two main sources of inputs: the market (represented by users and other domain stake holders) and the company itself (represented by product owners, production departments and service responsibles). Technology (that is what technology and solutions are feasible today) may be considered a third source.

Making architecture specifications

The initial architecture specification is normally made at an early stage before there are any application designs or frameworks. It shall express the initial requirements on the concrete system.

A possible approach is the following:

Architecture specification approach

1. *Consult the market and write down the general non-functional properties that it requires, see* User and customer related properties (p.6-169).
2. *Consult the various company departments and write down the company internal requirements, see* Company internal properties (p.6-170).
3. *Make the hardware specification object model.*
4. Make the software specification object model.
5. *Specify the non-functional properties that shall apply to each object and interface in the object models.*

Evolving architecture specifications

The reason for evolving an architecture specification is that some properties are to be added or changed.

1. *Specify the new or changed properties.*
2. *Analyse its impact on the architecture design and the specification object model.*
3. *(If the impact is acceptable then) make the changes to the specification object model.*

Designing architecture

Making architecture design

The main inputs are: (1) the Application models (p.-106) (in SDL or UML) and (2) the Architecture specification (p.6-167). When the architecture is made for the first time, there is normally not any framework.

The following step-wise procedure takes both hardware and software into account:

Make Architecture design

1. Perform Trade-off between hardware and software (p.6-177)
2. Make Hardware design (p.6-178)
3. Make Software design (p.6-178)

Evolving architecture design

This activity makes an incremental evolution of the architecture so that it satisfies a new specification increment. It involves the same basic activities as making architectures for the first time, only that the existing architecture is taken into account. At this stage of development an application framework may exist.

Evolve Architecture design

1. Analyse the impact of the specification increment on the Architecture design. Identify which parts that are affected and change them accordingly:
2. Evolve the Hardware design (p.6-178) if affected.
3. Evolve the Software design (p.6-178) if affected.

Trade-off between hardware and software

Guidelines:

Physical distribution
- *Analyse requirements to physical distribution of interfaces and services.*

Mean peak load
- *Calculate the mean peak load for each SDL process, channel and signal route. Allocate SDL processes to computers such that the sum peak load for each computer is less than a given load limit (typically 0.2–0.3).*

Real time response
- *Calculate the response times for time-critical functions and check that requirements will be satisfied. Use priority to ensure fast response. Isolate time-critical parts as much as possible.*

Reliability
- *Consider the need for redundancy. Add redundant units and restructure the system until requirements can be met.*

Hardware design

The task here is to define the overall (architectural) hardware design. It is based on the division into hardware and software where the main hardware units were identified.

A possible strategy:

1. *Describe the overall structure of general hardware, i.e. computers, networks.*
2. *Describe the application framework specific hardware.*
3. *Describe the physical interconnections between the hardware units.*
4. *Describe the signal synchronisation schemes and protocols to be used on the physical interconnections, to the extent they are not already covered in the specification.*

Some guide-lines:

Physical distribution
- *Distribute processes in a way where the bandwidth needed over physical channels is minimised.*

Hardware similarity
- *Look for similarities between hardware modules. Increase cost–benefit by using similarity to minimise the number of different component types needed and maximise the reuse of each.*

Software design

See Software design (p.6-178) for further details.

Implementation relations

Guidelines:

Implementation mappings
- *Define the implemented-by relations from software nodes to the application framework objects they implement.*
- *Define the implemented-by relations from physical nodes to the application framework objects they implement.*
- *Define the executes-on relationship between software units and computers.*

Software design

Guidelines:

Implementation mapping
- *Map the SDL description as directly as possible into the software implementation.*

I/O software

- *Look at the physical interfaces and design software modules that can take care of the physical layer, i.e. synchronisation, event detection, timing and format conversion.*
- *Try to hide the physical details of input–output in separate software modules that handle the physical layer and provide an SDL-like signal transfer service to the application software.*
- *When input–output modules are time-critical or need to perform event detection or need to wait, they should be implemented in software processes that may be scheduled independently from the application software.*
- *Always specify a max waiting time when waiting on external events in order to avoid infinite waiting in error situations. This applies to the input–output modules as well as SDL processes.*

Software communication

- *Look at the internal interfaces and choose a suitable communication mechanism for each, i.e. procedure calls, message buffers and continuous values.*
- *Prefer to use the most general and flexible communication scheme for SDL signals, i.e. buffered communication, except when performance is critical and direct procedure calls/method invocations are obviously sufficient.*

Support software

- *Use a general operating system that supports concurrent processes and buffered communication except when a simple sequential program structure is obviously sufficient.*
- *Select the implementation method for each SDL process. Use general support systems, RTS, whenever possible, to ease the implementation of application functions and to increase reliability.*

External signal priority

Give time-critical external events priority over internal processing.

Internal processing priority

Give the processing of internal signals priority over the processing of external signals

Load control

When overload occurs, give priority to service requests already in progress and delay fresh requests.

Free-pools

Limit the size (the number of free buffers) of free-pools where it helps to control the internal load in a system, in particular at the input side.

Use a separate message buffer free-pool for each consumer of message buffers.

Generality

- *Preferably use the most general and flexible implementation techniques wherever they satisfy the design constraints, i.e. implement :*
  - *communication by messages;*
  - *SDL processes by FSM-Support;*
  - *basic support by a general operating system.*

Optimisation

- *Confine time-critical functions to modules that can be optimised separately. Do not optimise more than necessary.*

Summary of dynamic architecture rules

t.b.d.

List of figures

The main descriptions used in TIMe 3
The facets of a type model 5
Specification and design related 6
The main development cycle 8
Elements of the Domain. 10
Parts of the (application) domain will be realised in the domain given part of systems 13
The main activities in TIMe. 20
Analysing 24
Domain - system iterations 26
Analysing domain 29
Analysing requirements 32
Designing 39
Software implementation 42
Domain Models 53
Domain model notations 54
Domain Object Models 55
The access control domain. 57
The class definition of Access Zone 57
Developing domain models 62
Software implementation 76
Application system reference model 81
Application specification and design 87
Ways that services are related to objects 89
A service layer and an interface layer 90
Two cases of interface given parts 91
Application system specifications 93
External Application relationships. 98
Internal application model relationships 101
Developing Application 106
Developing the various parts of an application 107
Application framework reference model 123

Framework models. 127
External Framework relationships 130
Framework definition and use 133
Framework definition with predefined instances. 134
Virtual block type as part of framework, with application specific instance specificed 134
Application specific virtual type at system type level 135
Framework with no instances 136
Virtual block type. 137
Creation of application specific processes. 137
Redefined block type with process set and actual context parameter 138
Framework with virtual creation procedures 139
Setup as a virtual procedure of CentralUnit. 140
Actual Access Control System. 141
Fragment of redefined setUp 141
Developing Framework 143
Redesigned Access Control system 146
Cluster with LocalUnits and ClusterUnits 146
AccessPoint used in both LocalUnit and ClusterUnit 147
Access Control System type as a framework. 150
Block type Cluster as part of framework for Access Control Systems 151
An actual system based upon a framework 151
Reference models for abstract and concrete system 155
Architecture reference model. 156
Architecture models 165
Architecture relationships 171
Developing Architecture 174

List of definitions

Actor 183
Application. 183
Architecture 183
Complete abstract system 183
Domain Statement 184
Framework 184
Helpers 184
Physical node 184
Software node 184
Stake holder 184
Subject entities 185
System family statement 185
Transactions 185

Actor

An actor is a stake holder that takes actively part in the services or work processes of a Domain.

Application

An application is an abstract system that provide the main services of a system and is therefore the most valuable part of a system from a user point of view.

Architecture

An architecture is an abstraction of a concrete system representing:

- the overall structure of hardware identifying at least all physical nodes and interconnections needed to implement an abstract system;
- the overall structure of software identifying at least all software nodes, software communications and relations needed to implement an abstract system (in terms of processes, procedures and data).

Complete abstract system

A complete abstract system models as completely as practically possible the abstract functionality implemented in a concrete system.

It covers the Application and the infrastructure functionality supporting the Application. Its behaviour is a valid model of the real behaviour and its structure is similar to the structure of physical nodes in the concrete system.

Domain Statement

A (problem) domain statement is a concise description of the problem domain with focus on stakeholders and their needs, the essential concepts, functions and work processes, rules and principles. It should also clearly state the nature of the problem, i.e. what one wants to achieve.

Framework

A *framework* is an abstract system or a collection of (large) system component with two parts:

- a redefinable application;
- a configurable infrastructure that takes distribution into account, and contains all additional behaviour and supporting functionality needed to support the application in the concrete system.

Helpers

These are general tools that are used by the actors to provide the services of a Domain. Examples are communication systems, radar equipment and keys.

Physical node

A physical node is a distinct physical entity, such as a computer, that implements one or more abstract system objects.

A physical node operates concurrently with other physical nodes.

Physical nodes may be aggregated and decomposed, but always in such a way that abstract objects are contained within physical nodes.

Software node

A software node is a distinct software entity, such as a software process (a concurrent thread), that implements one or more abstract system objects.

A software node will often operate concurrently with other software nodes, but not always.

Software nodes may be aggregated and decomposed, but always so that abstract objects are contained within software nodes.

Stake holder

A stake holder is someone or something holding an interest in something.

In TIMe, a stake holder is any person, institution or system with direct or indirect interest in the Domain, a Family or a System instance.

Typical examples are companies, users, operators, owners, and systems in the environment.

Subject entities

These are entities that are subject to manipulation, representation or control in the Domain. They may be materials in the case of a material transformation domain, e.g. moulding, or they may be entities represented in an information system, e.g. flights and seats, or they may be controlled machinery, e.g. a paper mill.

System family statement

The system family statement is a concise description of the system family with emphasis on specifications, i.e. the external properties.

Transactions

These are entities representing transactions or events in the dynamic behaviour of the Domain, e.g. the purchase of a car, or a user passing a door.