From architectural requirements to architectural design

Maarit Harsu
Institute of Software Systems
Tampere University of Technology
P.O. Box 553, 33101 Tampere
e-mail: `firstname.lastname at tut.fi`

Contents

1 Introduction

Product-line architectures emphasize software reuse among several closely related applications. Concerning product-line architectures, the requirements analysis and design of such applications are carried out together. These applications form a family sharing the same core architecture. Each application typically has a variant part the design of which is also supported by product-line architectures, for example, via parametrization. It is essential to find out the common features and components of the applications belonging to same family. Thus, the requirements analysis and design of a family of applications are more complicated than those of a single system.

In real-life software engineering, software requirements somehow lead to software design solutions (software architecture) in a more or less ad hoc manner. However, especially in the case of product-line architectures, it is important to find out such requirements that are architecturally essential. Conventionally, requirements are divided into functional and quality requirements, both of which can be architecturally essential. The experience of a software engineer seems to affect on how well the architectural process is performed and how successful the achieved architecture will be. However, it is useful to find a way (and a rationale) to divide the requirements into essential and less essential ones from the point of view of the architecture. In consequence of the above considerations, it is important to be able to somehow manage the process from the requirements to the architecture in a more systematic way.

The design process of a product-line architecture consists of several important aspects or phases. In architectural design, choosing the notation to describe an architecture and its requirements has an important role. For this purpose, we introduce and discuss some alternatives, such as UML (Unified Modeling Language) and ADLs (Architectural Description Languages).

The report describes the process from the software requirements to a software architecture (design). Such design process also includes selection of an appropriate architectural style. Thus, it is important to find out a mapping from requirements to design (including an architectural style). The process from requirements to an architecture is connected to the notations such that the notations can be used in describing both the requirements and design. Moreover, the report shows the mapping from one description to another. The description methods for an architecture and its requirements are applied to real industrial architectures.

The report proceeds as follows. Section 2 considers software architecting process in general to clarify the roles of notation establishment, requirements analysis, and architectural design. Section 3 concentrates on architectural description in general and by discussing the alternatives in such description. Section 4 deals with architecturally essential requirements and their description. These items are discussed mainly with a harvester system domain. Section 5 introduces how UML can be used to describe architectural design. This topic is discussed in telecommunication domain. Some parts of requirements description (under the latter domain) is also comprised in this section. Section 6 considers the mapping from requirements description to architectural design description. Finally, Section 7 concludes the report.

2 Software architecting process

Product-line software architectures consist of two parts: the common (application-independent) core architecture and the variant (application-specific) architecture. The former part includes those components that are common for (at least almost) the whole family of applications sharing the same architecture. The latter part includes (specialized) components that are specific for an individual application.

In the same way, the construction of product-line architectures is divided into corresponding two phases. The first phase, called *domain engineering*, takes care of producing the common core architecture, while the second phase, called *application engineering*, derives individual application from the core architecture. The second phase consists typically of composing and specializing of components, for example, by parametrization. This whole product-line architecting process is depicted in Figure 1.

As seen in Figure 1, both domain engineering and application engineering can be divided further into smaller phases which are considered in separate subsections in such detail that is necessary for understanding this report. More detailed descriptions can be found, for example, in [Har02a].

2.1 Domain engineering

This subsection first considers domain analysis in more detail. After that it concentrates on domain design and domain implementation which are discussed in less detail.

2.1.1 Domain analysis

Domain analysis produces *domain model* describing the domain. This model gives the boundaries for the domain in question and establishes the terminology of the domain. Moreover, it fixes the notations for descriping the applications of the domain. Such a notation can be, for example, UML (Unified Modeling Language) [RJB99].

An important task in domain analysis is *commonality analysis* describing both the commonalities and variabilities of the applications belonging to the domain. Commonality analysis studies the requirements and properties of the applications and the concepts in the domain. It can be applied to existing applications that are

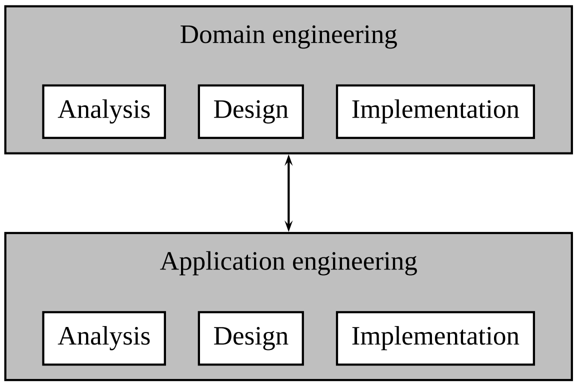

Figure 1: Product-line engineering phases

under re-engineering into a product-line fashion. In this case, besides the existing applications, more commonalities and variabilities can be found by examining the concepts of the domain. Moreover, commonality analysis can be applied to newly-designed applications. In this case, such an analysis covers only consideration of the domain concepts.

Another important task in domain analysis is *requirements engineering*. It comprises gathering, defining, documenting, verifying, and managing the requirements that specify the applications in the domain [CN02]. Concerning product-line architectures, the purpose is to reuse and configure the requirements among individual applications. Reusable requirements can also be called *features* [CE00]. Depending on the definition, a feature can cover both visible (for the end user) and invisible characteristics of the system.

Features can be depicted by feature models that also tell which combinations of features are meaningful. Feature models provide notations for different kinds of features such as mandatory, optional, and alternative features [KCH$^+$90]. However, these three feature types are not always adequate. For example, from alternative features you can select exactly one. In some situations, it should be possible to choose any number of features from a given set of features. Such features can be called or-features [CE00]. Besides different kinds of features, feature models may provide a way to present constraints among features or rationales to select features.

Figure 2 shows a feature diagram of a simple car. Mandatory features are marked

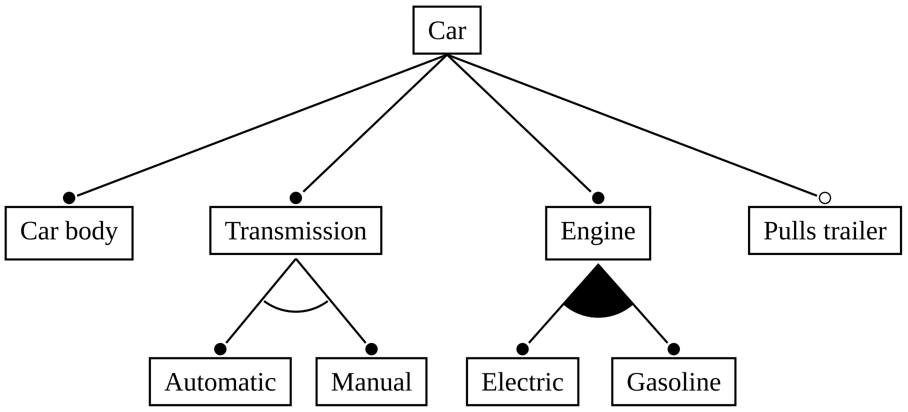

Figure 2: Feature diagram of a simple car [CE00]

with a filled circle in the head of the line. For example, a car must have a body, transmission, and an engine. Optional features have an empty circle in the head of the line. For example, a car can either pull a trailer or cannot. Alternative features are connected with an empty arc. For example, the transmission of a car can either be automatic or manual, but not both. A filled arc connecting features denotes or-features, meaning that you can choose one or several of such features. For example, a car may have an electric engine, a gasoline engine, or both.

Feature modeling is very close to conceptual modeling. Actually, the top item in a feature diagram (Car in Figure 2) is a concept having several properties called features. Each feature may have subfeatures (as shown in Figure 2). In addition, features have a very close connection to variabilities. Division into the subfeatures corresponds to a variation point. There are different kinds of variation points according to the division points of subfeatures. For example, a variation point corresponding to the division point of optional subfeatures is different from the one corresponding to the division point of or-features.

2.1.2 Domain design and domain implementation

Domain design means designing the core architecture for a family of applications. It comprises the selection of an architectural style [BMR$^+$96, SG96], which guides the design of the system. In addition, the common architecture under design should be represented using different views (for example [Kru95]). The core architecture should provide variability between applications, and in this phase, it is decided how to enable this variability or configurability. According to the feature models and commonality analysis, it should be selected which components or features (requirements) are provided in the core architecture and which items are implemented as variations in individual applications.

Domain design produces a *production plan* telling how the concerete application can be derived from the core architecture and from the reusable components. Production plan comprises descriptions of the systems with their interfaces per each customer, guidelines for the assembling process of the components, and guidelines for managing the change requests from the customers [CE00]. The assembling process may be manual, semi-automatic, or automatic.

Domain design may be involved in assessing the core architecture, i.e. analyzing the architecture against its quality requirements to reveal potential risks concerning the architecture. (Architecture evaluation is considered more precisely, for example, in [CN02] and in [Lah02]). An architecture should be evaluated at an early stage of the development of a product-line architecture, because the sooner the problems are detected the easier they are to solve [ABC+97]. Thus, as soon as there are artifacts to be evaluated, it should be checked how well the architecture meets its requirements.

Composing and other actions to provide final applications may require special tools that are outlined in the design phase. An environment constituted of such tools can be called an *application engineering environment* [WL99].

Domain implementation covers the implementation of the architecture, components, and tools designed in the previous phase. This comprises, for example, writing documentation and implementing domain-specific languages, generators, and other tools.

2.2 Application engineering

Application engineering uses the production facilities provided during domain engineering to produce applications of the family quickly. However, application engineering can be performed parallel with domain engineering (see Figure 1). Application engineering exploits those parts (tools, components) that are already available and implemented in the context of domain engineering.

The applications should satisfy customer requirements, and thus, application engineering is connected to customers either directly or via other people. Application engineering is an iterative process, because the customers are not necessarily satisfied with the application for the first time, and they may suggest improvements. Application engineering is also iterative with domain engineering, because the custom suggestions may have effect on the core architecture, and thus, on domain engineering.

The product-line architecting process described in this section can be supported by several process models. An example of such models is FODA (Feature-Oriented Domain Analysis) [KCH$^+$90] guiding domain analysis and requiremets analysis. It provides notations to describe different feature types (mandatory, optional, alternative). FODA has been further extended, for example, into FORM (Feature-Oriented Reuse Method) [KKLL99] to include also design and implementation phases. FODA can be used together with RSEB (Reuse-Driven Software Engineering Business) [JGJ97] in the form of FeatuRSEB (Featured RSEB) [GFd98]. In addition, FAST (Family-Oriented Abstraction, Specification, and Translation) [WL99] is a development process to produce software in a family-oriented way. (FAST is also considered in [Har02b].)

3 Architectural description

This section considers alternative ways to describe a software architecture. We first concentrate on UML and its profiles used in architectural description. Architectural description alternatives and architectural description in general are discussed next. After that, we introduce Bosch's archetypes [Bos00]. Archetypes are not an architectural description notation, however, they have an interesting relation to UML profiles. This relationship is considered last in this section. When comparing UML profiles with archetypes, we reveal similarities between archetypes, architecturally essential requirements, and architectural styles.

3.1 UML and its profiles

UML (Unified Modeling Language) [RJB99] is a graphical language for describing the models and design of software systems. It provides several kinds of diagrams to describe different aspects of a system. There are diagrams for high-level functionality, for static structure, and for dynamic behavior. UML diagrams are graphs consisting of elements and the relationships between the elements. Elements are graphical objects like rectangles and ellipses, when relationships are different kinds of lines.

Moreover, UML provides mechanisms to extend the basic elements of UML: stereotypes, constraints, and tagged values [Alh98, RJB99]. Stereotypes can be used to classify or mark elements and to introduce new types of elements. Each stereotype defines a set of properties that are received by elements of that stereotype. Constraints can be used to specify semantics or conditions that must stay true for elements. Constraints can be given as informal text or in OCL (Object Constraint Language). Tagged values can be used to specify keyword-value pairs of elements, where keyword is an attribute.

Together the UML extension mechanisms (stereotypes, constraints, tagged values) form a profile. UML profiles are useful in applying and customizing UML in different domains. UML is a large language having a plenty of diagrams that are not needed in every domain. Domains may have specific notions and particular needs, and thus, UML profiles can be used to restrict the notations of UML and to lead them to the appropriate direction. Predefined profiles exist for example for real-time modeling, data modeling, Web modeling, business modeling, and CORBA (Common Object Request Broker Architecture).

As an example, we consider UML profile for CORBA [OMG02]. CORBA enables defining client/server interfaces via IDL (Interface Definition Language). The profile specifies how to use UML in a standard way to define CORBA IDL interfaces, structs, unions, etc. It defines several stereotypes, for example «CORBAInterface», «CORBAValue», «CORBAStruct», and «CORBAUnion». These stereotypes are applied to classes to indicate what the class is supposed to represent.

Defining a profile includes several tasks. For a certain profile, such as CORBA profile, the subset of the UML metamodel needed in that profile is established. Such a subset includes, for example, the needed packages and metaclasses. Moreover, rules are needed to handle the elements of the selected subset. These rules can be given in a natural language or in OCL. The purpose of the rules is to ensure that the profile will be used in a disciplined way. In addition to the subset, specifications for needed stereotypes, tagged values, and constraints are given. For example, in the CORBA context, this means a hierarchy of stereotypes that model CORBA IDL. Such a hierarchy is defined as UML diagrams. The semantics for these specifications is expressed in a natural language. Finally, the profile definition includes a spefication for common model elements meaning that the instances of UML constructs are expressed in terms of the profile. CORBA profile, for example, defines a number of CORBA-specific type primitives, the definitions of which are given in this profile definition phase.

3.2 Architectural description in general

Besides UML, there are several other ways to describe an architecture. In such description, informal box-and-line diagrams are widely used. In addition, there exist several ADLs (Architectural Description Languages) [Cle96, MT00]. They describe a system architecture in an explicit and precise way. They provide mechanisms to describe architectural elements such as components, connectors, systems, properties, and architectural styles. ADLs are special-purpose notations having a great deal of expressive power. In addition to ADLs, there are architecture interchange languages such as ACME [GMW97]. These languages are meant to interchange architectural specifications across ADLs.

ADLs have the drawback that they are not well integrated with common development methods, and they do not support the concept of multiple views. Instead, UML is a widely used notation in different phases of software engineering, and thus, it is familiar to many developers. In addition, it can be directly mapped to implementations, and it has commercial tool support. As a drawback, UML does not directly support architectural aspects. This deficiency can be minimized by applying the extension mechanisms of UML. However, there are different opinions between UML and ADLs in architectural description. In addition, there are such solutions where UML is integrated with some existing ADLs [RMRR98].

This report concentrates mainly on UML in architectural description. However, in the rest of this subsection we consider the properties of the description methods in general. These properties are actually aims or requirements that a description method should reach. The common characteristics of the properties are derived by comparing existing ADLs [GMW97]. The most important architectural elements requiring a description notation are presented below [GMW97, GK00, Sel01, ZIKN01]:

- *Components* are units of computation or storages of the system. Examples of components are clients, servers, filters, blackboards, and databases.

- *Connectors* define interaction protocols among components. They can be very simple or more complex ones. Examples of simple connectors are pipes, procedure calls, and event broadcast. Examples of complicated connectors are client-server protocol, middleware architecture, and SQL link between a database and an application.

- *Systems* represent configurations of components and connectors. Systems can be hierarchical: both components and connectors may have an internal architecture. Thus, we have different representations of the architecture at different abstraction and refinement level.

- *Ports* define the interface of a component. A component may have several interfaces with different types of ports. Ports can be divided into provides-ports defining the services that the component offers and into requires-ports defining the services that the component needs. Interfaces can be simple such as a single procedure signature, or they can be complex such as a collection of procedure calls that must be invoked in a certain order.

- *Roles* define connector interfaces. They specify the participants of the interaction. Examples of binary roles are caller and callee of the connector concerning remote procedure calls, reading and writing roles of a pipe, and sender and receiver roles of a message passing connector. Connectors may also have more than two roles.

In addition to the above list, there are other architectural aspects that are required to be described. First, non-structural properties represent additional information about the architecture [GK00]. For example, some ADLs allow calculation of system throughput and latency based on performance of the related components and connectors. Second, architectural styles represent families of related systems [GK00]. They define the way and the termilogy to be used in designing the architectural parts such as components, connectors, ports, and roles. Third, especially in the context of product-line architectures, notations are needed for example, for different feature types [Has01].

3.3 Archetypes

Archetypes are the core abstractions based on which the system is structured [Bos00]. Archetypes form a relatively small set of abstract entities that describe the major part of the system behaviour at an abstract level. Such entities are very stable, and most often they do not change even if the system evolves. However, archetypes are not the subsystems of the whole system. Instead, archetypes are spread over the whole system.

Identifying archetypes is a step in architectural design and more precisely in functionality-based design. Such identification depends heavily on the architect and her experience. Archetypes can be identified through a holistic perspective and by incremental understanding of the architecture. Archetypes can be found by detecting recurring patterns within the system in different places.

Recurring patterns can be collected into a set consisting of candidate archetypes. Such collection is analyzed to find the final archetypes. The analysis may reveal overlapping or synonym archetypes that should be merged. Some archetypes may be too concrete, and thus, they should be reconsidered to decide whether to make them more abstract or to discard them. At the end of the analysis, there may be two subsets of candidates that represent different perspectives of the system. From these two alternatives only one could be chosen. Such a single consistent set of archetypes leads to the architecture that follows *conceptual integrity* [Bro95]. Thus, components that are instances of the same archetype share common rules, design decisions, and structural elements. These common concepts promote the understanding of the system.

Archetypes may have relations between each other. The relations describe the control and data flow of the system. However, the relationships should not be generalization or aggregate relations, because especially generalizations suggest that the archetypes should be merged. Moreover, architectures are often described as components and connectors. However, the relationships between archetypes are not the connectors between components.

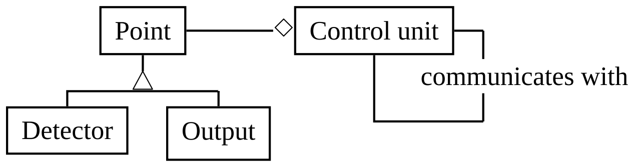

Figure 3: Example of archetypes [Bos00]

Identifying architectural entities is very close to domain analysis. However, archetypes cannot be found directly from the application domain. Instead, they are revealed through a creative process of abstaction. Moreover, the architecture of a system typically covers several domains.

As an example of archetypes, Figure 3 shows identified archetypes from a fire alarm system [Bos00]. Point (in Figure 3) represents the highest-level abstraction concerning the functionality of the domain. It is the abstraction of the two other archetypes: Detector and Output. Detector captures the core functionality of the fire-detection equipment, including smoke and temperature sensors. Output contains generic output functionality, including traditional alarms such as bells, extinguishing systems and alarm notification to fire stations. Control unit controls the behavior of a distributed fire-alarm system. It consists of small groups of points interacting with the detectors and outputs in the group. Control units are connected to a network and communicate with each other. The detector alarms in one control unit should often lead to the activation of outputs in other control units.

Figure 4 shows those components of the system that are related to the archetypes (in Figure 3). The component Physical point is an instantiation of the Point archetype. The Communication component, instead, is based on a solution domain. The Section component is also a domain entity representing a controller and the physical points monitored by it. This component is associated with the geographic area where the physical points identify alarm situations and react to them. The purpose of the Figures 3 and 4 is to get a conception of the difference between the archetypes and components of a system. (Note that Figure 3 depicts the generalization and aggregate relations, although Bosch does not courage to use them.)

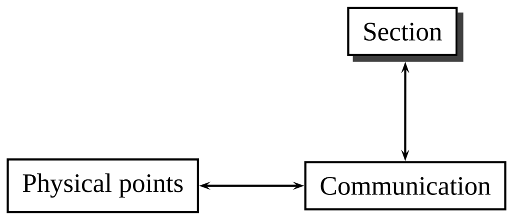

Figure 4: Example of main components [Bos00]

3.4 The relationship between archetypes and UML profiles

As mentioned in Subsection 2.1.1, requirements have a very close relationship to features. Features can be considered as reusable and configurable requirements [CE00]. Moreover, the relationship between features and requirements is that features are more abstract descriptions that users can understand easily, while requirements are more specific expressions about the same item [LW00]. Because features are expressed at a more general level, several requirements can be derived from one feature by refining the feature.

In architectural design, it is important to find out those requirements that are architecturally essential. Such requirements are related to archetypes. As discussed in Subsection 3.3, archetypes are system's core abstractions that are spread all over the system. The combination of the archetypes describes the major part of the system functionality at an abstract level. Because of the high abstraction, archetypes are very stable. In the same way, architecturally essential requirements form the core of the architecture and drive the design of the architecture. More precisely, archetypes are very close architecturally essential functional requirements, although we are making no sharp distinction between functional and quality requirements.

When considering archetypes and UML profiles, the relationship between them can be seen from two directions. On one hand, in the context of architectural profile, the stereotypes of the profile are very similar to archetypes. On the other hand, the relationships between archetypes can be seen as the (structural) constraints of the profile.

The purpose of archetypes is to provide a common termilogy and to make sure that the architecture is based on consistent and compatible concepts. Thus, revealing the archetypes is a process of restriction and refinement. Correspondingly, the purpose of applying UML profiles to a specific domain is to adopt UML to better follow the properties and restrictions of the domain. Such an adoption can be done by restriction and refinement. In the same way, both the archetypes and UML profiles correspond to an architectural style. An architectural style defines the vocabulary among the design constituents of the architecture such as components and connectors [SG96]. The purpose of all those three (archetype, architectural style, profile) is to reveal the core or base characteristics of the architecture.

When considering the relationship between archetypes and architectural styles, the main commonality is the definition of the vocabulary and the consistent concepts. As the archetypes are recurring patterns, they can also characterize as small-grained styles. Architectural style is actually a large entirety, while archetypes are small occurencies detected over and over again.

4 Architectural requirements and their description

This section concentrates on architectural requirements. Requirements control the selection of an architectural style. When considering the requirements, a specific architectural style emerges in software engineer's mind. Actually, the choise (the chosen architecture) is a result of the mapping between the properties of each architectural style and the individual requirements. The chosen architectural style is such that its properties best covers the requirements of the architecture.

In this section, we first consider the analysis of requirements. After that, we concentrate on the analysis of different architectural styles. Then we present a guidance to map the architectural requirements to the best suitable architectural style. Finally, we introduce different ways to describe architectural requirements.

4.1 Requirements analysis

As a starting point, it is supposed that the requirements have already been identified. However, the requirements may be unspecified or unclear and they do not necessarily have any precedencies. Thus, all we need in this phase is some kind of list of (maybe unqualified) requirements.

Analyzing the requirements means considering the requirements and dividing them into their smaller constituents. The purpose is to make the requirements more concrete and to find out the meaning and rationale of the requirements in this specific case. As an example, we consider a harvester monitoring system. It is a condition monitoring system that measures and makes analyses about the condition of the harvester. The identified requirements of the system are as follows [Kor01]:

- efficiency
- extendability
- understandability
- compatibility and consistency.

When analyzing the above requirements, we can describe each requirement in the following way. First, efficiency means that data should be received continuously. However, the system is not required to be real-time, and thus, received data could be processed with a buffer. Second, extendability means that the components of the harvester may be augmented with new functionalities. Consequently, new measurable quantities can also emerge. Moreover, new measure points (sensors) may be added to the system. Thus, new functionalities are needed to handle the new measurable quantities and new sensors. Third, understandability is required to make the new software developers more easily acquaintance with the system. Fourth, compatibility is a necessity, because the results of the measurements are to be used also by other applications. Thus, a consistent mechanism for information passing between different application is needed. Measurements produce a huge amount of information that should be saved to a database to be read by several applications. It is also required that an aggregation could be produced of the measurements in the long run. This can be done by fading away the unnecessary details.

Analyzing and refining the requirements, as done above, makes it easier to realize the features of the system. Moreover, analysis is needed to see which requirements are the most important (architecturally essential), which are the implementation possibilities for the requirements, and how the requirements are related to each other (whether they are consistent or conflicting).

Requirements analysis and especially refining the requirements is closely connected to goal-oriented requirements engineering [vL00, vL01, YM98]. The purpose of goals is to clarify requirements. Identifying goals leads to ”why”, ”how”, and ”how else” questions, and further to easier reveal of the requirements of the stakeholders. Thus, the goal-oriented approach is a way to decompose requirements to also satisfy the needs of the customer. These kinds of aspects are considered during application engineering as introduced in Subsection 2.2.

4.2 Architectural style analysis

The purpose of architectural style analysis is to reveal the properties of each style, and to find out which requirements each architectural style supports. In this way, we can find out which architectural style provides an easy and natural way to implement a specific requirement. On the other hand, we also find out which style prevents or makes it difficult to implement a specific requirement.

There are catalogs containing architectural styles and their properties. Such a catalog can be found, for example, in [BCK98, SG96]:

Data-centered architectures

emphasize integrability of data. They are appropriate for systems that describe the access and update of a widely accessed data store. Subtypes of these architectures are repository, database, hypertext, and blackboard architectures.

Data-flow architectures

emphasize reuse and modifiability. They are appropriate for systems that describe transformations on successive pieces of input data. Data enters the system and then flows through the components one at a time until some final destination is reached. Subtypes of these architectures are batch-sequential and pipe-and-filter architectures.

Virtual machine architectures

emphasize portability. They simulate such functionality that is not native to the hardware or software on which it is implemented. They can, for example, simulate platforms that have not yet been built (such as new hardware) or ”disaster” modes that would be too complex or dangerous to test with the real system (such as flight and safety-critical systems). Examples of these architectures are interpreters, rule-based systems, and command language processors.

Call-and-return architectures

emphasize modifiability and scalability. They are the most general architectural styles in large software systems. Subtypes of these architectures are main-program-and-subroutine architectures, remote procedure calls, object-oriented systems, and hierarchically layered systems.

Independent component architectures

emphasize modifiability by separating various parts of the computations. They consist of independent processes or objects that communicate through messages. They send data to each other but do not directly control each other. The messages can be passed to named receivers or they can be broadcast such that interested participants pick up the messages. Subtypes of these architectures are communicating processes and event systems.

Architectural styles can also be called architectural patterns. Besides the above styles, Buschmann et al. introduce the following styles and patterns to be exploited in each style [BMR$^+$96]:

Distributed systems

include one architectural pattern: broker. The broker pattern consists of decoupled components that interact by remote service invocations. A broker component coordinates communication by forwarding requests and by transmitting results and exceptions.

Interactive systems

include model-view-controller and presentation-abstraction-control as their patterns. Both of these patterns support human-computer interaction. The model part of the former one contains the core functionality and data. Views display information to the user, while controllers handle user input. Views and controllers together form the user interface. The latter kind of pattern provides a hierarchy of co-operating agents. Each agent manages a specific functional part of the system and consists of three components: presentation, abstraction, and control. With this division, human-computer interaction can be separated from the functional aspect of each agent and from the mutual communication of the agents.

Adaptable systems

include two patterns: microkernel and reflection. These patterns support applications to extend and to adapt to evolving technology and changing requirements. The microkernel pattern supports adaptation to changing requirements by separating the minimal functional core from the extended functionality and customer-specific parts. The reflection pattern supports dynamic changes to software systems. In this pattern, an application is divided into two parts. The meta level provides information about the selected system properties and makes the system self-aware. The base level includes the application logic.

The above two lists contain only examples of architectural styles, and there exist additional styles. Actually, each style defines a class of architectures. A style is an abstraction for a set of architectures applying that style. We cannot usually find clear occurrencies of particular styles. Instead, the styles appear in slightly different forms, and in some cases, the style is in the eye of the beholder.

Different architectural styles support different requirements. The purpose of the style analysis is to reveal the properties of each style to better make the decision which style to choose in a specific situation.

4.3 Mapping of requirements and styles

After we have analyzed both the requirements and the architectural styles, we should concentrate on the mapping between these two. Particularly, we are trying to find a systematic way to make the mapping. This kind of mapping is actually studied by Chung et al. [CNY95, CY98, CNYM00, TC99]. However, they only concentrate on non-functional (i.e. quality) requirements. To make the process from requirements to architectural design (style), the authors have defined a framework offering the following steps:

1. *Explicit representation of quality requirements* covers representing the requirements as conflicting or synergistic goals and guiding the selection process.

2. *Systematic use of architectural design knowledge* covers the methods to organize the knowledge concerning quality requirements.

3. *Management of tradeoffs among architectural design alternatives* covers correlation rules arising from goal conflict and synergy.

4. *Evaluation of goal achievement with a particular choice of architectural design* covers the guidance for selection among architectural design alternatives (architectural styles).

In the first phase, each requirement (i.e. goal) is considered by labeling it with importance value (e.g. critical) and with coverage value called parameter (e.g. system, process, function, data representation). The second phase applies decomposition methods to divide requirements and parameters into parts, as has been done in Figure 5. Together these two phases form a similar action to that done in Subsection 4.1. However, here the phases are presented more explicitly.

The third phase finds out the pros and cons of each architectural style in the context of the requirements. (The characteristics of several architectural styles are described in Subsection 4.2.) The results of this phase can be presented according to Table 1. The plus and minus in the table mean strong and weak support of an architectural style for a certain requirement, respectively. The scale could be presented even more accurately. For that purpose, Chung et al. provide more alternatives for correlations: strong positive satisficing (++), weak positive satisficing (+), weak negative satisficing (-), and strong negative satisficing (–) [CNY95].

The fourth phase provides the actual guidance for architectural design by combining the previous phases (i.e. requirements analysis and architectural style analysis). The purpose of this phase is to find a mapping between the requirements and the architectural styles. Such mapping is based on the evaluation of the styles (Table 1) and the decompositions (such as Figure 5).

According to the results of the mapping, the most appropriate architectural style can be chosen. However, the selection is not ad hoc one, instead it is based on the requirements and their importance. When the requirements are analyzed, they

Table 1: Correlation table [SG96]

	Shared Data	Abstract Data type	Implicit Invocation	Pipe and Filter
Change in Algorithm	-	-	+	+
Change in Data Representation	-	+	-	-
Change in Function	+	-	+	+
Performance	+	+	-	-
Reuse	-	+	-	+

can be set in an prioritized order, and such an architectural style should be chosen that best covers the most important (architecturally essential) requirements. However, several styles may be equal. If there are several satisfying styles, all the alternatives are good, and it does not matter a lot which one to choose. If none of the styles is satisfying enough, more styles can be analyzed, or perhaps create a system-specific style. If two alternatives are equally satisfying such that one style provides satisfaction for a group of requirements, and the other style for an opposite group of requirements, the situation is more difficult. The solution may be reached with a closer consideration of the requirements and styles. In some cases, it might be possible for each different part of the architecture to follow a different style.

4.4 Requirements description

There are several ways to describe requirements or features. A way to represent features is given in Figure 2. The diagrams like it provide notations for different kinds of features, such as mandatory, optional, and alternative. These notations are typically introduced with domain engineering methodologies such as FODA (Feature-Oriented Domain Analysis) [KCH$^+$90] and RSEB (Reuse-Driven Software Engineering Business) [JGJ97].

Feature diagrams are usually applied to describe functional features. However, quite a similar notation can be used to represent non-functional (quality) requirements, too [CNY95, CY98, CNYM00, TC99]. For example, extendability of the harvester system (see Subsection 4.1) can be described as Figure 5, where the arcs mean and-relationship.

According to the terminology of Chung et al. [CNY95, CY98, CNYM00, TC99],

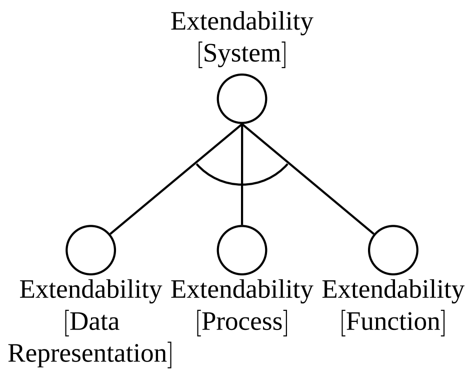

Figure 5: Extendability decomposition

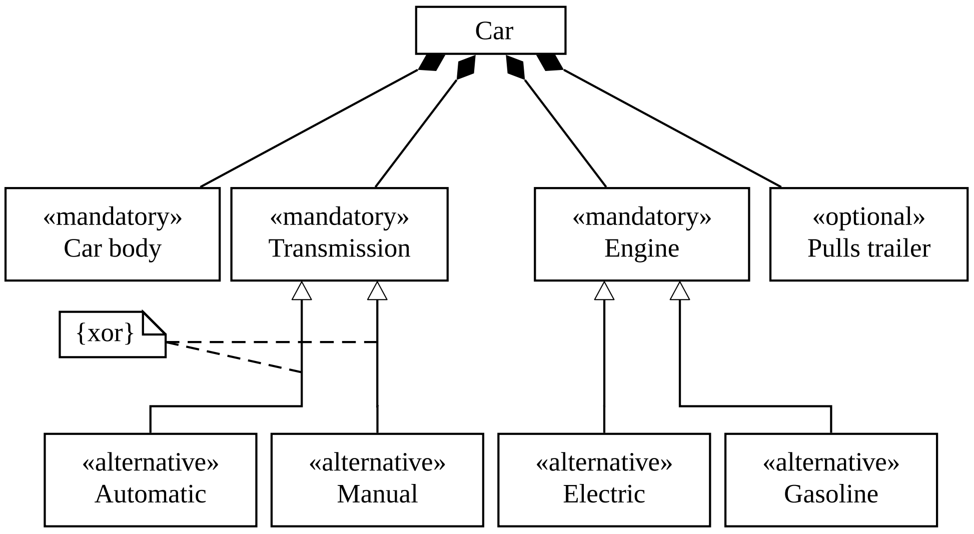

Figure 6: UML diagram for car features

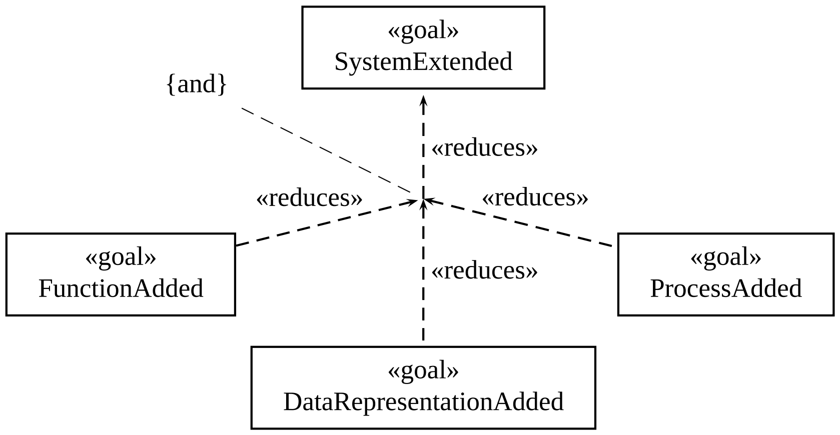

Figure 7: Requirements decomposition in UML

Figure 5 depicts parameter decomposition. Extending the harvester system with new measurable quantities brings new functionalities to manage the new quantities, new processes such as input process to collect the new quantities, and new data representations to describe the new quantities. In addition to parameter decomposition, Chung et al. introduce quality factor decomposition. For example, performance can consist of space performance and time performance. Similarly, modifiability can be decomposed into extensibility, updatability, and deletability.

In addition to the above ways to describe requirements or features, Clauß proposes applying UML to that purpose [Cla01a, Cla01b]. This notation is especially meant to describe the variability of features. It is closely related to the notation of the feature diagram (compare Figures 2 and 6): each feature kind (mandatory, optional, alternative) is represented as a corresponding UML stereotype. The relationship between the root and its siblings can be a composition or generalization relationship. Generalization is shown as a generalization arrow of UML and composition as a filled diamond. Exclusive and non-exclusive alternatives can be distinguished from each other by decorating the former with a {xor} constraint.

Another way to apply UML to requirements description is presented in [HF]. For example, extendability of the harvester system can be depicted as Figure 7. According to [HF], the requirements are represented in a goal-oriented way (in a past tense rather than as ”abilities”). Goals are represented as UML classes. It does not probably make sense to have several instances of goal classes. Thus, the stereotype «goal» introduces a singleton class.

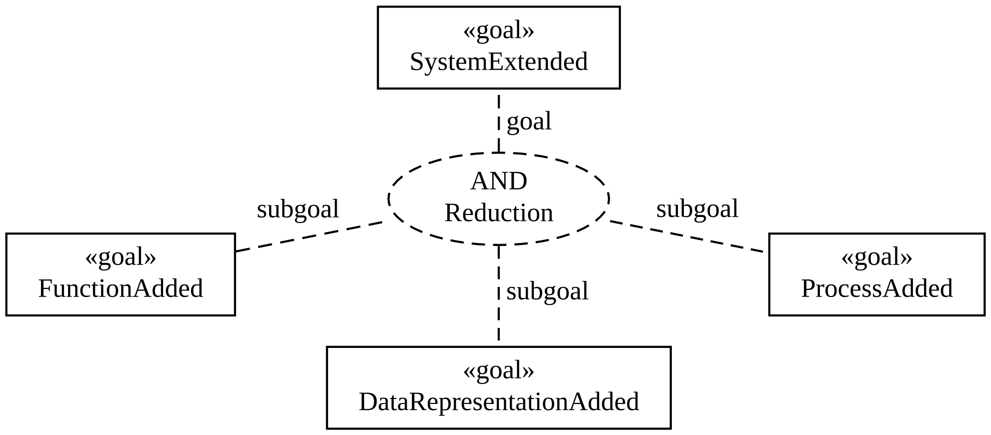

Figure 8: Another way to represent requirements decomposition in UML

According to the terminology of [HF], a goal is a requirement that is due to software. In addition to goals, a requirement can be derived from the domain in which case it is called an assumption and represented as the stereotype «assumption». Different goals have dependencies among each other. These dependencies are decorated with the stereotype «reduces» that is a specialization of the existing UML abstraction stereotype «refines». In addition to Figure 7, there is another way to express the same situation as depicted in Figure 8.

As shown, there are different ways to describe requirements. In sequel, we will mainly concentrate on UML representations, although there are some problems in this approach [Gli00].

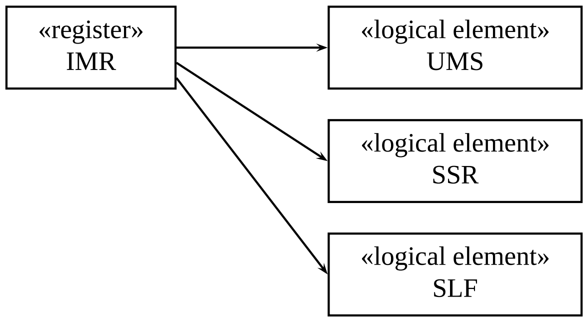

Figure 9: The relationship between logical elements

5 UML as an architectural description language

The purpose of this section is to show how both architectural elements (solutions) and architectural features can be described in UML. Such description is applied to a domain concerning third generation (3G) telecommunication network. This domain is introduced at the beginning of this section. After that we discuss some examples on describing architectural elements. Finally, we show some examples of architectural features to represent the architecture more precisely, although their description has already been considered in Subsection 4.4. Actually, architectural elements provide a higher-level view of the architecture, while architectural features represent the architecture in more detail.

5.1 Example domain

This subsection considers the domain which selected architectural description suggestions will be applied to. The domain in question is an industrial IMR (IP multimedia register) architecture where IP stands for ”Internet protocol” [Ein02]. IMR is a part of IMS (IP multimedia subsystem) network managing IP multimedia subscribers and services. IMR consists of three logical elements: UMS (user mobility server), SSR (service and subscription repository), and SLF (service and subscription locator function). The relationship between these logical elements is shown in Figure 9.

UMS concentrates on keeping information about IP multimedia subscribers. SSR is a storage for storing the actual implementation of IP multimedia services and related data. In addition, it keeps track of the services that each subscriber has ordered. SLF is used by network elements external to IMR to access the name of

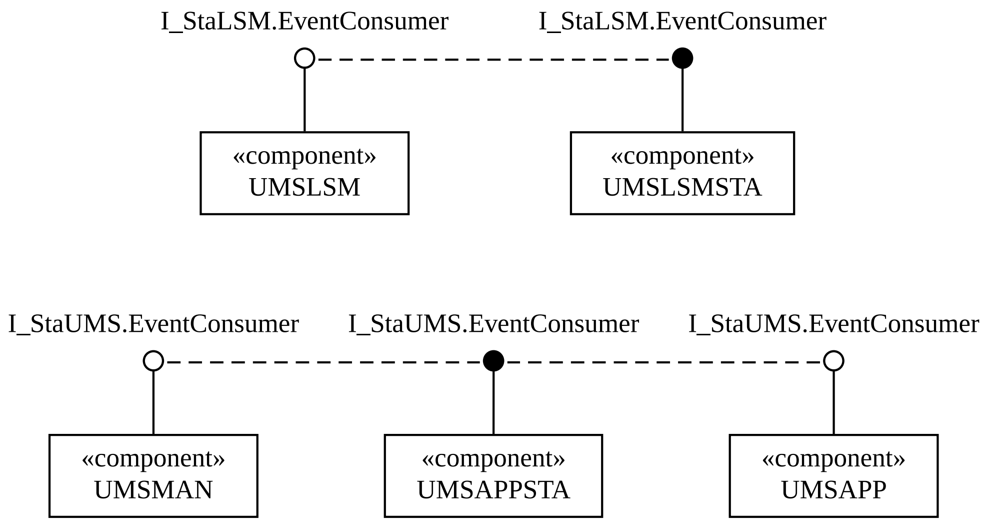

Figure 10: Examples of UMS statistics interfaces

UMS or SSR that keeps the required information [Ein02, Myl02]. Each logical element (UMS, SSR, SLF) has a very similar architecture, but their implementation varies. However, some services can be shared between logical elements.

5.2 Describing architectural elements

There are several alternatives to use UML to describe a software architecture [GK00]. Unfortunately, all of them have some drawbacks. It seems that those descriptions that emphasize completeness (by providing a semantic mapping for each aspect of architectural design) are too verbose. On the other hand, graphically appealing descriptions are usually incomplete. Thus, there is a need for a profile for architectural design or the UML meta-model should be extended.

There are several proposals of UML elements to be used in describing architectures. For describing an architectural component, at least UML class, component, package, and subsystem have been suggested [HNS99, Sel01, ZIKN01]. However, UML component corresponds to an executable software module, and thus, it is not very suitable for describing an architectural component. UML class is not very flexible, because it does not directly support the specification of composite architectural elements. Composition is provided by UML packages. However, UML subsystem is actually a combination of a package and a classifier, and thus,

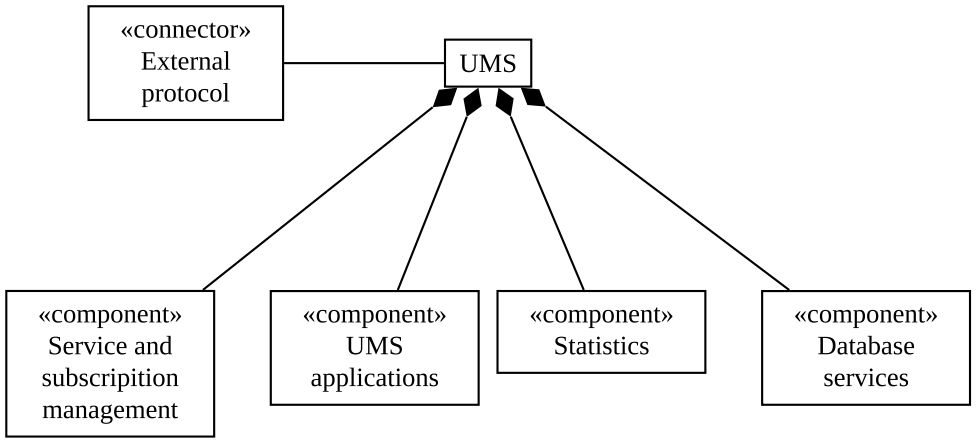
«connector» External protocol
UMS
«component» Service and subscripition management
«component» UMS applications
«component» Statistics
«component» Database services

Figure 11: UMS functionalities

it is often the most suitable solution to describe architectural components. In addition, stereotypes can be used to make it clear that the subsystem denotes an architectural component.

Architectural connectors can be very simple ones or more complicated. Simple connectors can be described with UML associations. However, in more complicated cases, classes [HNS99] or specializations from component (subsystem) and simple connector (association) [ZIKN01] can be used. In addition, they can be decorated with stereotypes. Roles can be described with stereotyped associations [HNS99].

Ports can be described with UML classes [HNS99]. Another solution is to use the UML ”lollipop” notation for interfaces. As mentioned in Subsection 3.2, ports can be divided into requires-interfaces and provides-interfaces. Thus, there are suggestions that requires-ports occupy the normal ”lollipop” notation while provides-ports are depicted as darkened lollipops [LR01]. However, the new version of UML (UML 2.0) will provide better support for software architectures, and thus, it will probably provide ports, too.

When considering the domain introduced in Subsection 5.1, Figure 10 shows the representation of provides-ports and requires-ports as suggested in [LR01]. This figure is modified from a figure belonging to a run-time view [Ein02].

Very often, it is necessary to consider a software architecture from several viewpoints [Kru95]. UML can be used to provide different views of the architecture

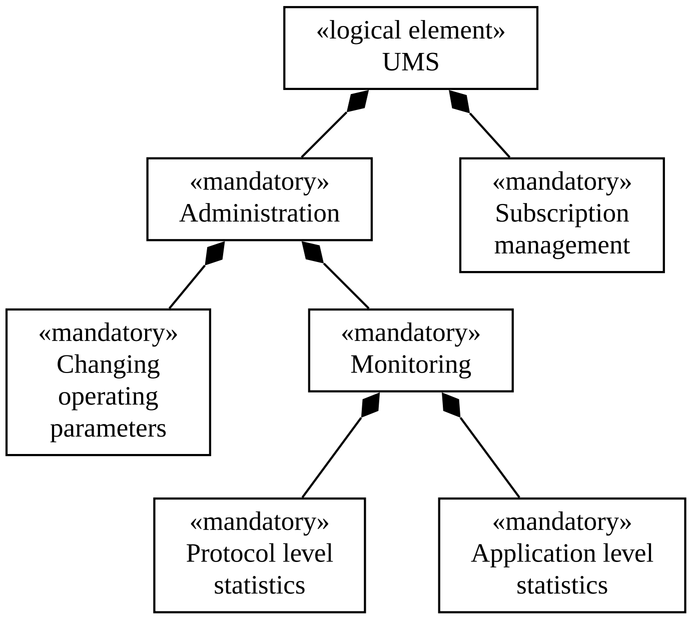

Figure 12: UMS features described in UML

[KS00]. Static views can be created by components, connectors, and constraints on them. For this purpose UML class diagrams, component diagrams, and deployment diagrams can be used. For behavioral views, UML statechart diagrams are often suitable. Configuration views depict instances of component and connector types. Different connector types can be decorated with stereotypes, and different icons can be used to describe different port types.

When considering the example domain, IMR architecture is currently described via several views: design view, data view, run-time view, and functional view [Ein02]. Design view shows the components, subsystems, and interfaces of the architecture. Data view depicts database connections. In IMR, the database is fragmented, and thus, a special mechanism is needed to locate a specific fragment. Database information is essential especially for SSR containing the subscriptions and services. Run-time view shows the configurations for each logical element as some kind of deployment views. Functional view provides scenario diagrams, for example, for system start-up and shut-down.

When considering, for example, data view, a representation for components and connectors is typically needed. The following figures follow a box-like notation where boxes mainly correspond to classes in UML. However, other UML symbols, for example subsystem symbol, could be used when more accurate informa-

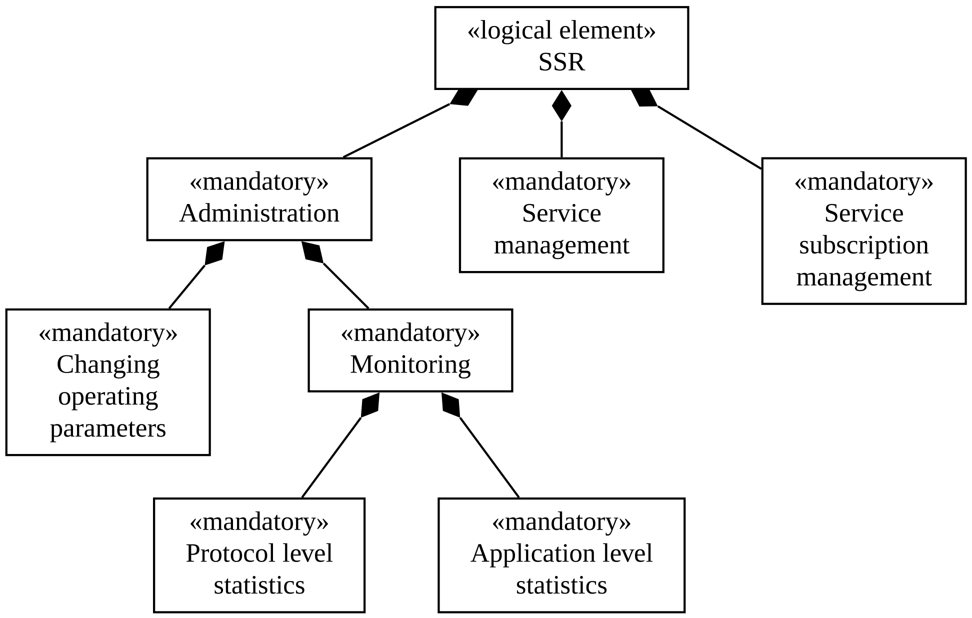

Figure 13: SSR features described in UML

tion is needed. Figure 11 shows necessary functionalities for UMS. A connector is represented as a class symbol of UML.

5.3 Describing architectural features

As shown in Subsection 4.4, there are several ways to describe architectural features or requirements. Concerning telecommunication domain, we mainly concentrate on functional requirements. We use here Clauß's notation, because it suits well our purpose, i.e. functional features.

When considering the example domain, each logical element (UMS, SSR, SLF) has a very similar architecture, but their implementation varies. The following figures show features related to functionality for each logical element. Logical elements are shown separately, because their functionality has only low commonality [Myl02]. Figure 12 depicts functionality concerning UMS element, Figure 13 for SSR element, and Figure 14 for SLF element. If there were more commonality, a common box could be denoted with a «common» stereotype (as has been done in Figure 15).

Figures 12, 13, and 14 are quite similar. However, the corresponding implemen-

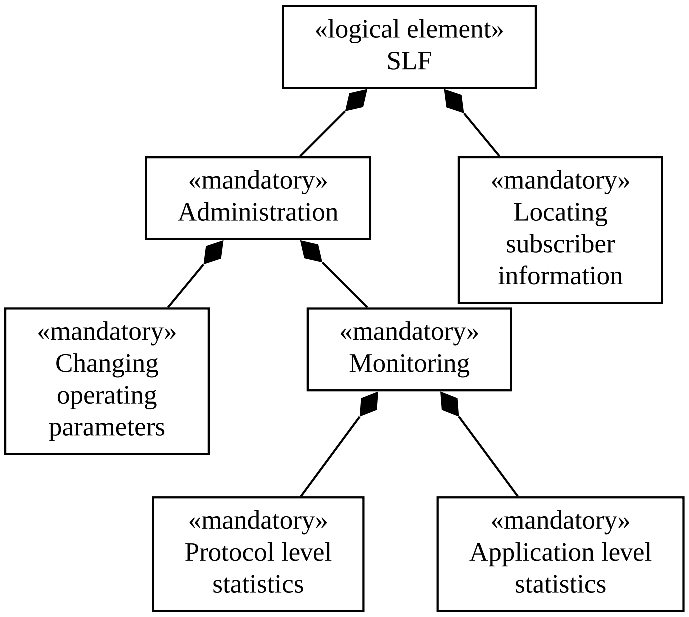
«logical element»
SLF
«mandatory»
Administration
«mandatory»
Locating subscriber information
«mandatory»
Changing operating parameters
«mandatory»
Monitoring
«mandatory»
Protocol level statistics
«mandatory»
Application level statistics

Figure 14: SLF features described in UML

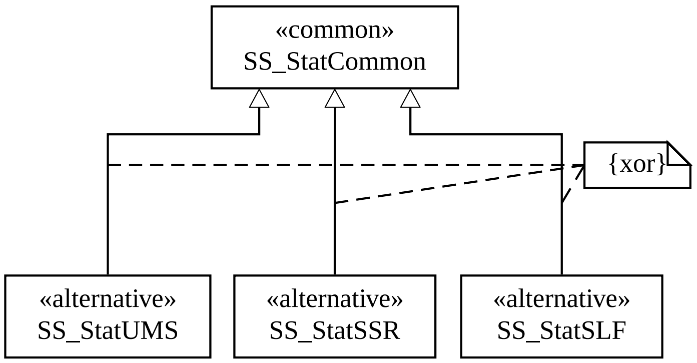

Figure 15: Alternative IMR features concerning statistics

tations vary according to each logical element. For example, each figure has a mandatory feature Changing operating parameters. Although it cannot be seen in the figures, this feature is specific for each logical element: UMS, SSR, SLF, respectively. Similarly, a mandatory feature Application level statistics is specific for each logical element. Moreover, a mandatory feature Protocol level statistics has different variants for different logical elements, because each element has a different protocol. Due to these differences, the features of the logical elements cannot be common.

All the logical elements concern with service and subscription management. UMS is responsible for adding, removing, changing, and reading IMS subscriptions. Thus, it has a mandatory feature for subscription management. SSR, in turn, needs both service and subscription management. It may add and remove services as well as change and read the implementation of service attributes. In addition, it adds and removes subscribers and changes and reads the service subscribtions of subscribers. SLF concentrates on locating subscriber information. It may add, remove, and change bindings to subscriber information.

Figures 12, 13, and 14 are derived from the feature descriptions presented in [Myl02]. In these figures, all features are mandatory, and all relationships are composition relationships. To show also generalization relationship and other kinds of features, Figure 15 is generated from [Ein02].

6 Description mapping from requirements to design

Until now we have considered on one hand requirements engineering and its description and on the other hand architectural solutions and their description. The purpose of this section is to find out a mapping between these two descriptions. We consider the problems associated with mapping separated from the solutions to the problems. We apply the solutions to the example domain introduced in Subsection 5.1.

6.1 Problems concerning mapping

When considering the process from architectural requirements to architectural design, some problems arise. Especially from the description point of view, there is the problem that the requirements models and the design models are not compatible with each other [CHOT99, Cla02, HF]. Requirements specification concerns with the features and capabilities of the system, while design documents (and code) focus on classes, methods, and interfaces. There are also other difficulties. On one hand, features or requirements typically have effect on many classes; such a problem is called *scattering*. On the other hand, a class may implement several requirements, which is called *tangling*. Thus, although described in the same notation, for example in UML, the requirements models do not directly lead to design models. Unfortunaly, UML provides no support for such mapping.

The above problem concerns particularly architecturally essential requirements, i.e. such requirements that are spread all over the system. The same problem lies in Bosch's archetypes which are the core abstractions of the whole system or some kind of recurring patterns that can be found from several points of the architectural solution. Such recurring pieces or instances of the main purpose of the system are very difficult to express in the design models. Instead, more local requirements (usually functional requirements) may more directly correspond to a specific class or package in the design model.

The lacking trace from the requirements model to the design model and especially the opposite trace causes problems also to the maintenance process and to the appearance of new requirements. If a clear trace existed between these models, the models created earlier phases of the software engineering process would not get out-of-date so easily as typically happens in real life.

In addition to the tracing problem, the misalignment between software requirements and design causes several other problems such as poor comprehensibility,

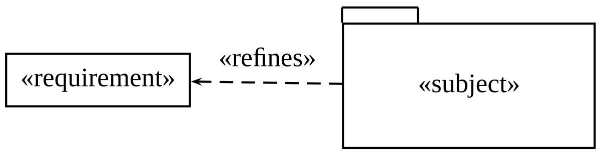

Figure 16: A UML package containing a design model of a requirement [HF]

coupling, poor evolvability, low reuse, high impact of change and reduced concurrency in development [CHOT99, Cla02]. However, here we are mainly interested in the tracing problem.

6.2 Solutions to mapping problems

Because of the appearance of the many-to-many relationship between the requirements and the designed UML elements, there cannot be a single dependency between a requirement and a design element. Thus, some tracing facility between these two is necessary.

The main problem can be considered as a inconsistency between requirements models and design models, although both models can apply the same notation such as UML. The problem is due to the many-to-many relationships between the elements of each model. A solution to this problem could be sketched as follows. First each requirement (especially architecturally essential requirement) should have a name. When describing the design of the system with UML diagrams, the names of such requirements that have lead to a specific UML element should somehow be found in this UML element. By this convention, we at least pay attention to the requirement that each UML element should satisfy.

A trace between requirements and design can be provided by *subject-oriented design* [CHOT99, Cla02]. Subject-oriented design is related to *subject-oriented programming* [HO93, OKK$^+$96]. Actually, each *subject* provides a different view to the whole system. Subject-oriented design and subject-oriented programming are related to *aspect-oriented programming* [KLM$^+$97]. Subject-oriented programming can be seen as a decomposition approach in aspect-oriented programming [CE00], or in other words, subject-oriented programming provides concrete mechanisms to implement aspect-oriented programming.

In subject-oriented approach, each requirement is considered as a subject that can

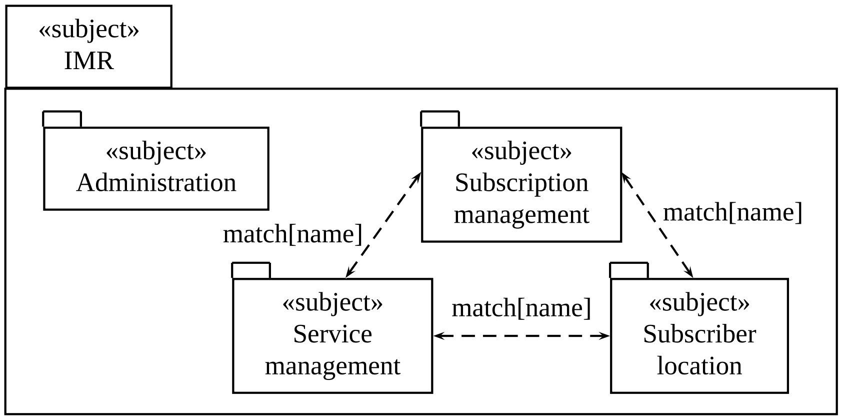

Figure 17: High-level subject composition relationships in telecommunication domain

be described as a UML package [CHOT99]. This relationship is shown in Figure 16. Further, each subject inside a package could be described more precisely as an own UML diagram. Moreover, UML can be applied to provide a high-level diagram showing the relationships between the subjects. Thus, the system can be viewed in pieces separated according to the subjects (or requirements) instead of only one monolithic diagram describing the whole system.

When considering the telecommunication domain (see Subsection 5.1), we can present a high-level view for subject composition relationship as Figure 17. Thus, IMR has subjects (or aspects) for administration, subscription management, service management, and subscriber location. In addition, there could be an outside subjects that are needed also with other systems than only IMR. The subjects related to services and subscriptions are connected to each other as described in Subsection 5.3. Figure 17 is derived from Figures 12, 13, and 14. We will consider the relationships between the subjects later.

Each subject in the high-level package can be described in more detail [CHOT99]. Now, each subject is described as a separate UML model, i.e. UML package. Different subjects (or packages) may be overlapping, describing those elements that are peculiar for that specific subject. Thus, the scattering problem is avoided, because one requirement corresponds to one UML package. Moreover, the tangling of multiple requirements in individual design units is resolved, because requirements are separated into different design models.

In the context of telecommunication domain, the focusing described above, is

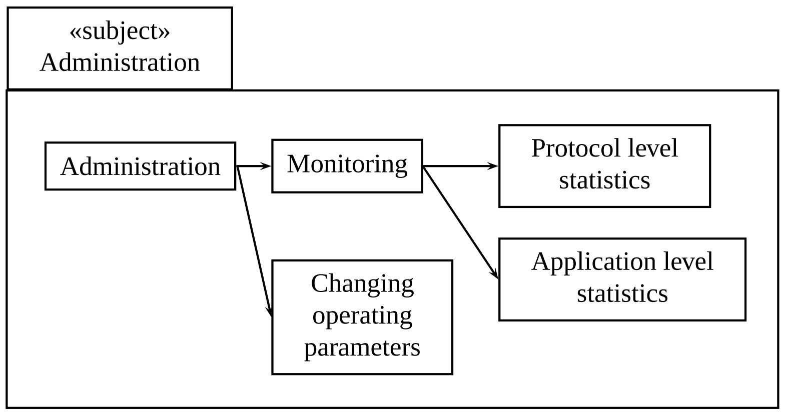
«subject»
Administration
Administration
Monitoring
Protocol level statistics
Application level statistics
Changing operating parameters

Figure 18: A structure diagram for administration subject

represented in Figure 18 showing the administration subject more precisely. The content of the package is derived from Figures 12, 13, and 14. Similarly, we could show the packages for other subjects, too. These packages could be partially overlapping.

With the above solution, requirements can be decomposed into separate UML packages. However, there is also need to compose the separate subject together to understand the whole system design. For that purpose subject-orieted design provides a composition pattern. Composition can be carried out in two ways: by overriding and by merging [CHOT99, Cla02]. Overriding can be applied, for example, when an old solution should be overriden with a new one. This is described with a dashed arc from the new solution to the old one. These solutions are UML elements (such as operations of a class) from different packages. The arc may have label match[name] to indicate that the package elements having the same name correspond to each other. Merging, in turn, is needed because of the separate design of different subjects. In some situations such designs should be merged. This is indicated in the same way as overriding except that the arc has an arrow in both ends. Similarly, the arc can be decorated with the label match[name]. However, the composition described above requires extensions to the UML metamodel to support composition relationships.

The relationships described above can be seen in Figure 17. The relationships between services and subscribtions correspond to merging operations. The elements in the packages in Figure 17 have common elements. However, these elements cannot be overriden, instead they should be merged, when producing a common view about the elements.

In subject-oriented design, a system can be decomposed into separate (or nearly separate) subjects that together form the whole system. Moreover, there can be subjects (or aspects) that are common to several different systems. It is also possible to expand the high-level diagram to contain the information of all the subjects, i.e. the whole system.

7 Conclusions

This report considered product-line architecture process and a couple of key topics of it such as requirements engineering and notations needed in architectural design. We have paid attention to the description of both architectural requirements and architectural solutions as well as the mapping between these two. These topics were discussed in the context of industrial examples.

The topic of this report is related to architectural design and how it can be derived from requirements. The process from architectural requirements to architectural design is very often carried out in an ad hoc manner. The purpose of the report is to provide some guidelines for architectural design and requirements analysis. To make the design process more systematic we considered requirements engineering and its connection to architectural styles. The requirements can be used as a basis to systematically select the most suitable architectural style. Architectural styles were also discussed in connection to Bosch's archetypes and UML profiles.

The report introduced different notations that can be used in architectural description. UML was selected to the main notation due to its familiarity, availability, and tool support. Both the requirements and the design can be described in UML. However, there exist problems in the consistence of the requirements description (concerning features and capabilities) and design description (concerning classes, methods, and interfaces). Although there is not a direct mapping between these descriptions, such a mapping can be found.

Actually, mapping considered in this report is two-fold. On one hand, requirements are mapped to an architectural style (i.e. architectural design) to find out the most appropriate style. On the other hand, requirements description is mapped to design description to make the trace from requirements to design more visible in both diagrams. Thus, the former kind of mapping is related to the rationale behind design decisions, while the latter kind is related to notational aspects.

The report provides a description notation for a real industrial architecture. However, the description is not perfect, but it can be completed by such people that are more familiar with the example architecture. This report should give some ideas of how to go further in architectural design. In addition, the related report [Myl02] gives a more detailed description of the underlying industrial architecture.

There exists specific UML profiles for several domains, for example for CORBA [OMG02]. In addition, there are guides to design a UML profiles and stereotypes [BGJ99]. However, the report does not describe a UML profile for any industrial domain. Again, such a description would require more acquaintance with the domain.

References

[ABC$^+$97] Gregory Abowd, Len Bass, Paul Clements, Rick Kazman, Linda Northrop, and Amy Zaremski. Recommended best industrial practice for software architecture evaluation. Technical Report CMU/SEI-96-TR-025, Software Engineering Institute, Carnegie-Mellon University, 1997.

[Alh98] Sinan Si Alhir. *UML in a Nutshell: A Desktop Quick Reference*. O'Reilly, 1998.

[BCK98] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.

[BGJ99] Stefan Berner, Martin Glinz, and Stefan Joos. A classification of stereotypes for object-oriented modeling languages. In *Second International Conference on the Unified Modeling Language (UML'99)*, pages 249–264, Fort Collins, Colorado, October 1999.

[BMR$^+$96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, 1996.

[Bos00] Jan Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Addison-Wesley, 2000.

[Bro95] F. P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 1995.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[CHOT99] Siobhán Clarke, William Harrison, Harold Ossher, and Peri Tarr. Subject-oriented design: towards improved alignment of requirements, design and code. *Sigplan Notices*, 34(10):325–339, 1999. Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99).

[Cla01a] Matthias Clauß. Generic modeling using UML extensions for variability. In *OOPSLA Workshop on Domain-specific Visual Languages*, pages 11–18, Tampa Bay, Florida, October 2001.

[Cla01b] Matthias Clauß. Modeling variability with UML. In *Third International Symposium on Generative and Component-Based Software Engineering (GCSE'01), Young Researchers Workshop*, Erfurt, Germany, September 2001.

[Cla02] Siobhán Clarke. Extending standard UML with model composition semantics. *Science of Programming*, 44(1):71–100, 2002.

[Cle96] Paul C. Clements. A survey of architectural description languages. In *8th International Workshop on Software Specification and Design*, Paderborn, Germany, March 1996.

[CN02] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[CNY95] Lawrence Chung, Brian A. Nixon, and Eric Yu. Using non-functional requirements to systematically select among alternatives in architectural design. In *1st International Workshop on Architectures for Software Systems*, pages 31–43, Seattle, Washington, April 1995.

[CNYM00] Lawrence Chung, Brian A. Nixon, Eric Yu, and John Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishers, 2000.

[CY98] Lawrence Chung and Eric Yu. Achieving system-wide architectural qualities. In *OMG-DARPA-MCC Workshop on Compositional Software Architectures*, Monterey, California, January 1998.

[Ein02] Kari Einamo. HSS / SSR / SLF: software architecture description (SAS), version 2.0.0+. Technical report, Nokia Networks, 2002.

[GFd98] Martin L. Griss, John Favaro, and Massimo d'Alessandro. Integrating feature modeling with the RSEB. In *Fifth International Conference on Software Reuse (ICSR'98)*, pages 76–85, June 1998.

[GK00] David Garlan and Andrew J. Kompanek. Reconciling the needs of architectural description with object-modeling notations. In *Third International Conference on the Unified Modeling Language (UML'2000)*, October 2000.

[Gli00] Martin Glinz. Problems and deficiencies of UML as a requirements specification language. In *10th International Workshop on Software Specification and Design (IWSSD'10)*, pages 11–22, San Diego, California, November 2000.

[GMW97] David Garlan, Robert T. Monroe, and David Wile. Acme: an architecture description interchange language. In *IBM Centre for Advanced Studies Conference (CASCON'97)*, pages 169–183, November 1997.

[Har02a] Maarit Harsu. FAST product-line architecture process. Technical Report 29, Institute of Software Systems, Tampere University of Technology, January 2002.

[Har02b] Maarit Harsu. A survey on domain engineering. Technical Report 31, Institute of Software Systems, Tampere University of Technology, December 2002.

[Has01] Willi Hasselbring. Reference architecture modeling with the UML and Vital: a comparative study. In *1st ICSE Workshop on Describing Software Architecture with UML*, Toronto, Canada, May 2001.

[HF] William Heaven and Anthony Finkelstein. *A UML profile to support requirements engineering with KAOS*. Available from: http://www.cs.ucl.ac.uk/staff/w.heaven/paper.pdf [read 04/2003].

[HNS99] C. Hofmeister, R. L. Nord, and D. Soni. Describing software architecture with UML. In Patric Donohoe, editor, *Software Architecture*, pages 145–159. Kluwer Academic Publishers, 1999.

[HO93] William Harrison and Harold Ossher. Subject-oriented programming (a critique of pure objects). *Sigplan Notices*, 28(10):411–428, 1993. Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'93).

[JGJ97] Ivar Jacobson, Martin Griss, and Patrik Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success*. Addison-Wesley, 1997.

[KCH$^+$90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie-Mellon University, November 1990.

[KKLL99] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, and Kwandoo Lee. Feature-oriented engineering of PBX software for adaptability and reuseability. *Software — Practice and Experience*, 29(10):875–896, 1999.

[KLM$^+$97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Christina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, Jyväskylä, Finland, June 1997. Springer.

[Kor01] Mika Korhonen. A harvester monitoring application (in Finnish). Master's thesis, Tampere University of Technology, Department of Information Technology, May 2001.

[Kru95] Philippe B. Kruchten. The 4 + 1 view model of architecture. *IEEE Software*, 12(6):42–50, 1995.

[KS00] Mohamed Mancona Kande and Alfred Strohmeier. Towards a UML profile for software architecture descriptions. In *The 3rd International Conference on the Unified Modeling Language (UML'2000)*, York, UK, October 2000.

[Lah02] Essi Lahtinen. Scenario-based assessment of software architectures. Master's thesis, Tampere University of Technology, Department of Information Technology, April 2002.

[LR01] Chris Lüer and David S. Rosenblum. UML component diagrams and software architecture — experiences from the Wren project. In *1st ICSE Workshop on Describing Software Architecture with UML*, Toronto, Canada, May 2001.

[LW00] Dean Leffingwell and Don Widrig. *Managing Software Requirements: A Unified Approach*. Addison-Wesley, 2000.

[MT00] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

[Myl02] Tommi Myllymäki. A process of designing and maintaining a platform for IP multimedia related 3G network elements. Technical report, Institute of Software Systems, Tampere University of Technology, 2002.

[OKK$^+$96] H. Ossher, M. Kaplan, A. Katz, W. Harrison, and V. Kruskal. Specifying subject-oriented composition. *Theory and Practice of Object Systems*, 2(3):179–202, 1996.

[OMG02] OMG (Object Management Group). *UML Profile for CORBA Specification*, April 2002.

[RJB99] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[RMRR98] Jason E. Robbins, Nenad Medvidovic, David F. Redmiles, and David S. Rosenblum. Integrating architecture description languages with a standard design method. In *20th International Conference on Software Engineering (ICSE'98)*, pages 209–218, April 1998.

[Sel01] Bran Selic. On modeling architectural structures with UML. In *1st ICSE Workshop on Describing Software Architecture with UML*, Toronto, Canada, May 2001.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[TC99] Quan Tran and Lawrence Chung. NFR-assistant: tool support for achieving quality. In *The Second IEEE Symposium on Application-Specific Systems and Software Engineering Technology (ASSET'99)*, Dallas, Texas, March 1999.

[vL00] Axel van Lamsweerde. Requirements engineering in the year 00: a research perspective. In *22nd International Conference on Software Engineering (ICSE'2000)*, pages 5–19, Limerick, Ireland, June 2000.

[vL01] Axel van Lamsweerde. Goal-oriented requirements engineering: a guided tour. In *5th International Symposium on Requirements Engineering (RE'01)*, pages 249–263, Toronto, Canada, August 2001.

[WL99] David M. Weiss and Chi Tau Robert Lai. *Software Product-Line Engineering: A Family-Based Software Development Process*. Addison-Wesley, 1999.

[YM98] Eric Yu and John Mylopoulos. Why goal-oriented requirements engineering. In *4th International Workshop on Requirements Engineering: Foundations of Software Quality (REFSQ'98)*, pages 15–22, Pisa, Italy, June 1998.

[ZIKN01] Apostolos Zarras, Valerie Issarny, Christos Kloukinas, and Viet Khoi Nguyen. Towards a base UML profile for architecture description. In *1st ICSE Workshop on Describing Software Architecture with UML*, Toronto, Canada, May 2001.