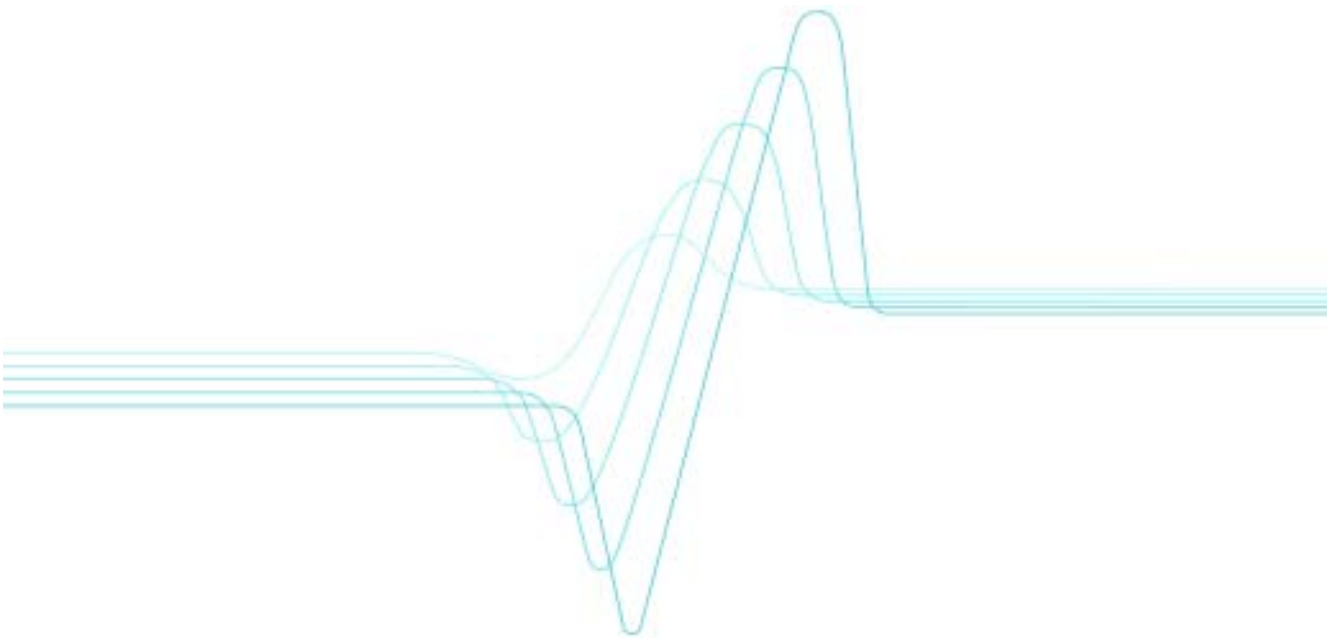
ESPOO 2002 VTT PUBLICATIONS 477

Anu Purhonen

Quality driven multimode DSP software architecture development



VTT PUBLICATIONS 477

Quality driven multimode DSP software architecture development

Anu Purhonen
VTT Electronics

ISBN 951–38–6005–1 (soft back ed.)
ISSN 1235–0621 (soft back ed.)

ISBN 951–38– 6006–X (URL: http://www.inf.vtt.fi/pdf/)
ISSN 1455–0849 (URL: http://www.inf.vtt.fi/pdf/)

JULKAISIJA – UTGIVARE – PUBLISHER

VTT, Vuorimiehentie 5, PL 2000, 02044 VTT
puh. vaihde (09) 4561, faksi (09) 456 4374

VTT, Bergsmansvägen 5, PB 2000, 02044 VTT
tel. växel (09) 4561, fax (09) 456 4374

VTT Technical Research Centre of Finland, Vuorimiehentie 5, P.O.Box 2000, FIN–02044 VTT, Finland
phone internat. + 358 9 4561, fax + 358 9 456 4374

VTT Elektroniikka, Kaitoväylä 1, PL 1100, 90571 OULU
puh. vaihde (08) 551 2111, faksi (08) 551 2320

VTT Elektronik, Kaitoväylä 1, PB 1100, 90571 ULEÅBORG
tel. växel (08) 551 2111, fax (08) 551 2320

VTT Electronics, Kaitoväylä 1, P.O.Box 1100, FIN–90571 OULU, Finland
phone internat. +358 8 551 2111, fax +358 8 551 2320

Technical editing Marja Kettunen

Otamedia Oy, Espoo 2002

Purhonen, Anu. Quality driven multimode DSP software architecture development. Espoo 2002. VTT Publications 477. 150 p.

Keywords software engineering, quality, design methods, analysis methods, wireless systems

Abstract

Traditionally, DSP software development has concentrated on optimising the algorithms. The future wireless communication systems create challenges to the DSP software. In order to handle the new requirements, more emphasis has been placed on software architecture. This thesis examines the way quality driven architecture development can be applied to multimode DSP software. First, the main quality attributes for DSP software are defined. Performance ensures that the timing requirements are fulfilled, with simultaneously minimising the resource usage. Cost attribute ensures that the development of the system is affordable. Variability is for evaluating how well the architecture can adapt to changes that are required to the system during its lifetime.

It is proposed that the DSP software architecture should be described with four architectural views. A logical view shows the required functionality in an implementation independent way; a physical view depicts the deployment of logical components to the hardware architecture and the interfaces that are relevant to the software; a process view is used for understanding the runtime functionality of the system; a development view describes how the system is actually implemented with today's software platforms and technologies.

The process of developing the architectural views is iterative and incremental. More details are added to the diagrams when the development continues. View development is a series of iterations between refinement of architectural structures and evaluation of the decisions made. An evaluation strategy is presented for comparing architectural decisions against quality requirements.

The results are validated with a case study of a future multimedia terminal that supports three systems: GSM, WLAN, and WCDMA. It is shown that the quality-driven development clarifies the design decisions so that it is easier to compare and refine architecture candidates.

Preface

This research has been carried out at the Technical Research Centre of Finland (VTT Electronics). The major part of the research is based on work done in the MULTICS project (”The Architecture Study of Next Generation Multistandard Mobile Terminal and Basestation”) at VTT during the years 1998-2001. The MULTICS project was an initiative that aimed to develop methods for system architecture design of multistandard products. The MULTICS project was jointly funded by Tekes (Technology Development Centre of Finland), VTT and industrial partners that included Nokia Research Center and Elektrobit Ltd. The work has been originally published as a licentiate thesis at the university of Oulu.

I wish to thank Prof. Juha Röning, who has been the supervisor of this thesis at the University of Oulu, for his comments and guidance. Prof. Eila Niemelä deserves my deepest gratitude for encouragement and feedback on the manuscript.

I would also like to thank my colleagues at VTT Electronics. I would like to thank Dr. Tapio Frantti for commenting the manuscript. I am grateful of co-operation of Mr. Tapio Rautio, Mr. Juha-Pekka Soininen and other members of the MULTICS project team. I also wish to thank Mr. Jarmo Kalaoja for many discussions on various topics of software architecture design. Thanks are also due to Ms. Marjo Jussila for proofreading this thesis.

Finally, Kari, my companion in life, thank you!

Oulu, June 2002

Anu Purhonen

Contents

List of abbreviations

3GPP	Third Generation Partnership Project
ABAS	Attribute-Based Architectural Style
ABD	Architecture Based Design Method
ADL	Architecture Description Language
ATAM	Architecture Tradeoff Analysis Method
COM	Component Object Model
CORBA	Common Request Broker Architecture
COTS	Commercial Off-The-Shelf
CPN	Coloured Petri Nets
CPU	Central Processing Unit
DMA	Direct Memory Access
DSP	Digital Signal Processing
FPGA	Field Programmable Gate Array
GSM	Global System for Mobile communications
HW	HardWare
I/O	Input/Output
LQN	Layered Queuing Network
MSC	Message Sequence Chart
NFR	Non-Functional Requirement
OSI	Open Systems Interconnection Model
QARCC	Quality Attribute Risk and Conflict Consultant
QFD	Quality Function Deployment
SAAM	Software Architecture Analysis Method
SA/SD	Structured Analysis/Structured Design

SoC	System-on-Chip
SDR	Software Defined Radio
SPE	Software Performance Engineering
RAM	Random Access Memory
RMA	Rate Monotonic Analysis
RTOS	Real-Time Operating System
UML	Unified Modelling Language
WAP	Wireless Application Protocol
WCDMA	Wideband Code Division Multiple Access
WLAN	Wireless Local Area Network

1. Introduction

Digital signal processing (DSP) is an essential part of the mobile wireless systems. DSP is defined as the application of mathematical operations that are performed to represent signals digitally (Vihavainen & Marttila 1998). It ensures that the wireless signal can be transferred as faultlessly as possible through the air. The DSP system creates a platform on which various applications are built. Nowadays, a large part of the DSP systems is implemented in software.

Since the beginning of digital mobile communications, mobile wireless terminals have developed from simple phones for speech to nearly personal computers. In the future wireless systems, it is the Internet that will be the core application (Lu 2000). New applications, such as videoconferencing, are being planned. However, the channel capacity does not yet allow high-bandwidth services due to the limited spectrum available, power restrictions, and noise levels (Varshney & Vetter 2000).

The restricted capacity of the current networks has motivated the standardisation and development of next generation systems. Originally, the goal was to define one common air interface for all parts of the world (Ojanperä & Prasad 1998). However, today the general attitude is that the future wireless network should be an open platform supporting multicarrier, multibandwidth, and multistandard air interfaces (Lu 2000). In addition to several cellular network standards, also wireless local area network standards have been developed. Handsets supporting two networks are already available, but in the future, the users should get accustomed to use whichever air interface is available or select the most favourable one from several possibilities (Varshney & Vetter 2000). The task of the user would be made easier with a terminal that makes the various networks transparent to the user.

Increasing competition in the mobile phone business has resulted in a situation in which, in order to keep up with the development, new versions with more interesting features should be released regularly. Furthermore, different types of feature sets should be available to different types of users. In the future, terminals may be fully reconfigurable and new applications are loaded from the Internet. All this creates demands to the product development.

When the requirements to the product features are changing, also the technology that implements the features changes. More powerful processors and better compilers have entailed that the development of DSP software in C language is now possible (Oshana 1998). The software development tools and methods from other embedded software development can now be taken into use. However, DSP software still has some specific features so that the same tools and methods may not be fully applicable or some additional tools are needed.

In industry, since the beginning of the ‘90s, many companies have moved away from developing software from scratch for each product and instead focused on the commonalties between the different products and capturing those in a common architecture and an associated set of reusable assets (Bosch 1998). Traditionally, systematic reuse has been difficult in DSP software because each product has required a careful optimisation of resources so that the implementation language has been mainly assembly. Software architecture creates a common ground between software development and systematic software reuse. It can be used to form a common understanding of what is being developed between different stakeholders such as software designers, architects, managers, and marketing people.

Traditionally, features of a system that are not covered by its functional description have been called non-functional requirements (Buschmann et al. 1996; Bosch & Molin 1999). However, nowadays they are often referred to as quality attributes. The term non-functional requirements (NFR) may imply that there are requirements that exist independently of the system’s behaviour (Bass et al. 1998). There is a danger that viewing qualities in this way leads to their being omitted until the end of the specification task.

Quality attributes are of explicit interest when designing software architecture (Buschmann et al. 1996; Bass et al. 1998; Bosch & Molin 1999). When a good architecture cannot guarantee an attainment of quality goals, a poor architecture can prevent these goals from being achieved. The larger the system, the bigger is the effect of architecture on the quality attributes. All realistic, practical computing systems have to fulfil multiple quality attributes.

1.1 Definitions

The following are the main concepts used in this thesis:

Architectural style	An architectural style defines a family of software systems in terms of a pattern of structural organisation (Perry & Wolf 1992; Buschmann et al. 1996). More specifically, an architectural style defines components and connector types and rules how they can be combined.
Architectural view	An architectural view represents a partial aspect of a software architecture that shows specific properties of a software system (Buschmann et al. 1996).
Multimode	A multimode product means a product that can operate in several networks, sometimes even simultaneously (Soininen et al. 2001).
Product family	A product family is “a group of systems built from a common set of assets” (Bass et al. 2000).
Product line	A product line is “a group of products sharing a common, managed set of features that satisfy specific needs of a selected market or mission area” (Bass et al. 2000).
Runtime reconfiguration	Runtime reconfiguration means that the functionality of the system is changed during runtime without interfering the normal operation of the system. There are other names for it: dynamic, on-the-fly, or on-line reconfiguration.
Scenario	“A scenario is a brief description of a single interaction of a stakeholder with a system” (Bass et al. 1998). Scenarios are useful in validating both

	functionality and quality. The type of the scenario depends on the quality that is assessed.
Software architecture	Software architecture "is the structure or structures of a program or computing system, which comprise software components, the externally visible properties of those components, and the relationships among them" (Bass et al. 1998).

1.2 Scope of the research

1.2.1 Application domain

This research is focusing on the digital signal processing parts of the wireless mobile multimode systems. In the telecommunication domain, the functional grouping is often made using layers more or less according to the OSI-model. For example, a GSM radio interface is modelled in five planes (Mouly & Pautet 1992) and WCDMA in three planes (Ojanperä & Prasad 1998). Usually these basic planes are further divided into sublayers.

Figure 1-1 depicts a general division of wireless systems' radio interface functionality into layers. The scope of this research is situated in the upper-part of the lowest plane, L1, which is the same as the physical layer in the OSI-model. The responsibility of the lowest layer is typically the same regardless of the system or standard. It is devoted to the physical transmission of information between distant entities. The upper-part includes functionalities such as encoding, decoding and multiplexing. The focus of the thesis is the functionality that has been traditionally implemented by software using digital signal processors.

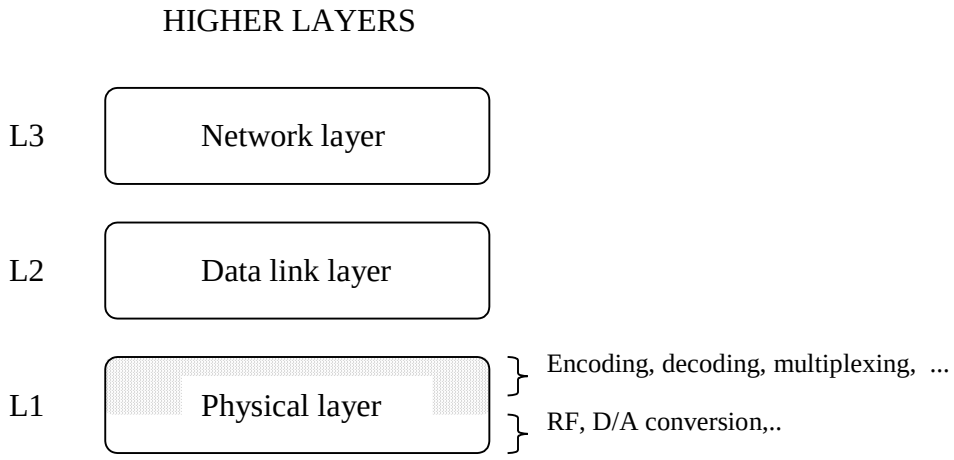


Figure 1-1. Radio interface layers.

The development of DSP radio software needs special skills. On the one hand, it is embedded software; on the other hand, it is data processing software. Extending the support for several standards causes new problems: the control of modes and even more efficient use of scarce resources.

1.2.2 Research domain

Software architecture is defined as a structure or structures of the computing system, which comprise software components, the externally visible properties of those components, and the relationships among them (Bass et al. 1998). The general software design flow includes tasks such as requirements specification, architecture design, detailed design, implementation, and testing. This thesis focuses on defining the software architecture design flow for DSP software. It is assumed that before the architecture specification begins, the requirements specification is already done. The utilisation of the software architecture during the detailed design and later phases of software development is only briefly discussed.

Taking an architecture-centric approach to software development has many advantages (Oreizy et al. 1998). Software architecture supports component reuse and helps to understand the system's overall behaviour. Application behaviour

can be separated from configurability issues when the decisions regarding change application policy and scope are encapsulated within connectors. Furthermore, control over the configurability issues is placed in the hands of the architecture designer who has the best overall knowledge of application requirements and semantics. In case of embedded systems, software requirements to the hardware can be studied before any detailed design has started.

The starting point for software architecture design in this thesis is the study of quality requirements. Software architecture development requires a knowledge of the product requirements, the development organisation, and the architecture selection and evaluation methods. In order to be able to select and evaluate architectural structures, the quality goals for the system have to be defined. Many software projects have failed because of a poor set of quality-attribute requirements although they have had well specified functional and interface requirements (Boehm & In 1996).

The focus of the thesis is on the generation of the architecture and reuse of existing methods in the evaluation of the architecture candidates. The goal is to find methods to provide help for the work of the DSP software architect.

1.3 Problem statement

1.3.1 Research problem

Software architecture methods have been studied in many research groups but none of the methods covers the specific problems with digital signal processing systems. In fact, it is impossible to define a general-purpose architecture development method because all of them are based on certain quality attributes (Bass et al. 1998).

Based on the previous discussion, the research problem is defined as follows:

How can architecture development take into account the special requirements of the multimode DSP software?

The research problem can be divided into subproblems in a following manner:

Q1. What are the quality attributes of the multimode DSP software?

Q2. What are the architectural descriptions that should be used to specify multimode DSP software architecture?

Q3. How do quality attributes affect the selection of the architectural structures?

Q4. How are quality attributes used in the evaluation of architecture candidates?

1.3.2 Research methods and results

The research strategy is divided into the following three phases: problem analysis, construction, and demonstration.

Problem analysis is performed in order to obtain product and domain knowledge. DSP software is constrained by the requirements from three sources: product specifications, standards, and hardware platform. Information was gathered using literature and interviews. In the DSP domain, standards serve as an important source in clarifying the requirements of the systems. Interviews were used for understanding the product requirements and the design decisions in the current software implementations. Problem analysis is presented in Chapter 3.

Construction is divided into three subphases: identification of quality attributes, architecture specification, and architecture evaluation. Identification of quality attributes is based on the problem analysis. Architectures are defined using architectural views (Kruchten 1995; Hofmeister et al. 1999a) and architectural styles and patterns (Buschmann et al. 1996; Shaw & Garlan 1996; Klein & Kazman 1999). Analysis methods are applied for evaluating individual quality attributes (Kazman et al. 1996) and for finding out conflicts between the attributes (Boehm & In 1996; Kazman et al. 1998). Quality attributes are specified in Chapter 4. Architecture development is covered in Chapters 5 and 6. Chapter 7 concentrates on architecture evaluation.

Demonstration validates the methods defined in the construction phase, with a case study of a future mobile terminal in Chapter 8. Quality requirements are analysed using the quality taxonomies developed in the construction phase. Equally, software architecture is developed and evaluated using the proposed methods. And finally, the applicability of the architecture-based design for multimode DSP software is evaluated.

2. Related research

The purpose of this chapter is to study the related research concerning architecture selection and evaluation. So far, there have been no published examples of architecture-based DSP software development. Vihavainen and Marttila have studied the high-level design of embedded DSP systems by using a hardware/software codesign approach (Vihavainen & Marttila 1998). It is similar to this work in that they emphasise that there should be a clear understanding of the required functionality before it is possible to consider implementation. In addition, they point out that there are various design metrics that have an effect on the composition of the final architecture. However, they concentrate mainly on the optimisation of hardware/software partition based on performance and resource usage. On the other hand, software radio research has addressed some points in the future DSP systems (Mitola 1995; Srikanteswara et al. 2000) but they are more concerned with the feasibility of the software radio concept than with how software should be actually developed by a manufacturer.

2.1 Software quality

Software quality attributes can be divided into three groups (Bass et al. 1998): system qualities, business qualities, and qualities about the architecture itself. System qualities are further divided into qualities observable via execution (e.g. performance, availability) and qualities not observable via execution (e.g. modifiability, integrability). They can also be called as development and operational qualities (Bosch & Molin 1999). Business qualities include cost and schedule considerations and marketing considerations. Conceptual integrity and buildability are examples of the qualities of the architecture itself.

Quality attributes are derived from system requirements, standards, and documents from old products and from interviews with the stakeholders. Quality attributes themselves are not definitive enough but they must be made more concrete. The approach applied in this work for concretising quality attributes are quality taxonomies (Barbacci et al. 2000). In these taxonomies, quality attributes have the following characteristics:

- Stimuli are operational or developmental activities that exercise the system. Stimuli are the events that require the system or the developers to respond.

- Response is the observable effects of operational or developmental activities.

- Architectural parameters link stimuli and response. They are the capabilities of the system, which affect the way the system or developers can respond to stimuli.

In another definition of the quality taxonomy, the properties of the attributes are classified into concerns, attribute specific factors, and methods (Barbacci et al. 1995). In contrast to the first definition, the concerns are defined in the response section. The attribute-specific factors can include both internal (i.e. architectural parameters) and external properties affecting the concerns. Stimulus is mainly described in attribute-specific factors. In this approach, methods and tools for addressing the concerns are also included.

The third approach to concretise quality attributes is to divide them into external and internal quality attributes (Briand et al. 1998). External quality attributes are the ones usually referred to as quality attributes: reliability, usability, maintainability, etc. Internal quality attributes such as coupling and cohesion are refinements of the external quality attributes. The refinement is necessary because usually, external quality attributes are not directly measurable.

Instead of quality attributes, Hofmeister and her colleagues discuss factors that influence the architecture design (Hofmeister et al. 1999a). These factors are categorised into three main groups and several subcategories. Organisational factors include aspects such as the schedule and budget, and the skills and interests of the people involved. They do not describe the product but they capture aspects of the organisation that could affect the architecture. Technological factors arise from the external technology solutions that are embodied in the product. They primarily include hardware and software technologies and standards. The third factor category is product factors that describe the product's requirements for functionality, the features seen by the user, and qualities such as performance.

Quality attribute taxonomies are used as a starting point when pursuing to find out the quality requirements of a system. A quality attribute analysis includes the following tasks: identification of quality attributes, characterisation of quality attributes, prioritisation, and identification of interdependencies.

Quality attribute identification and characterisation can be performed using questioning (Barbacci et al. 2000). Screening questions are used to uncover factors that are important to stakeholders and narrow the focus of evaluation. Elicitation questions deal with the way a quality attribute or a service that was identified as important is achieved by the system. The emphasis is on extracting concrete values to the parameters in the response and stimulus sections of the taxonomies. Analysis questions refine the information gathered by elicitation questions.

Domain analysis is useful for capturing the commonality and variability of related software systems. It is based on a study of existing systems and their development histories, knowledge captured from domain experts, underlying theory, and emerging technology requirements within the domain. Although commercial tools are rare, there are several approaches for domain analysis (Prieto-Diaz & Freeman 1987; Kang et al. 1990; Tracz et al. 1993; Coplien et al. 1998; Kang et al. 1999).

Quality attributes may contradict as well as complement each other. You need to consider the interdependencies and tradeoffs that exist between them and to define a preference of one requirement against another in case of conflict. Quality Function Deployment (QFD) is a widely used technique that helps translate customer needs into the technical requirements needed at each stage of product development (Akao 1990). QFD is supported by a specific graphical notation and by a well-defined process for translating requirements to realisation mechanism. In quality attribute analysis, it can be used to establish the relative importance between attributes and their values (Bot et al. 1996; Dueñas et al. 1998).

The Quality Attribute Risk and Conflict Consultant (QARCC) is a knowledge-based tool that can be used early in the life cycle to identify potential conflicts in quality goals (Boehm & In 1996). QARCC works by examining quality-attribute tradeoffs involved in software architecture and process strategies. It can alert

stakeholders to conflicts among their software-quality requirements and can help them identify additional, potentially important quality requirements.

In this work, the quality attributes are concretised with taxonomies that define stimuli, response and architectural parameters for each quality attribute. In addition, external factors are defined with an additional taxonomy because not only the product factors are important in the architecture development (Hofmeister et al. 1999a).

2.2 Architectural views

Several authors have stated that in order to fully specify a software architecture, multiple views should be used (Kruchten 1995; Soni et al. 1995; Bellay et al. 1997; Lung et al. 1997). An architectural view represents a partial aspect of a software architecture that shows specific properties of a software system (Buschmann et al. 1996). Such views can be used in different stages of development process and they can be developed iteratively.

Each view is composed of at least one structure. Structures can include module structure, conceptual structure, process structure, physical structure, uses structure, calls structure, data flow, control flow, and class structure (Bass et al. 1998). Each structure uses its own notation and architectural styles, and defines its own component and connector types as well as constraints.

Kruchten defines four views and proposes a use of scenarios to validate the other views (Kruchten 1995). Views are independent of notations and tools used as well as of design methods. Kruchten uses the following views:

- The logical view describes the object model of the design when an object-oriented design method is used. It is useful for functional analysis and identifying mechanisms and design elements that are common across the system.

- The process view describes the concurrency and synchronisation aspects of the design. It is useful for estimating the message flow and process loads.

- The physical view describes the mapping of the software onto the hardware and reflects its distributed aspect. The various elements identified in the logical, process, and development views are mapped onto the nodes in the physical view. Several different physical configurations can be used, e.g. for system deployment at various sites or for different customers.

- The development view describes the static organisation of the software in its development environment. Software is seen as program libraries and subsystems that can be developed by one or more developers. The development view is intended for the allocation of requirements and work to teams. In addition, it supports cost evaluation, planning, and reasoning out software reuse, portability, and security.

Kruchten uses a small subset of important scenarios – instances of use cases – to show that the elements of the four views work together seamlessly. This view is redundant with the other ones but it helps designers discover architectural elements during the architecture design, and it validates and illustrates the architecture design. A slightly modified version of this approach, using three plus one views, is suggested by Jaaksi et al (Jaaksi et al. 1999). They have combined process and physical views to a view called 'runtime view'.

Bass and Kazman point out that each view exposes some information and hides other information (Bass & Kazman 1999). Therefore, a view can be used for reasoning out, for example, performance, thereupon leaving out the other qualities for other views. Bass and Kazman use five views derived from those presented by Kruchten: functional structure, code structure, concurrency structure, physical structure, and developmental structure. Kruchten's development view more or less combines the responsibilities of their code and developmental structures. Properties such as maintainability, security, availability and capacity are attached to separate structures.

Hofmeister, Nord and Soni studied several organisations and architectures and observed that different structures were used at different stages of the development process (Soni et al. 1995; Hofmeister et al. 1999a). The structures they found fell into the following four categories:

- Conceptual architecture of a system describes its major design elements specific to the domain and the relationships among them. Conceptual architecture is independent of implementation decisions.

- Module architecture comprises two orthogonal structures: functional decomposition, and layers. Functional decomposition of a system captures the way the system is logically decomposed into subsystems, modules and abstract program units. Layers reduce and isolate external and internal dependencies. Unlike conceptual architecture, module architecture reflects implementation decisions; however, it is independent of any programming language.

- Execution architecture describes the dynamic structure of a system. It depicts runtime elements, communication mechanisms, assignment of functionality to runtime elements, and resource allocation. The structuring decisions are based on performance and distribution requirements, and the runtime environment.

- Code architecture presents how the source code, binaries, and libraries are organised in the development environment. The choice of the programming language, the development tools and environment, and the structure of the project and the organisation influence this architecture. It has generally not been described as part of the software architecture.

The Architecture Based Design Method (ABD) combines two of the above approaches (Bachmann et al. 2000). They create the conceptual architecture defined by Hofmeister, Nord and Soni using Kruchten's four views. Another way of combining is to add several structures to one view. Lung et al. have defined four views but included several diagrams in describing each view (Lung et al. 1997). Compared to the above approaches, they differ in that they have a map view for identifying the style, design violations and the mapping between components and functions or features. For example, tables can be used for creating a map view.

The number of views that are needed depends on the application. For example, Bellay and his colleagues came up with seven categories of architectural properties after studying large commercial embedded systems written in the

programming language C (Bellay et al. 1997). They include information exchange, system control, communication type, dynamic behaviour, structure, special quality requirements, and other non-functional requirements. However, they note that the categorisation is not complete and might be extended. Consequently, they propose some methods for representing the categories but in their opinion, any appropriate and already available technique can be used.

It is the division into problem architecture and solution architectures that is common to all these approaches. Every approach defines the major functional components and their relationships independently of the implementation with one view, the problem architecture. Another view that is used in every approach is the dynamic view describing the runtime behaviour of the system. In addition, an iterative approach and well-known methods such as object diagrams and message sequence charts are usually utilised in view development.

The approach in this thesis is a combination of previous approaches. It is based on Kruchten's four views (Kruchten 1995). The logical view differs from Kruchten's logical view in that it defines both hardware and software functions. The required functionality, the problem domain, is thereupon first covered before going into implementation platforms. Therefore, also the physical view covers all the hardware components, not just processors. The physical view is also the second view to be developed instead of being the last one because the hardware/software partition and the software distribution are decided upon in this context. The development view rather resembles the module architecture in (Hofmeister et al. 1999a). It is independent of the programming languages and the modules are defined at abstract level. In addition, in this work several models, of which some are optional, are included in each view. Scenarios are used for validating the functionality in each view.

2.3 Refinement

Stepwise refinement is an old concept in developing software (Wirth 1971). The development of architectural views is also a series of iterations from abstract architecture to a more concrete form. Concrete architecture should not lose properties of the abstract architecture and no new properties about the abstract architecture should be inferred from the concrete architecture (Moriconi et al.

1995). To ensure this, transformations should be made iteratively. Five categories for architecture transformations have been identified (Bosch & Molin 1999): imposing architectural styles, architectural patterns or design patterns, converting non-functional requirements to functionality, and distributing requirements.

An architectural style defines a family of software systems in terms of their structural organisation (Perry & Wolf 1992; Buschmann et al. 1996). Architectural styles can be divided into several categories such as dataflow systems (e.g. pipes and filters style) and data-centred systems (e.g. blackboard style) (Shaw & Garlan 1996). The purpose of architectural styles is to move architecture design closer to being an engineering discipline. Attribute-based architectural styles, ABASes, construct another development in this field (Kazman et al. 1998; Klein & Kazman 1999). ABASes associate architectural styles with quality attributes and give a reasoning framework for selecting architectural styles.

Architectural patterns and architectural styles are very similar; they can even have same names. However, architectural patterns are more problem-oriented than architectural styles. An architectural style only describes the overall structural frameworks for applications whereas a pattern expresses a very specific reoccurring design problem and presents a solution for it (Buschmann et al. 1996). Several authors (Buschmann et al. 1996; Douglass 1998) have studied architectural patterns.

Design patterns provide schemes for refining the subsystems or components of a software system, or the relationships between them (Gamma et al. 1995). Compared to architectural patterns, they generally affect only a limited number of classes in the architecture.

In some cases, non-functional requirements can be converted into functionality. Exception handling is a well-known example that adds functionality to a component thereby increasing the fault-tolerance of the component. Another way to overcome problems in fulfilling the non-functional requirements is to distribute them. For example, to achieve some performance requirement, a function is divided so that it can be performed in parallel in two processors.

One approach to refining architectures is strategies (Hofmeister et al. 1999a). Strategies are a way to reuse successful design approaches. They are usually more general than patterns. The view development starts with selecting key issues that should be solved in the architecture development. After this, strategies are developed for solving these key issues.

In the ABD method, the refinement is handled using so-called software templates (Bachmann et al. 2000). Software templates are types of design elements. They include patterns that describe how all elements of this type must interact with shared services and the infrastructure. Software templates are a way to inherit responsibilities. New responsibilities are added to them when they are refined, starting from the conceptual level and moving to the concrete level.

A taxonomy of Orthogonal Properties of Software Architecture (TOPSA) defines three dimensions for the software architecture (Bratthall & Runeson 1999). Abstraction level can be either conceptual or realisational. A conceptual architecture is not usually directly visible in the code. Aggregation level is high if a structure is hierarchically composed of lower-level structures, which in turn contain structures. In addition to the abstraction level and aggregation levels, which are mainly handled with the above methods, there is a third direction for transformations, namely, dynamism. Software architecture is not always static, i.e. it can change in time and sometimes even at runtime. The dynamic reconfiguration techniques may be useful especially in multimode systems.

This work utilises the concept of strategies including the possibility of adding new strategies to any phase of the architecture development. Strategies are solutions to the key issues that should be solved within an architectural view. In this thesis, the key issues are presented in the form of questions and therefore they are called problems. This is a less extensive way of defining the critical issues in the system, although in larger systems the approach in (Hofmeister et al. 1999a) could be useful. We will group all the possible refinements presented here under the term strategies. Writing down the strategies is one of the ways to make the system more understandable especially if the original designer is not available any more. In addition, the used strategies can be stored in one place, thereby making the actual architecture description shorter and easier to read.

2.4 Reconfiguration

In a multimode system, one feature that the architecture may have to support is runtime reconfiguration. Runtime reconfiguration means that the functionality of the system is changed during runtime without interfering the normal operation of the system. This is important in systems that have minimum downtime requirements. One part of the runtime reconfiguration is the downloading of the new functionality. Usually, runtime reconfiguration systems are designed for updating - for example, correcting errors. An obvious reason for runtime reconfiguration in multimode systems is adaptation - for example, adding new modes and deleting modes that are not needed to be supported anymore.

Before starting to design a system, one should decide what kind of configurability is needed. Runtime change is not a critical aspect of many software systems and additional functions are a risk to reliability; furthermore, its management may cause a performance overhead. Widely used means for getting configurable software are (Oreizy et al. 1998; Oreizy & Taylor 1998): dynamic programming languages (Lisp, Smalltalk), dynamic linking (used in general purpose operating systems like Unix and Windows), and dynamic object technologies (CORBA, COM). However, these methods are usually quite slow and they do not support controlled reconfiguration.

An important goal for a reconfiguration system is that it should preserve the application program correctness before, during and after the reconfiguration. This means that configurability should not cause any side effects to the normal operation of the system and the transfer between two configurations should be seamless. Furthermore, the system should be designed so that only valid changes can be done. The thing that separates runtime configuration from other techniques that provide dynamic modification capabilities such as dynamic linking and dynamic languages is change management (Segal & Frieder 1993; Oreizy et al. 1998). Change management takes care of preserving the program correctness in a runtime reconfiguration system.

The runtime reconfiguration can be divided into language-based and operating system-based approaches. For example, in the language-based approach Conic (Magee et al. 1989) there are separate languages for the configuration of logical nodes and for implementing the individual software components. Another way

of classifying the systems is to do it according to what kind of changes are performed. The changes in software can be divided into two categories (Wermelinger 1997): component level changes (change implementation, add or remove functions) and architectural level changes (add or remove components or connections). Changing the component internals is also called as source level reconfiguration (Stuurman & van Katwijk 1998). The application is seen as code: processes, functions, data etc. The goal is to be able to reconfigure any software, although there are usually certain restrictions to the way the software should be implemented in order to simplify the reconfiguration system.

When a dynamic reconfiguration system is aimed at architectural level changes, it is usually based on an architectural style that supports modifiability, such as C2-style (Oreizy et al. 1998, Oreizey & Taylor 1998) or subscription based style (Stuurman & van Katwijk 1998). Reconfiguration systems that manage changes at the architectural level are also usually based on tool suites (Oreizey & Taylor 1998). They can describe the architecture with a particular language and possibly the changes with a different language.

The most ambitious problem in reconfiguration research is to find methods for seamless updating. These methods can be also utilised in less difficult reconfiguration problems. The techniques that could be considered for embedded systems are usually process-based (Alonso & de la Puente 1993; Gupta & Jalote 1993; Hauptmann & Wasel 1996). Commercial operating systems have features that can be used in their implementation. In these systems, the most difficult problem is to transfer the state of the replaced process to the replacing one. In hard real-time systems, the focus is also to ensure the schedulability of the system (Alonso & de la Puente 1993). Gupta and Jalote have provided a theoretical framework for on-line modification of programs (Gupta & Jalote 1993).

Sometimes process-based systems are not fine-grained enough. PODUS is a procedure-oriented updating system where a program is updated by loading the new version of the program and replacing each old procedure with its corresponding new procedure during execution (Segal & Frieder 1993). Updating a procedure involves changing the binding from its current version to the new version. Implementation of PODUS is based on compiler and linker modifications.

In this work, reconfiguration techniques are not studied in detail. They are only included in order to take into account the ways the reconfiguration influences the definition of realisational components. The configuration unit and the basic guidelines of reconfiguration should be defined during the architecture design. In embedded systems, software may also have to control the reconfiguration of hardware.

2.5 Architecture evaluation

In this work, the process of analysing the properties of the architecture and comparing them to the quality requirements is called architecture evaluation. In some cases, architecture analysis can be equivalent to architecture evaluation. However, the results of architecture analysis can be used for several purposes but architecture evaluation is usually done for a specific purpose, for example for maintenance prediction or risk assessment. It is important that the goal of the evaluation is set beforehand because it has an effect on how the evaluation is actually performed (Lassing 2001). Usually, several architecture analysis techniques can be utilised in one architecture evaluation.

Architecture analysis techniques can be divided into quantitative measurements and qualitative questioning. Quantitative measurements are direct methods for acquiring information of a system, whereas qualitative questioning is an indirect method. Questioning techniques can be applied to evaluate architecture for any given quality. When answers to specific questions are needed, measuring techniques should be used. Following approaches for assessing non-functional requirements have been identified (Abowd et al. 1997; Bass et al. 1998; Bosch & Molin 1999):

- Scenarios. Scenarios concretise the actual meaning of a quality attribute. The type of the scenario used depends on the quality attribute to be assessed. For example, use-case or operational scenarios can be used for validating functionality and qualities that are assessed during runtime. Consequently, maintainability can be analysed with change scenarios. The effectiveness of the quality scenarios is largely dependent on the representativeness of the scenarios.

- Questionnaires. A questionnaire is a list of general and relatively open questions that apply to architectures of similar products. They are used in mature domains in architecture reviews. They exist before project begins. Questionnaires are tools that are directed more to the reviewers of the final architecture than to the architect.

- Checklists. Checklists are detailed sets of questions that are more focussed on particular qualities of the system, compared to questionnaires. Checklists are also tools for reviewers. Checklists grow out of the scenarios produced by several reviews.

- Simulation, prototypes, experiments. The main components of the architecture are implemented and other components are simulated, this resulting in an executable system. Simulation is particularly useful for evaluating operational non-functional requirements such as performance or fault-tolerance.

- Mathematical modelling. Various research communities have developed mathematical models that can be used to evaluate operational qualities (Klein et al. 1993; Xu & Kuusela 1998; Petriu et al. 2000). Once qualities can be discussed in terms of an abstract model, systems can be analysed to determine their quality attribute values. Performance is an example of such a quality where mature models exist and early architectural performance evaluations are possible. Mathematical modelling can be an alternative to simulation, or the methods can be combined.

- Metrics. Quantitative measures that are used for finding out particular properties of architecture, such as complexity, are called metrics. Metrics have often been developed for analysing source code but they can also be applied in analysing architecture. When using metrics, the evaluation needs to focus also on the validity of the assumptions under which the technique is used (Abowd et al. 1997). For example, metrics may have assumptions on the types of functionalities embodied in the components being examined.

- Objective reasoning. Experienced software engineers often have valuable insight that may prove extremely helpful in avoiding bad design decisions.

The previous techniques are related to each other. The objective reasoning can be generated into scenarios. Questionnaires are developed from mature scenarios. Checklists can be developed from mature questionnaires. Different techniques are used in different types of projects and in different phases of the projects. Usually combining different techniques is the best approach.

Traditionally, the analysing of real-time behaviour has been a combination of worst-case scenario experiments, engineering judgement, and empirical testing (Nord & Cheng 1994). Worst case scenarios are created and manually simulated to ensure that systems meet their real-time constraints. When there is code available it can be run in a simulator and the scenarios can be validated. However, often when architecture is being designed this is not the case.

Rate Monotonic Analysis (RMA) is a collection of quantitative methods that enable real-time system developers to understand, analyse, and predict the timing behaviour of many real-time systems (Klein et al. 1993). The term 'rate monotonic' comes from a method of assigning priorities to a set of tasks in which priorities are assigned as a monotonic function of the rate of a periodic task: the higher the rate, the higher the priority. RMA gives designers tools to mathematically guarantee that when the system is deployed and running, critical deadlines will always be met, even in worst-case situations. Although RMA can be applied manually, there are commercial schedulability analysers available that are based on RMA.

Software Performance Engineering (SPE) is actually a definition of a process to ensure that we are developing a system that will meet its performance goals than just an evaluation method (Smith 1990; Smith & Williams 1993). Several techniques can be used together with it but in the original description of the SPE, the performance assessment is based on two models: the software execution model and the system execution model. First, the (static) software execution models are used to confirm the basic system concept. Second, (dynamic) system execution models are used to examine external influences on performance. Although tools are not necessary, there are tools that support SPE type of performance evaluation.

Coloured Petri Nets (CPN) have been used for analysing the execution architecture of mobile phone software (Xu & Kuusela 1998). CPN is a formal

modelling method that can be used as a specification or presentation. It also can be extended with time, so it can be used for the validation of both functional and performance properties. CPN supports hierarchical descriptions so that large and complex systems can be modelled in a manageable way. CPN has computer tools supporting model building, simulation and formal analysis. The module architecture is defined with UML notation and the elements of the module architecture are mapped to the execution architecture defined with CPN. CPN models are used for simulating the UML use cases derived from the requirements. The simulation allows the comparison of the performance and timing properties of different communication mechanisms and control policies. The formal analysis can help in debugging the system.

Layered Queuing Network (LQN) is an extension of the well-known queuing network model (Petriu et al. 2000). The main difference is that a server, in which customer requests are arriving and queuing for service, may become a client to other servers from which it requires nested services while serving its own clients. An LQN model is represented as an acyclic graph whose nodes are software entities and hardware devices and whose arcs denote service requests. LQN has been developed especially for modelling concurrent and/or distributed software systems. A systematic approach can be used for building LQN models from UML descriptions of a system. Furthermore, the architectural patterns, such as pipes and filters, can be used as a basis for translating software architecture into performance models. Typical results of an LQN model are response times, throughput, utilisation of servers on behalf of different types of requests, and queuing delays.

Scenarios are used in many quality attribute specification techniques to form a profile. “A *profile* is a set of scenarios, generally with some relative importance associated with each scenario” (Bosch 2000). There can be different types of profiles depending on quality attribute. For example, usage scenarios from functionality validation or new ones can be utilised in performance analysis, hazard scenarios are used for specifying safety, and scenarios used to analyse variability create a change profile. The software architecture analysis method (SAAM) (Kazman et al. 1996) is the most cited scenario-based analysis method. However, there are also several other scenario-based methods, which define several differences or extensions to SAAM (Dobrica & Niemelä 2000). For

example, one of the approaches uses scenarios for analysing modifiability (Lassing et al. 2002).

Often, the analysis methods tend to concentrate on averages (Lassing et al. 1999). This is a problem, because really complex scenarios may be overlooked or their effect gets smoothened. Lassing and his colleagues defined a two-dimensional framework to structure the process of finding complicated scenarios. The dimensions of the framework consist of the source of the scenario and the types of adaptations that are needed.

Another work for supporting scenario elicitation consists of general scenarios (Bass et al. 2001). This work defines what kinds of scenarios can be defined for each quality attribute. A general scenario describes how the architecture should respond to a certain stimulus. General scenarios are used in the evaluation as a guideline or checklist to create the concrete scenarios. Using them could make it less likely that some scenario will be neglected. In addition, malformed scenarios that create ambiguity problems could be avoided.

Because there seems to be confusion in the research community even on how exactly the metrics should be defined (Briand & Morasca 1996; Weyuker 1988), it may not be easy to use them in the architecture evaluation. The other thing is that they are usually made for evaluating source code. The following metrics are promising also to the architecture evaluation:

- "*Coupling* is the measure of the strength of association established by a connection from one module to another" (Buschmann et al. 1996). Coupling measures especially for high-level design have been defined both for the module and the system levels (Briand et al. 1999).

- "*Cohesion* measures the degree of connectivity between the functions and elements of a single module" (Buschmann et al. 1996). Several measures have been used for analysing cohesion (Bieman & Kang 1998; Briand et al. 1999).

- *Structural complexity* is a system property that depends on the relationships between elements, and is not a property of an isolated element (Briand & Morasca 1996). Several complexity measures have been defined, for

example, McCabes cyclomatic number (McCabe 1976) and Oviedo's data flow complexity (Oviedo 1980).

- *Size* of the system can be defined as the number of modules, the number of procedures etc. Size has been used for example for estimating effort (Bengtsson & Bosch 1999).

Metrics-based and scenario-based approaches can also be combined (Briand & Wüst 2001). Although mathematical measures for metrics were not used, analysis of coupling and cohesion can be used together with scenario-based approach (Kazman et al. 1996). Low coupling means that a single scenario does not affect large number of components. On the other hand, high cohesion means that components are not hosts to scenario interactions.

Because the quality attributes are not independent of each other, one part of the evaluation should be to analyse whether there are conflicting requirements somewhere. In addition to conflicts between quality requirements, there can be so called architectural mismatches (Gacek 1998) which are conflicts that may occur when subsystems and components are integrated to one system.

The Architecture Tradeoff Analysis Method (ATAM) is a structured technique for understanding the tradeoffs inherent in the architectures of software intensive systems (Kazman et al. 1998). It was developed to provide a principled way to evaluate a software architecture's fitness with respect to multiple competing quality attributes. ATAM has grown out of the work on architectural analysis of individual quality attributes, SAAM (Kazman et al. 1996). The phases of the analysis method are (see Figure 2-1): scenario and requirements gathering, description of architectural views, attribute specific analysis, and tradeoff analysis.

This work utilises many of the previous methods. The idea behind selecting the methods was that it should be possible to use them without any tools and also that the evaluation can be done purely on the level of architecture. Simulation and prototypes are a good way to support analysis but in the early phases of the development it may not be possible to use them. Because our focus is on the tools for an architect, scenarios therefore constitute the basic approach; especially the ideas in SAAM and ATAM are utilised. RMA is good for

analysing performance because it can be applied manually but there are also commercial tools for it. There is no need to prepare any additional diagrams only for RMA. The purpose of the evaluation affects the way the evaluation is performed. Because the purpose of this work was to find tools for an architect, the evaluation results are mainly used for comparing architectural decisions.

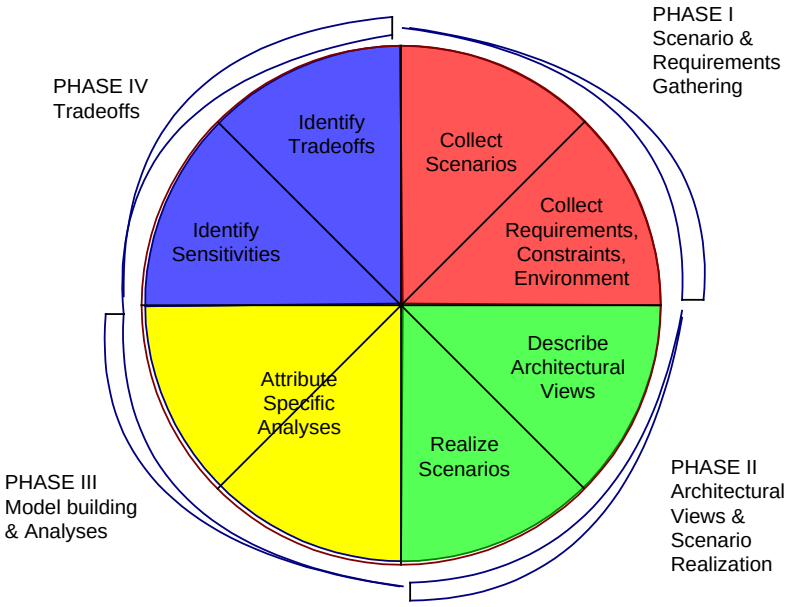


Figure 2-1. Steps of the Architecture Tradeoff Analysis Method.

2.6 Architecture design

Architecture design methods are usually applicable in creating new architectures or in analysing existing architectures. The design methods provide support in making the architectural decisions and selecting between solution candidates.

Bosch and Molin present an architectural design method that employs iterative evaluation and transformation of the software architecture in order to satisfy the non-functional requirements (Bosch & Molin 1999). The outline of the method is depicted in Figure 2-2. The method includes the following steps:

1. Architectural design based on the functionality. The non-functional requirements (NFR) are not explicitly addressed at this stage. The result is a first version of the architecture.

2. The design is evaluated with respect to the non-functional requirements. Each NFR is given an estimate in using a qualitative or quantitative assessment technique and the estimated NFR values are compared to the values in the requirements specification.

3. If the estimation results are not acceptable, an architecture transformation phase is executed. Architecture is improved by selecting appropriate transformation: imposing an architectural style, applying an architectural pattern, applying a design pattern, converting NFRs to functionality, or distributing requirements.

The phases 2 and 3 are repeated until all NFR are fulfilled or until the software engineer decides that no feasible solution exists.

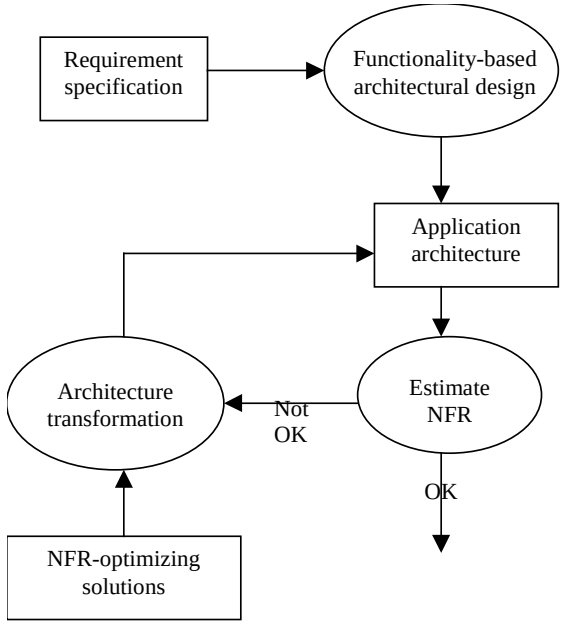


Figure 2-2. Outline of an architectural design method.

COMPARE framework, in Figure 2-3, can be used to analyse an existing system or to choose among competing architectures (Briand et al. 1998). The goal has been to take advantage of all the relevant and practical existing ideas, approaches, methods, and representations currently used in the field of software architectures, and integrate these into an operational framework.

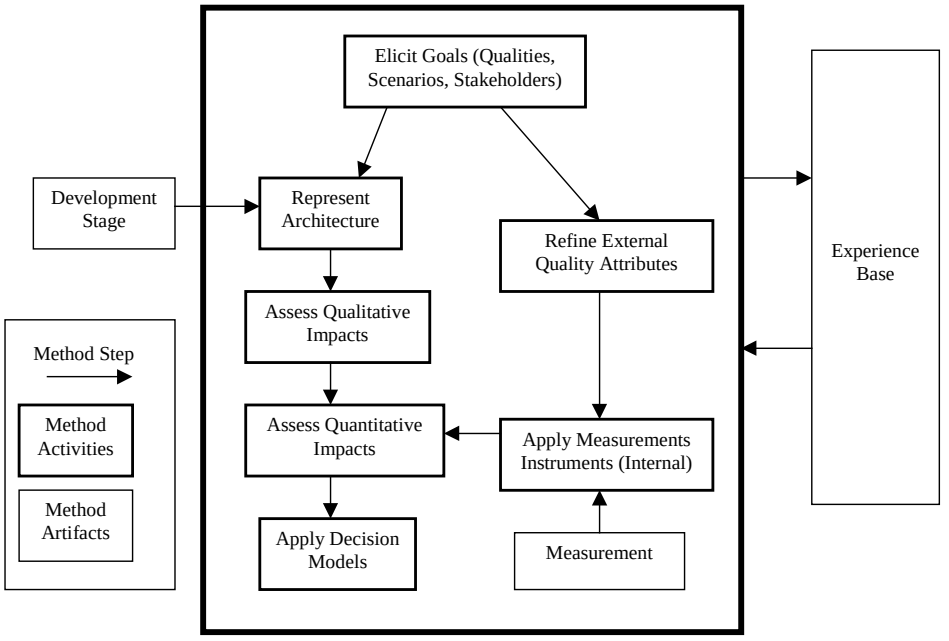


Figure 2-3. The COMPARE framework.

The goals in COMPARE are specified as scenarios. The development stage determines how much information is available in the representation of the architecture. Qualitative impact assessment is used for finding out the very areas of the architecture that will be investigated more deeply, by building quantitative models of the impact. For carrying out quantitative impact analysis, it is necessary to refine the quality attributes, i.e. decompose them into measurable attributes, which are deemed good indicators. A measurement instrument is the means of turning a measurement, as a description of information and its combination into something that an evaluator can act upon. This translation will take the form of a set of guidelines for choosing appropriate tools. For example, the coupling can be measured subjectively by asking the architect to present the ranking of connector strength among architectural components. A decision

model, in this context, is a means of using the output of the impact quantification models for making decisions about the architecture.

Architecture description languages (ADL) are methods for formal representations of architectures. A number of ADLs have been proposed for modelling architectures such as Wright (Allen & Garlan 1997), Darwin (Magee & Kramer 1996), Rapide (Luckham & Vera 1995), and UniCon (Shaw et al. 1995). ADL development has mostly focused on verification of system functionality and interface matching and for automatic generation of applications whereas in industry functionality and quality attributes have equal importance (Bosch 1998). Only a handful of ADLs support the specification of non-functional properties and architectural refinement and constraint specification have also remained largely unexplored (Medvidovic & Taylor 2000). Recently, there has been considerable interest in using general-purpose object design notations such as Unified Modeling Language (UML) for architectural modeling (Hofmeister et al. 1999b; Egyed & Medvidovic 1999).

As Bosch and Molin (Bosch & Molin 1999), we start the architecture development from the functionality and then use the non-functional requirements for refinement. We also use the iterative approach of going back to refinement after comparing the architecture to the requirements as long as the requirements are met. As in COMPARE (Briand et al. 1998), we analyse both qualitative and quantitative attributes. In addition, we measure the internal capabilities of the architecture in order to get values to the (external) quality attributes. ADLs are not used in this work because their usage seems to require too much work for the purposes of DSP software development. In addition, there have been some doubts about the adequacy of their support for quality requirements. Therefore, mainly UML is applied in describing the architectural views.

2.7 Summary

The related research has contributed both ideas and practical solutions to this work. The summary of the related activities in the architecture development of multimode DSP software is illustrated in Figure 2-4. The main influences on this work have been as follows:

- The architecture description is based on Kruchten's four views (Kruchten 1995) added with a language-language independent development view (Hofmeister et al.1999a).

- The quality attributes create a basis for architecture development. They are concretised by using the taxonomy and quality attribute definition of Barbacci and his colleagues (Barbacci et al. 2000).

- The ideas of reconfiguration by Segal and Frieder (Segal & Frieder 1993) are used for answering to some problems derived from the multimode feature.

- In the architecture refinement, the initial architectural views are modified according to the quality requirements and reconfiguration needs. The used concept of defining the main problems for each view and defining strategies for solving them is similar to that of Hofmeister and her colleagues (Hofmeister et al. 1999a). Architecture refinement utilises the architectural styles and patterns (Buschmann et al. 1996; Bass et al. 1998).

- The architecture evaluation is based on the quality requirements. The ideas of possible evaluation methods and the definition of profiles are inherited from Bosch (Bosch 2000). Scenario-based analysis is based on SAAM (Kazman et al. 1996) and ATAM (Kazman et al. 1998) with additions to how to elicit the scenarios (Lassing et al. 1999). RMA is applied in performance analysis (Klein et al. 1993).

- The architecture development denotes iteration between refinement and evaluation. The design approach has gained ideas from Bosch and Molin (Bosch & Molin 1999), and from the COMPARE approach (Briand et al. 1998).

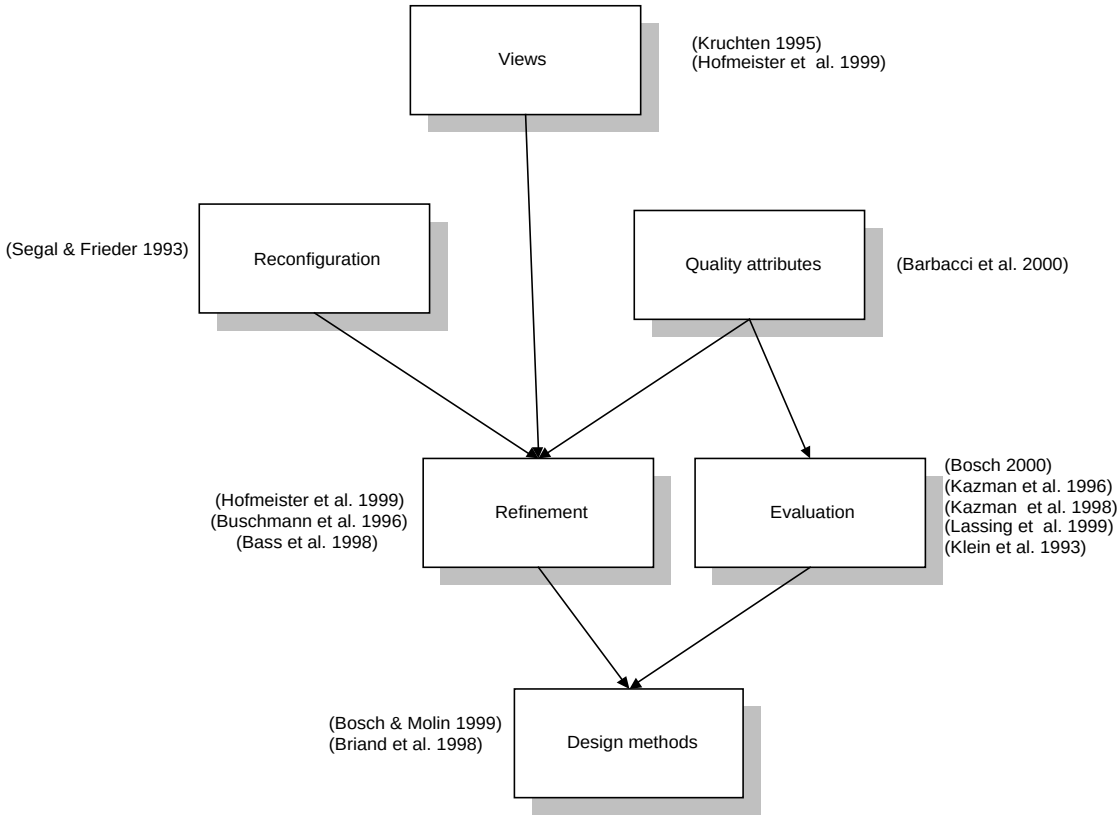


Figure 2-4. Summary of related research.

3. Problem analysis

The purpose of this chapter is to analyse the characteristics and constraints in the development of multimode DSP software. The information has been gathered using product documents, interviews and literature.

3.1 Multimode products

There has been a growing interest to multimode wireless products in the last few years. Firstly, there are already several wireless standards in use, and new standards are being developed. It is not likely that the current systems are removed from service in the near future. Secondly, one network may not be able to fulfil all the requirements of the users. For example, WLAN is mainly directed to be used indoors and in areas where high availability is needed, whereas WCDMA is designed for use in urban areas. So there will be several wireless systems available at the same time. If not any of them is chosen to be the one and only system, as seems to be the case, there will be a need for multimode phones and base stations. When several modes are included in one system, additional control is needed for handling the mode changes (Kukkohovi 1996).

In the multimode systems, there are several questions that should be answered relating to the transparency of the multimode functionality. Should the user know in which network he is camping? Should he know when a handover from one network to another occurs? Is it even possible to make the handover transparent? What are the responsibilities between the network and the handsets? Many of these questions should be defined by the standards but so far only the GSM and WCDMA co-operation has been considered.

The future wireless terminals will not be only multimode but also multimedia terminals. All the media forms will be represented: audio, video, text etc. In addition to new applications, the end user needs will be catered for with more customisation. There will be handsets that have all the possible services but also handsets that have only a minimum capability. The Internet will be the core of the operations. In fact, new services may be downloaded to the terminal when needed, similarly to the downloads to PCs nowadays, even over the air interface.

There are visions that not only applications are downloadable but also the underlying platform including the radio interface functionality (Lu 2000; Srikanteswara et al. 2000; Varshney & Vetter 2000). A fully reconfigurable mobile device could be able to select the wireless network that best meets user requirements.

One possibility in the incorporation of mobile communications, Internet, multimedia, dedicated point-to-point communications, and personal computing is software defined radio (SDR) (Mehta et al. 2001). “SDR is defined as one that implements a specified range of capabilities through elements that are software-reconfigurable” (Mitola 1999). The radio is a software radio when the functions may be redefined in software (e.g. by downloads). Ideal software for a mobile phone would be therefore a multiband multimode radio with dynamic capability defined through software in all layers of the protocol stack, including the physical layer. The requirements for reconfigurable mobile communications can be considered from several perspectives (Drew & Dillinger 2001), viz. end-user, application developer/content provider, service provider, network operator, and terminal and component manufacturer perspectives. As a rule, end-users value, for example, ubiquitous mobile access, low cost and relevant services. On the other hand, terminal and component manufacturers expect that the product creation is fast and the software updates are easy to make.

Portable unit challenges, which are used in the marketing, are usually size, weight, power consumption, and cost. Not only the appliance but also the cost of phone calls is important. Power can be saved by sleep modes, which exploit the low duty cycle of voice and data communications. A large part of handset power is consumed in clock generation and distribution. Multimode handsets in particular must generate several different clock rates for each supported standard. The cost of handsets in volume production can be seen as a nearly linear function of parts count (Mitola 1999). Therefore, handset software has to be extraordinarily efficient in the use of computation resources. In addition to the product requirements, the development has also other goals. For example, right timing of the releases and error-free products are emphasised (Kaikkonen 1996).

Challenges in the base station development are similar to handsets as regards to performance and cost. However, power consumption and resource utilisation are

not as critical. Reliability is important because the network cannot be allowed to be down very long. A thorough testing should make sure that crashes occur extremely seldom. In addition, the life cycle in base stations is longer than in handsets. Therefore, maintenance should be provided at least for ten years. There is also a smaller selection of products in a product family. The amount of channels to be supported is the main influence on the cost of base stations.

3.2 DSP system components

A signal processing system can have three types of components (Figure 3-1): hardware platform, software platform, and application software.

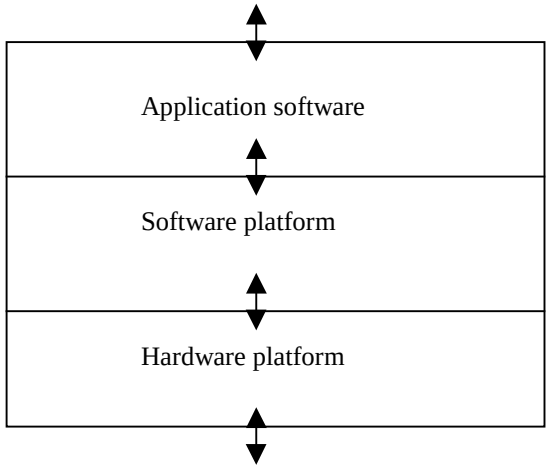


Figure 3-1. DSP system components.

The functionality that the application software has to implement can be divided into signal processing components and control components. Signal processing parts are usually further divided into transmitter and receiver parts. Signal processing functions include, for example, channel coding and interleaving in the transmitter side and respective decoding functions in the receiver side. The receiver is usually much more complex (Mitola 1999). Control includes functions such as communication with other processors, power level control, and high level control of signal processing. The application software also controls the communication with the other parts of the multimode product.

A software platform includes hardware drivers, interrupt servers and operating system if there is one. The evolution of real-time operating systems (RTOS) has had a significant impact on the performance of DSP solutions (Oshana 1998). There are now RTOSes defined specifically for DSPs. DSP RTOSes have features such as several scheduling options from pre-emptive priority-based real-time multitasking to time-slicing, deterministic critical times, and support for direct memory access (DMA).

A hardware platform controls the radio interface. It may be composed of one or more of the following components: digital signal processors, reduced instruction set computing (RISC) processors, application-specific integrated circuits (ASICs), core-based ASICs, analog circuits, field programmable gate arrays (FPGAs), memory, buses, and all sorts of peripherals.

The following features should match the application when selecting a processor: type of CPU, direct memory access, memory access, on-ship memory, and I/O port (Oshana 1998). Digital signal processors come in many varieties. They can be fixed points or floating points. DMA is used in applications demanding high data rate and I/O. Digital signal processors designed to support high data throughput have one or more communication ports to allow a fast transfer of data in and out of the processor. The speed of general-purpose processors has increased to the point that they can be used for running signal-processing applications (Oshana 1998). However, some intensive applications have high enough complexity and speed requirements to justify a development of an application-specific multiprocessor system.

One or more ASICs can be used for the performance-intensive portions of a system. Core-based ASICs are especially common in portable applications. Programmable processor cores are combined with custom datapaths within a single die. FPGAs are fast and can do certain functions very quickly but they are difficult to develop compared to programming digital signal processors.

It is possible to use registers, on-chip memory, or external memory. Most digital signal processors do not have an extensive amount of on-chip memory. However, the speed, power, and cost penalties for using off-chip memory are often prohibitively high for DSP software.

Compared to multi-chip systems the migration of discrete components into a single-chip footprint enables designers to take advantage of the cost, power and speed (Peters 1999). Developers can integrate peripherals, memory, DSP cores, microprocessor cores, application-specific cores, and proprietary third-party cores. A system-on-chip (SoC) attempts to integrate different design technologies into a single chip. Thus, software and hardware development brings about many challenges, along with the selection of tools, training, and methodologies. The development of highly integrated systems has resulted in more emphasis being put on hardware/software co-design.

3.3 DSP software characteristics

The complexity of implementations of software for embedded DSP applications arises from intense time-to-market pressures and tradeoffs between different implementations with respect to area, power, cost and performance (Kalavande & Lee 1993; Bhattacharyya et al. 1999). DSP software is constrained by standards. However, only the transmitter side is usually defined in the standard and the receiver has to be made so that it is able to receive the transmitted signal. In addition, although designers basically have different choices in implementing some functionality, functionalities are nevertheless often designed in a traditional way that has been found reasonable for that problem.

DSP algorithms are typically periodic real-time operations. Latent periods of interrupts should be as short as possible. A response time requirement for each message or message type should be specified. Time consuming algorithms must be written in such a way that they are as fast as possible. Because of stringent timing requirements, some of the algorithms have to be implemented in hardware. Therefore, DSP software can include complex hardware interfaces.

Especially in base stations, functionality can be distributed into several processors. There is a need for scalable software architectures that would enable the increasing of the capacity easily. In handsets, on the other hand, it is important to save power e.g. with power saving modes. However, the saving of the memory – for example, the same memory area is used for several purposes in different modes – is the most important factor when speed is not critical

(Kaikkonen 1996). Consequently, unfortunately, the complexity of the code increases.

The development of DSP software includes many compromises. Which characteristics are emphasised depends on the product's requirements and the needs of the applications that utilise the DSP platform. If speed is optimised, the system may consume less power - often, at the cost of memory. If memory is optimised, it may reduce memory accesses and hence power consumption but generally slows down code. Finally, power-optimised code can be equivalent to optimising speed but may just as well slow down code. In addition, all these optimisations affect the development time. At some point, the time used in optimisations does not pay off. It is also typical of DSP software that the standards are not completed before the software design begins so that the designed components should be made adaptive in order to prepare for changes in requirements.

3.4 Development characteristics

This chapter describes the position of DSP software design in the DSP system development and the current practices in the DSP software development. It also discusses the current status of the reuse of artefacts.

3.4.1 Development process

The issues in the design of signal processing systems range from the partitioning of algorithms between hardware and software to the selection of the type and number of processors, selection of the interconnection network, synthesis of the software, and synthesis of custom hardware. DSP system development has usually the following phases (Figure 3-2): algorithm design, software/hardware partitioning, hardware and software implementation and test, and integration and test of the combined functionality (Kalavade & Lee 1993; Vihavainen & Marttila 1998).

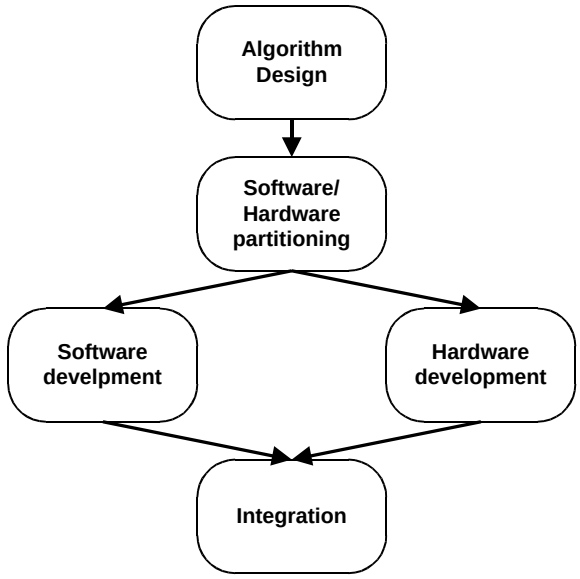


Figure 3-2. DSP system design phases.

Algorithm developers think about the problem to be solved on an abstract level. Their concern is to make sure that the algorithm fulfils the specifications but they are not necessarily interested in the implementation. This is the phase, which corresponds to the requirements analysis in other software development. Algorithm development usually includes simulations with special tools and the performing of complexity estimates. This phase gives an ideal goal to the performance that the system should have.

In the next phase, different hardware platforms are studied and functionality is divided between selected platforms in accordance with the complexity estimates. Usually, an experienced hardware system designer is responsible for this. In this phase one can also first carry out a feasibility analysis in order to analyse several options for hardware platform at a high level as well as to analyse the requirements further. After this, software development and hardware development continue in parallel. However, the hardware/software interface specification usually requires participation from both parties. In the integration phase, the results of the hardware and software development are combined and tested.

DSP software development has traditionally comprised functional specification, software architecture design, software design, software implementation, simulation, and emulation. The case is often such that software requirements specification has not been written; instead, the functional specification has partly compensated it with clarifying the functionality to be developed. The functional specification has been based on higher-level system documents and standards.

In the software architecture specification, the system has been decomposed into processes, and the interfaces between components and to the hardware are clarified. Usually this denotes some sort of block diagram. Interfaces are defined with message sequence charts (MSC) or equivalent. In handset development, this phase has often been left out completely (Kaikkonen 1996). In simpler and more resource-restricted systems, the architecture has traditionally been based on the main-loop and subroutines style and no operating system or tasks have been used.

The process of coding algorithms is usually iterative. If the implementation does not fulfil the performance requirements, they have to be optimised. The implemented units are first tested in a simulator environment. Although tools have become better, most of the errors are still found in integration tests, because unit tests and other intermediate tests are not kept up-to-date (Kaikkonen 1996). Cycle accurate simulators are needed in order to obtain execution estimates. Emulation is needed for timing related problems and parallel processing applications.

Traditionally, the hardware platform has been fixed first, after which software has been adapted to it (Vihavainen & Marttila 1998). HW/SW co-simulation is not yet very common because hardware simulation, in particular, is quite slow. Therefore, usually the developed hardware and software units are first integrated and tested in an environment that may include emulator and FPGAs. Often the software developers also carry out the integration testing (Kaikkonen 1996).

3.4.2 Development in practice

Time-to-market is very important especially in the handset development. Because of the limited human resources, the importance of an overall

development effort has grown. There is an increasing shortage of capable new employees. However, usually the cost of the development has not been a problem.

DSP algorithms have until recently been developed in assembly. When processors and compilers have become more efficient, C has been taken into use instead. This is because assembly is not portable, it increases time-to-market, it is harder to maintain and harder to write (Oshana 1998). However, in low-power embedded products using fixed-point processors code is still written extensively in processor-specific assembly (Kaikkonen 1996). Furthermore, although code were written mainly in C, some critical functions may have to be made in assembly. Today, DSP compilers still produce significant overhead with respect to assembly code written and optimised by hand (Zitzler et al. 2000).

Usually the product schedules are so tight that as many design phases as possible are done concurrently. This may sometimes lead to situations in which such decisions have to be made, at least temporarily, that are not based on facts but more or less on educated guesses. Later, when all the facts are known, the decisions may have to be changed. The other thing that causes uncertainty is that the standards that the products should be based on are not stable and may change during the development.

The embedded software development is also influenced by the hardware that it should interface with. The complexity of the software may depend on the complexity of the hardware. There is always a tradeoff in the hardware/software interface on what functionality should be put to what part and whether the increase of hardware complexity is afforded in order to get a more efficient software implementation. Often, software is considered to be free. Consequently, whenever new features are required or hardware seems to become too costly, the responsibilities of software are increased.

DSP software tools have not been on the same level as in other embedded software development. One thing that makes the situation harder is the close operation with the hardware. DSP code is usually difficult to test with simulators. Therefore, the software components and hardware have often been integrated and simultaneously tested in an emulator environment (Kaikkonen 1996).

3.4.3 Reuse of artefacts

Building sets of related systems together can yield improvements in productivity, time to market, product quality, and customer satisfaction (Bosch 1998; Bass et al. 1998; Bass et al. 2000). These systems create product families. Many mechanisms are commonly used to achieve reuse and sharing in product families. The most common way is to manipulate source code through conditional compilation and source code configuration management (Karhinen et al. 1997).

In DSP software, the lack of standard coding guidelines combined with scarce processing power has often resulted in an unclear software structure (Huotari 1999). This has caused problems for reusability. In our definition, real reuse denotes the possibility of choosing a code component without testing it and trusting that it does what it promises. Although these kinds of reusable components may be found in DSP software, there is an additional problem that the components may never be frozen. The algorithms will keep on evolving. Bit exact functions that are not optimised with assembly are the best candidates for reuse. For example, most channel coding functions may be reusable because they are bit exactly described by the 3GPP WCDMA standard. Another problem for reusability is that new standards are not exactly stable themselves and after they are fixed, modifications may be caused by the customer preferences especially in the base station development.

Standardisation creates possibilities for using COTS components. However, this is not without problems. For example, while many cellular phones incorporate software components licensed from third parties, their definition is ad-hoc, their architecture unsophisticated, and their role in a system totally static (Lee 1999).

The industry still shows indifference to software architecture benefits (Maccari & Saridakis 1999). They often generally underestimate the software architecture or they do not have enough knowledge to use it in a right manner. They are trying to evolve their old systems with minimum modifications. Finally, they do not have clearly defined system requirements and often the strictest requirement is the deadline for product delivery.

Products in a product line share a common architecture that is used to structure the components of which the products are built. The architecture and components are central to the set of core assets used to construct and evolve the products in the product line. In addition to components, architecture supports the reuse of other assets in the product line. They include personnel, defect elimination, project planning, performance issues, processes, methods and tools, and exemplar systems (Bass et al. 1998). One of the problems in managing product lines is their evolution. It is not realistic to expect that assets could be designed taking into account all thinkable forms of variability, but assets should be designed so that the introduction of new variability requires minimal effort (Bosch 1998).

3.5 Summary

Until recently, DSP software design has concentrated on optimising the functionality of the algorithms. In the emergence of new technologies and need for new types of products, the current DSP software development has the following characteristics and challenges:

- Multimode operation and new applications create more complicated products and more control functionality in addition to the traditional signal processing features.

- DSP software should fulfil the requirements imposed by standards and designed into algorithms by algorithm experts. On the other hand, it should be done within the boundaries defined by the hardware such as processing time, memory usage, and power usage restrictions.

- Time-to-market pressures necessitates the starting of several design activities in parallel - due to this, requirements may still be unclear at the moment of setting about a design task.

- Hardware and software designs are done concurrently. The hardware/software partition is difficult to make. In addition, there is a need for getting hardware interfaces that are more easily controlled by software. However, this may cost in hardware.

- Software development is dictated by hardware architecture. Often, hardware architecture is fixed before software design even starts. In addition, it is frequently the case when reacting to the changing requirements that software is considered free, with only hardware cost counting.

- The overall system functionality is difficult to manage and understand because of rigorous optimisation. Often, only the original developer is able to make changes to software modules. This causes problems in the training of new people, in maintenance, and for reuse between different products.

- In order to shorten the time-to-market, there is a need to increase reuse. Reuse between similar products creating product lines, in particular, but also reuse from earlier products should be facilitated. Because high-level languages are starting to be used in the development, this is now possible.

- Although control functionality increases, the main problem remains the same: how to implement the signal processing functionality efficiently?

4. Quality attributes

The purpose of this chapter is to study the DSP system qualities from the software architecture point of view. Quality attributes are the common goals that different interest groups have for the system. Usually in a software development project, some of them have been implicitly taken into account but they are not written down anywhere. The quality attribute taxonomies have been earlier published in (Purhonen 2001).

4.1 Introduction

There is a difference between the academic view of the software architecture and the industrial view (Bosch 1998; Maccari & Saridakis 1999). Whereas academic research has concentrated more on finding methods for validating the functionality, the industry struggles with ensuring other factors of their products such as performance.

The requirements that affect the DSP system's architecture come from three sources: standards, customers and end-users, and developing organisation. The primary quality goals for a DSP system are classified in Figure 4-1 based on the analysis of the previous chapter. The focus should be on the most important quality goals because the biggest gain can be found there. This is only a list of the primary goals; the weight of each quality should be specified for each product individually. Consequently, the optimisation of the architecture can be prioritised further.

Standards are the main source of performance and functionality requirements for a DSP system. Functionality can also be derived from customer requirements, and a development organisation can add some functions, for example, for testability reasons. Customers and end-users value the low cost and usability. Consequently, especially in the handsets, the weight and size of the system and power consumption are important issues. A development organisation can have both short-term and long-term goals. A short-term goal is to get the product released as early as possible and a long-term goal is to also make it modifiable.

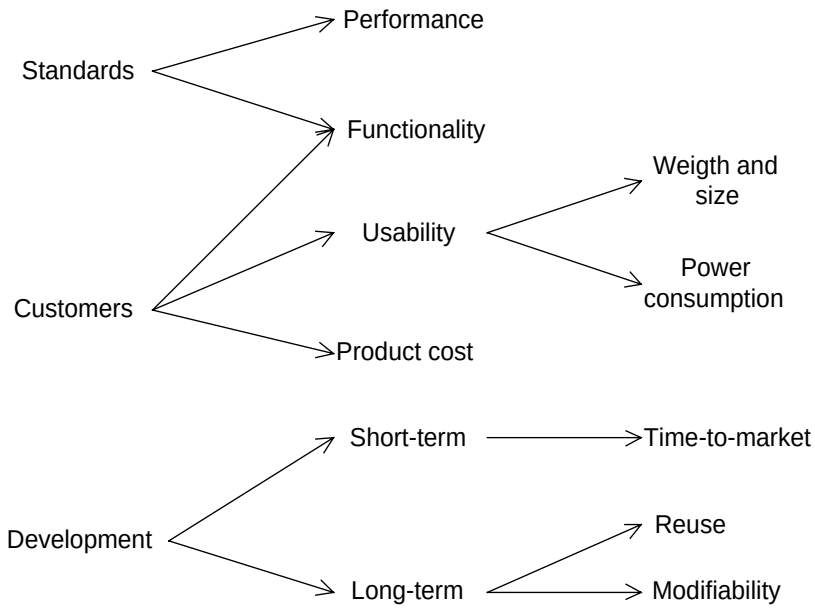


Figure 4-1. DSP system quality goals classified by their sources.

The non-functional quality attributes are further studied in the following chapters. The qualities are classified under three titles: performance, cost, and variability (see Figure 4-2). Performance handles all the resource-related quality goals. Hardware cost is relevant to performance in measuring resource utilisation. Cost concerns all the money-related goals. Variability is for studying the architecture's ability to endure change. Variability includes a modifiability goal of one product but it also measures how well the architecture could be applied to another similar product. The degrees of variability cover the cases where the architecture cannot be used for another product at all, where significant or minor modifications have to be done, or where the architecture is valid as such.

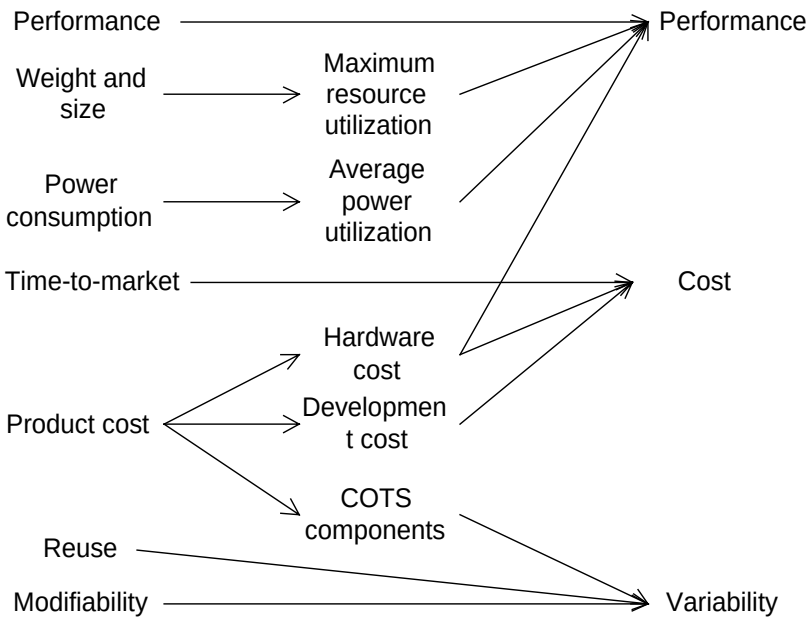


Figure 4-2. From quality goals to quality taxonomies.

The quality attributes are concretised with taxonomies. They are used for getting a better understanding of the quality goals; how they can be measured or how architectural choices affect them. Additionally, taxonomies offer a rationale for asking elicitation questions in the evaluation phase and a base for tradeoff studies. The taxonomies have the following branches (Barbacci et al. 2000):

- *Response* section describes the measurable or observable quantities that characterise the quality.

- *Stimulus* can be divided into two types of events: some cause the execution of the operations of the system, some require operations on the system.

- *Architectural parameters* are the decisions that are made during the software architecture design. Architectural parameters are the internal factors that affect the way the system reacts to stimulus.

A fourth branch not suggested by Barbacci et al. consists of external factors. Like architectural parameters, external factors affect the qualities but they are not included in the taxonomies because they are outside the scope of the software architecture design. However, they have equal importance in achieving the qualities and it can be said that the whole embedded system design is a tradeoff between all the factors. For example, performance measurements may have to include the time spent in hardware calculations and the cost measurements should count the cost of hardware components although they cannot be changed by software architecture. External factors are characterised in Figure 4-3.

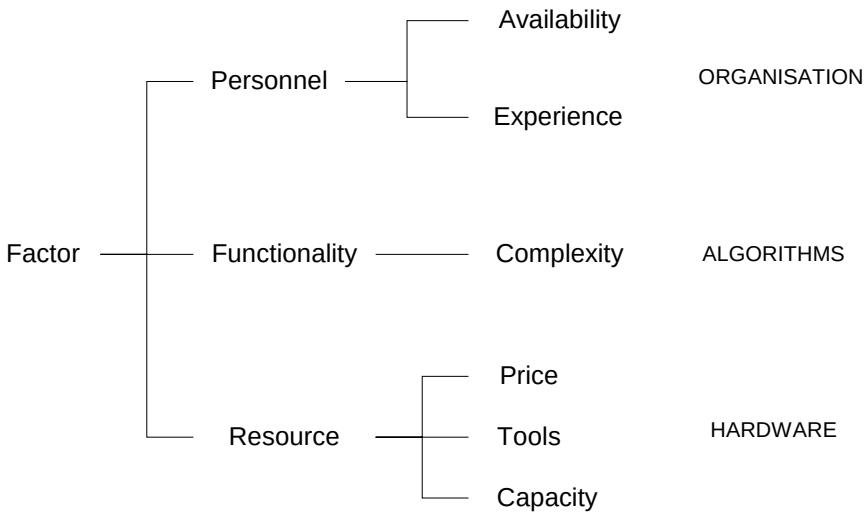


Figure 4-3. External factors.

The software architecture design is constrained by the organisation, algorithm designs and hardware designs. Because it is difficult to get trained personnel nowadays, the importance of finding right people for development projects has grown. In a DSP system, the complexity of the algorithms is dependent on how good a result is required from it. Thus, in some cases, the lowering of the output requirements may simplify the implementation significantly. During software architecture iteration, the hardware characteristics as well as the other factors should be fixed. However, after the software architecture is optimised, the other factors can be changed thus causing further iterations of the software

architecture. On the one hand, hardware gives a limit to the maximum available resources and on the other hand, it gives parameters in achieving the performance goals, e.g. in the form of execution time estimates.

The taxonomies are used as a base for questions when one is trying to attain an understanding of the system requirements. In each chapter, there is a table of example questions, which can be asked in interviews during the requirement study. In addition, the chapters include examples of what the answers could include.

4.2 Performance

In DSP systems, the performance often denotes a presentation of how well the signal that is produced by the algorithms fulfils the standards. In this context, however, we are not considering signal-interference-ratios and such; instead, we consider how well our implementation of the algorithms fulfils their specification and more precisely, how the architecture affects the possible outcome of the implementation work. Performance would not be a problem in a perfect world. There would always be enough memory, enough processing power etc. However, in our case there usually is not. Architecture can reduce the performance by causing features such as indirection and additional communication. Architecture can facilitate reaching performance requirements with better understanding of the functionality of the system, reuse of optimised components etc.

The performance taxonomy is presented in Figure 4-4. Performance analysis is a mature domain and therefore the existing taxonomies and research has been extensively utilised in creating this taxonomy (Smith & Williams 1993; Barbacci et al. 1995; Bellay et al. 1997; Klein & Kazman 1999; Barbacci et al. 2000).

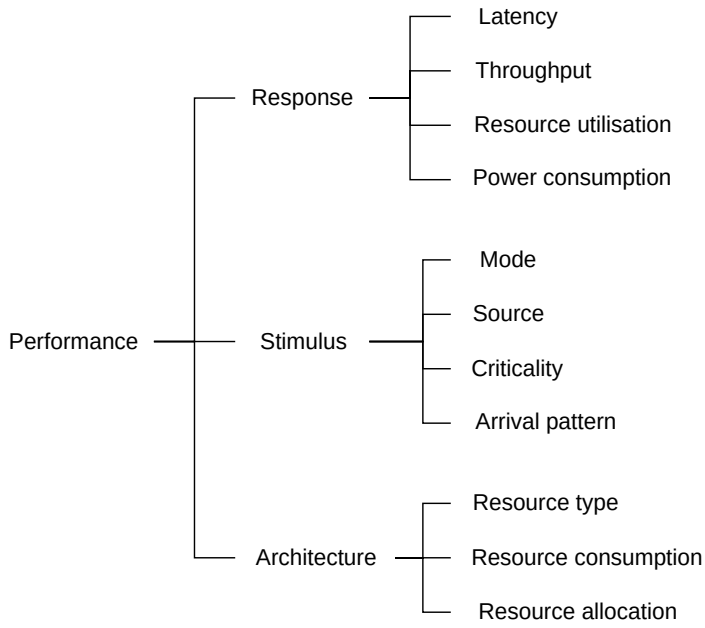


Figure 4-4. Performance Taxonomy.

The *response* section contains the measurable or observable quantities that are used for assessing the performance. Response section contains latency, throughput, resource utilisation, and power consumption. They are further classified in Figure 4-5.

- Latency is a measure for how long it takes to respond to a specific event. The worst, best and average values for latency can be calculated, although the worst case is usually the most interesting. Jitter is a measure of the variation of the time at which a computed result is available from cycle to cycle.

- Throughput refers to the number of events that can be responded to over a given interval of time. The best, average and worst case measurements can be made from it.

- Resource utilisation is measured because the software has to work with the resources that are given by the hardware platform. The software decisions have an effect on what kind of hardware is needed but the hardware brings

about restrictions on what kind of software can be developed. Especially in the hardware/software interface, a detailed tradeoff should be made on what is done in software and what is done in hardware. Obviously, the software side hopes that the data it receives from hardware is in a form that can be easily used in software. However, this may mean an additional amount of die area that is not acceptable. In addition to tradeoffs between software and hardware, there are also tradeoffs between different kinds of hardware. For example, minimisation of processor utilisation can increase memory utilisation.

- Power consumption is separated from other resource utilisation because in the resource utilisation side the equivalent measurement would be the price of a battery, and therefore, the maximum usage of power. Instead, power consumption is measured using the average utilisation of power.

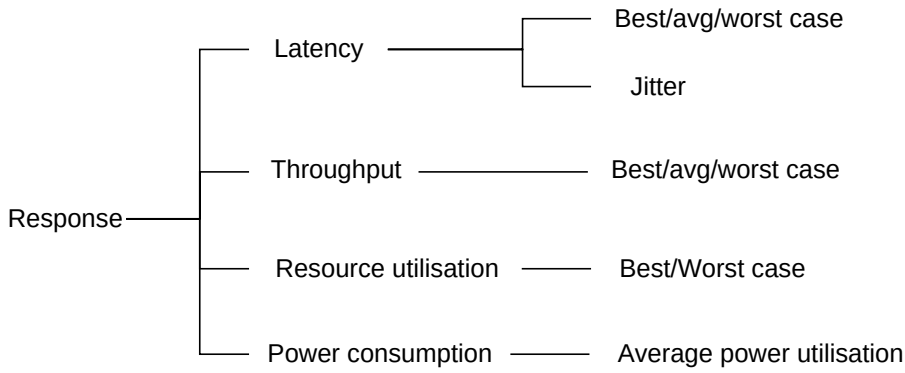


Figure 4-5. Performance taxonomy: Response.

Stimulus describes the events that cause the architecture to respond or change. Systems can have different requirements for different modes. In addition to the normal mode, a system can have, for example, a testing mode. Source specifies the different types of sources of events: internal, external or clock interrupt. Internal events come from hardware to software and external events are commands to the DSP system from other subsystems of the product. Arrival pattern denotes the frequency of the events and it can be periodic or aperiodic. When it is aperiodic, it can be further specified as sporadic (minimum inter-

arrival time) or random. Important information is also gained by the criticality of the events, which can be used in prioritising the handling of the events.

Architectural parameters define how the architectural decisions affect the response of the system. Figure 4-6 presents details of the architecture section. Performance depends on the resources and their allocation. Resource types include CPU, memory, bus, dedicated hardware, and battery. Respectively, resource consumption can be expressed as execution time, memory size, bandwidth, silicon area, and power consumption. The resource allocation policy has a primary influence on the performance of a system. For example, the worst case latency depends on how CPU time is allocated.

Resource allocation deals with policies and procedures for deciding between competing requests for a particular resource.

- Scheduling policies. There are various aspects in scheduling that should be considered. Firstly, scheduling can be constructed off-line or on-line. Off-line scheduling usually denotes a cyclic executive approach. In on-line scheduling there are several algorithms that can be used, but before that, one should define whether dynamic or fixed priorities are used. Additionally, one aspect of scheduling is whether pre-emption is allowed or not.

- Queuing policies include decisions upon such issues as the queuing order and whether there should be one queue for each resource. Mutual exclusion methods may be needed when resources are shared.

- The communication method depends on whether the communication takes place in the same hardware node or between separate hardware nodes. In the same node, information can be exchanged through memory, with shared memory, global variables or parameter passing. Methods to exchange information between separate nodes include techniques such as interrupts, polling, and DMA. The communication can be classified as synchronous or asynchronous or as direct or indirect. In addition, the communication protocol should be defined.

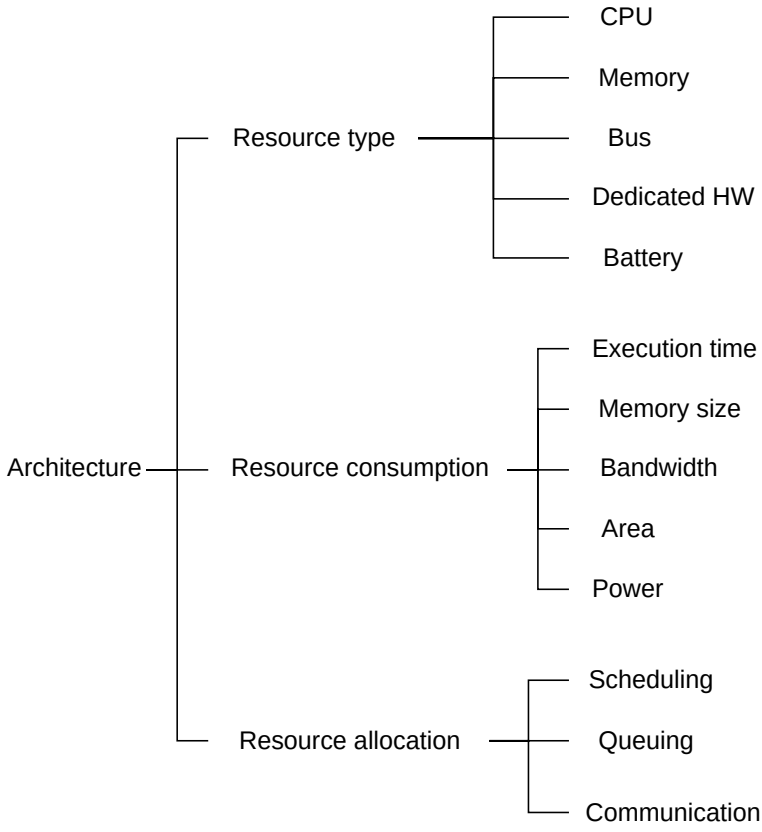


Figure 4-6. Performance Taxonomy: Architecture parameters.

Table 4-1 shows examples of the questions that can be presented for clarifying the performance requirements for a DSP system. The questions can be answered by algorithm designers, hardware experts, business experts etc. These questions are from different phases of development. For example, execution time estimates cannot be given before the complexity estimates of the algorithms exist and the implementation platform has been defined.

Table 4-1. Performance questions.

Question	**Example answer**
What are the modes of the system?	The modes of operation are defined in a state model in page xx in the requirements specification.
What are the services that are requested from DSP system i.e. what are the external events?	The external events for DSP are defined in table xx in the requirements specification.
What are the hardware interfaces, i.e. what are the internal events?	The hardware interfaces are defined in the hardware interface specifications x, y, and z.
What is the criticality of the events?	The criticality of the external events is defined in table yy in the requirements specification. The criticality of internal events is defined in table zz of the hardware interface specification x.
What is the execution time estimate for algorithm x?	Algorithm x is mapped to processor y. Therefore, its execution time estimate is z.

4.3 Cost

In this context, cost refers to what can be earned with the product. The customer is usually interested in the price of the product, whereas the manufacturer thinks about what is left when the production and development costs are taken out of it. In addition, the earlier the product can be released, the more can be earned by it.

Architecture can help in reducing the development time in making the system more understandable. It can be used when different interest groups discuss the features of the system, so that the most difficult problems are found early in the development cycle. It can be used for training the new developers to understand the overall system. Architecture also helps in reducing the development time in dividing the system into more manageable units. In addition, being a base for

further development, architecture can be used for organising the development unit and assigning duties to developers. Architecture is a common tool for all the phases of the system development - especially, integration.

Defining software cost is a complex problem and a research subject in itself (Withey 1996; Briand et al. 2000; Boehm et al. 2000; Strike et al. 2001). However, the goal at this point is not to give precise values to software cost but to see how architecture can affect diverse business goals. This study is useful especially if such features can be found that are suspected to be sources of undue risk and, potentially, unbound cost. Characterisation of product and development cost qualities is presented in the cost taxonomy, Figure 4-7.

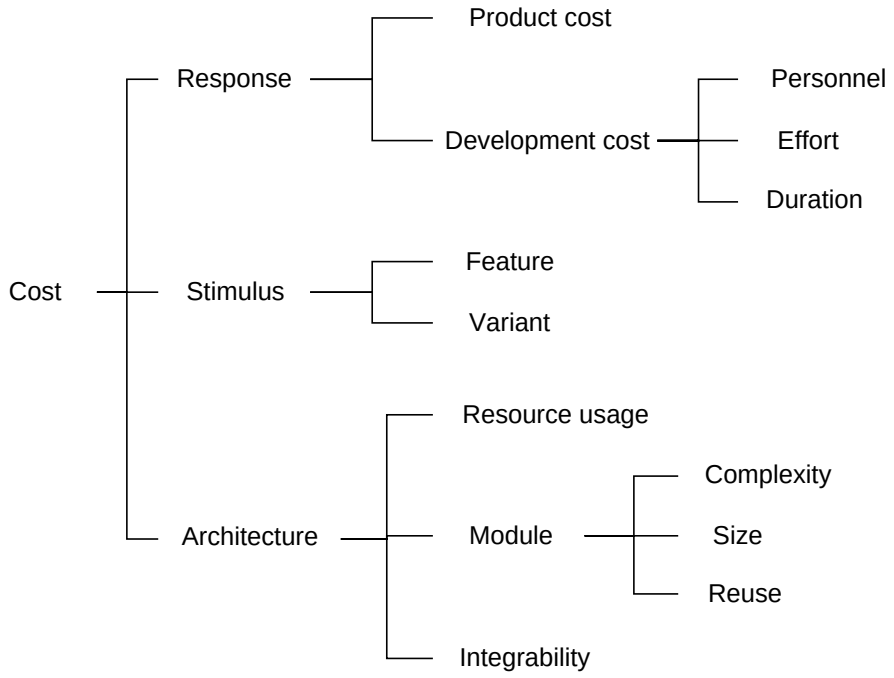


Figure 4-7. Cost taxonomy.

Response is divided into product and development cost measures. Product cost consists of the cost of the used resources, i.e., hardware cost. Development cost is divided into three measures (Vigder & Kark 1994): the amount of personnel that is needed, the effort that has to be put to the development, and the estimated

time-to-market i.e. duration. The cost of the tools, environments etc. is not included in this study.

Stimulus to the cost quality consists of the features that are required from the product. Often there are several variants of a product with different feature sets. The architecture should support all these feature sets. Cost gives constraints to what kinds of features can be included in the product. For example, the need of the processing capacity of some feature can be so huge that it cannot be implemented. In this case, the feature has to be modified or left out of the system or the processing capacity has to be increased.

Architectural parameters are divided into resource usage, modules, and integrability. The maximum resource usage defines, e.g. how efficient the processor should be. In this way, cost is dependent on performance quality. In addition, the cost depends on what type of processor is selected i.e. general-purpose processor, DSP or some other. The release date can be calculated from the time spent to develop individual modules and the integration time. Module development time depends on the complexity and the size of the modules, and reuse possibility. The module development phase can be shortened with enforcing concurrent development. On the other hand, concurrent development is supported by modularity. Integration time is affected by how integrable the architecture is. Integrability means the ability to make separately developed components work correctly together (Bass et al. 1998). Integration time will be diminished if the architecture is integrable. In addition, the work is easier to divide between developers, the system is more understandable, and incremental builds are enabled. When architecture supports integrability and modularity, it should have:

- well and completely defined component interfaces,
- well and completely defined component interaction mechanisms and protocols, and
- clean partition of responsibilities to components.

The architecture can affect the development time with supporting the component reuse from previous products or facilitating the use of COTS components. Using patterns in solving the architectural problems increases reuse potential.

Table 4-2 shows examples of the questions that can be presented for clarifying the cost requirements for a DSP system.

Table 4-2. Cost questions.

Question	**Example answer**
What is the goal for release date?	Release date 1 is x1.y1.z1. Release date 2 is x2.y2.z3
What is the estimated development time of algorithm x?	The estimated development time of algorithm x is y.
What is the goal to the hardware cost?	Cost of the hardware is estimated to be x.
How resource usage is prioritised?	Processor utilisation should be minimised even at the cost of memory usage.

4.4 Variability

A variability study includes the issues of how well the architecture adapts to different types of changes and whether the same architecture can be used in several similar products. Variable architecture should be able to cope with new product features with minimal or no changes. Variability is a measure for studying if the developed architecture could be taken as a base architecture to a product line.

Reuse became an important goal also in DSP software with the possibility of using higher level languages instead of assembly. New products are needed in increasing pace; also, manufactures can no longer afford to develop each product from scratch. There are needs to achieve productivity gains, improve time-to-market, maintain market presence, compensate for an inability to hire etc. A

common architecture provides a base for developing a common set of assets. It is used to structure components from which the products are built.

In this work, the focus is on the development of one reconfigurable product. However, we want to prepare for future changes and therefore have similar goals as in the product line architecture development. A reconfigurable multimode product could be actually thought of as several products combined in one product. Multimode DSP software architecture should facilitate reuse and modifiability. In reuse, there are several viewpoints. Firstly, in a multimode system, the architecture should facilitate reuse between different modes. Secondly, the architecture should take into account the reuse of existing components and encourage the developing of reusable components. The modification of the product during its lifetime should be anticipated.

The variability taxonomy is depicted in Figure 4-8. Several sources have been utilised in its development (Briand et al. 1998; Lassing et al. 1999; Barbacci et al. 2000).

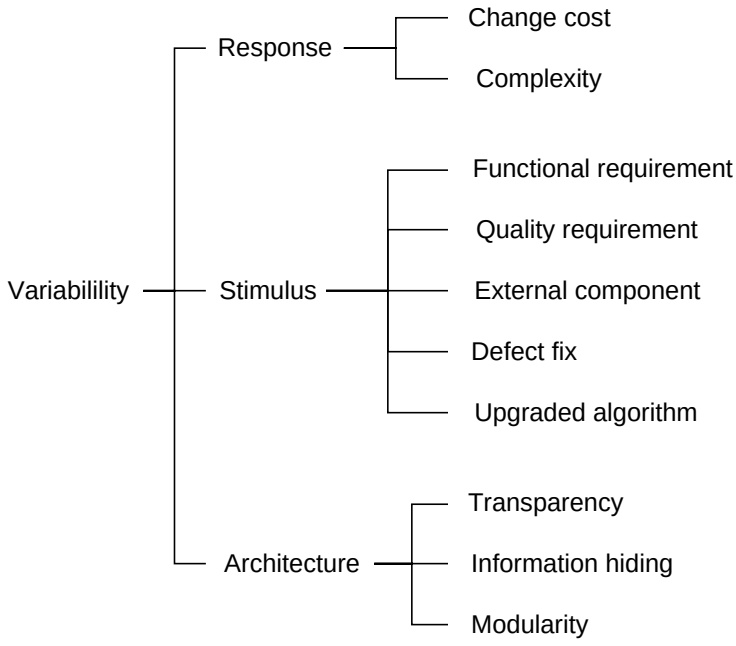


Figure 4-8. Variability Taxonomy.

The *response* of the system or development organisation to changes is the effort that is needed to make the change and the complexity increase that results when the architecture has to be changed. As with the quality attribute cost, the change cost can be classified with the following terms: product cost, personnel, effort, duration. One example of the measures of complexity is the ratio of the number of the interfaces to the number of components.

Stimulus is composed of various kinds of changes the architecture should adapt to. Requirement changes come from the new or improved standards, customer requirements, and development organisation. External components are COTS components or internal reusable assets or hardware components. External component change can mean a new COTS product that is taken into use, an improved version of a COTS product or other reusable asset, a new version of a processor etc. Most often, perhaps, the change is a defect fix. Algorithms can be further developed during the architecture development, so they may also affect the architecture.

The *architectural parameters* that support this kind of variability are:

- Transparency. The ability to move functionality within the system without affecting other functions. The transparencies that are needed for supporting variability include, at least, configuration and scaling transparencies (Anthony 2001). Applications and services should function independently of the way in which the system is configured. In addition, it should be possible to scale-up parts of the system without changing the system structure or algorithms.

- Information hiding, abstraction. Buschmann and his colleagues define information hiding as “concealing the details of a component’s implementation from its clients, to handle system complexity better and to minimise coupling between components” (Buschmann et al. 1996).

- Modularity. Modularisation means the meaningful decomposition of a software system and its grouping into subsystems and components (Buschmann et al. 1996).

In order to be able to respond to the above-mentioned changes, the architecture should include the following characteristics:

- Portability is needed for preparing software for hardware modifications. The effect of hardware modification on software should be minimised. This can be accomplished, e.g. by restricting the communication with the hardware to few components or by using a hardware interface layer between the hardware and application software.

- Scalability is needed for a multimode terminal in order to be able to add or remove modes, channels etc. Additionally, the modification of the set of applications may require that the architecture can flexibly adapt to rich or modest feature sets.

- Extensibility ensures that the new requirements that emerge during the lifetime of the product can be added to the system. On the other hand, it should be possible to remove features that are not needed any more.

- Modifiability is important as standards evolve and new and better solutions to old problems are developed. A modifiability goal in DSP software means that new versions of algorithms can be changed to the system without affecting the architecture. Modifiable architecture is modularised with loosely coupled well-defined components.

- Reusability, here, means not only that it is possible to reuse components from older products but that the components developed for this product should be made reusable for the future products. Component reuse is based on excellent definitions of components and on the reliability of their implementation. In order to reuse a component, you should know its restrictions to the full and you need to trust that it is implemented correctly and according to its definition. If you are in any doubt, you should be able to retest the component with the original test plan. On the other hand, developing reusable components takes extra time and the saving can be seen only in the future products.

Table 4-3 shows examples of the questions that can be presented for clarifying the variability requirements for a DSP system.

Table 4-3. Variability questions.

Question	Example answer
What are the anticipated product types that the architecture should support?	Product types are specified in the requirements specification in page x.
What kinds of changes are expected to standards and customer requirements?	The presumed changes to standards and customer requirements are presented in document x.
What kinds of changes in the external components might affect also DSP software?	If components x and y change then component z in DSP software should be modified.
What kinds of changes are the most likely ones?	The change types in the order of their likelihood are presented in document y.

4.5 Tradeoff

The analysis of individual attributes is only a start in the architecture design. Quality attributes may contradict as well as complement each other. Therefore, you need to consider the interdependencies and tradeoffs that exist between them and to define a preference of one requirement against another in case of conflict. The classification of quality attributes based on their relative importance is part of the tradeoff analysis. In this chapter, a very general study of dependencies is performed in order to point out possible sources of quality conflicts.

There are conflicts already inside the performance taxonomy. The goal in performance is to maximise throughput and minimise latencies, but usually this cannot be done without sacrificing resource utilisation. Furthermore, minimising power consumption using power saving routines may cause additional delays.

Variability goals are often not favourable to performance. Portability can affect the performance by introducing indirection, and consequently, overhead. Architecture that is suitable for a large-scale product may not be optimal enough

for a small-scale product. New features may decrease the optimal performance of the overall system. An optimal solution for several products may not be the most optimal solution for the one product that is under development. Modularisation that is needed for modifiability means indirection, but modifiability also allows the most optimal solution of a module to be taken into use when one is available.

From the viewpoint of performance, loosely coupled components can affect the performance by introducing indirect calls. Components that are reused from older products may not be optimised for the current product. However, these are the favourable goals for cost and variability attributes. Product cost restricts the choice of hardware; therefore, it also restricts the maximum performance that can be achieved. On the other hand, rigorous optimisation of performance may need some additional development time.

Variability and time-to-market are also often in conflict. Designing reusable software adds to the development time of the current product. However, reusability usually decreases future costs and makes the system more understandable.

Table 4-4 summarises the possible impacts the quality attribute measures can have on each other. Positive and negative signs mean respective correlation. In some cases, there can be both positive and negative consequences.

One of the things that should be considered during tradeoff analysis is the relative importance of the quality goals. In Table 4-5, the importance of the attributes is studied in a very general manner. In the actual product development, the importances should be studied on a more detailed level. For example, an importance factor can be attached to each scenario that is used for analysing the architecture. Technology roadmaps are used as a source for importance factors in addition to stakeholder interviews.

Table 4-4. Quality attribute tradeoff.

	Latency	Resource utilisation	Power consumption	Product cost	Development cost	Change cost	Complexity
Throughput	+	-	-	-	-	-	-
Latency		-	+-	-	-	-	-
Resource utilisation			+-	+	-	-	-
Power consumption				+	-	-	-
Product cost					-	-	-
Development cost						-	-
Change cost							-

Table 4-5. Quality attribute importance.

Quality attribute	**Current importance**	**Trend**
Performance	High	Lower; with hardware that is more efficient, less work needs to be done for optimisation.
Cost	High	Higher; with increasing competition, cost always matters.
Variability	Low	Higher; it is not economical to design every product from scratch.

5. Architecture Development

5.1 Multimode DSP system design

The DSP software architecture design flow is based on the assumption that the multimode DSP system design is divided into the tasks defined in Figure 5-1. Emphasis is given on the fact that the required functionality of the whole system needs to be analysed before the implementation aspects are considered - as a consequence, the three main tasks of system architecture design will be of equal importance. Traditionally, only the algorithms are considered in the initial phases of the development. Whereas in this context, the first task is to define all the functionality that the system should implement. It is assumed that the requirements specification is used as an input for this work. After the functional specification, the work continues in parallel with software and hardware architecture designs and algorithm design. In the third phase, the results of the separate work are integrated when the quality of the system design is evaluated. The process of the system architecture design is iterative and incremental.

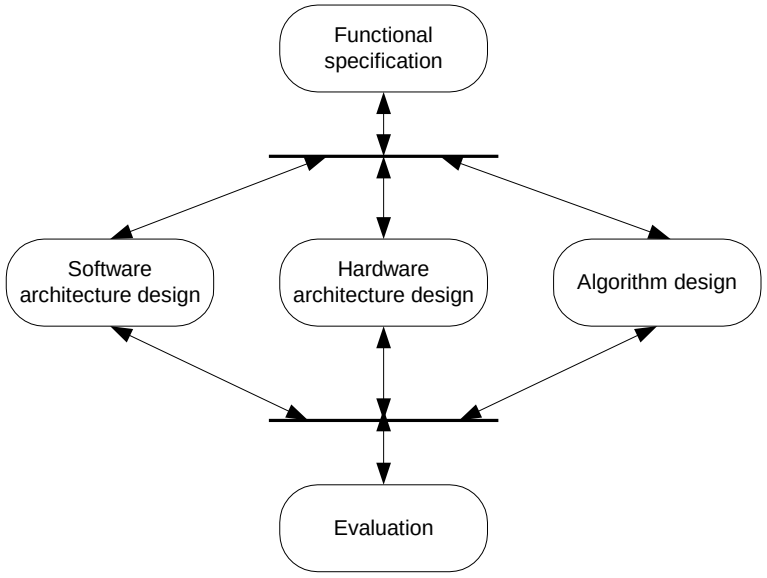


Figure 5-1. System architecture development.

The software and hardware architecture designs and algorithm design are done concurrently but they are also dependent on each other. Algorithm design completes the work in the functional specification ensuring that the signal quality produced by the algorithms fulfils the requirements. This work should be positioned slightly ahead of the other two tasks because the complexity of the functionality affects architectural decisions. In the early phase, the hardware/software partition should be made in co-operation with the hardware and software architects. After the coarse-grained partition of functionality between platforms, there will be finer-grained details such as defining which party should be the more intelligent part in the interface. This work is based on the system quality attributes causing tradeoff between software complexity and hardware resource usage. System architects can iterate the complexity of the system with algorithm designers. In some cases, the quality goals can be easier to fulfil if it is possible to lower the complexity of an algorithm.

The summary of the main goals in different design tasks is illustrated in Figure 5-2. The goal of the system design is to optimise system quality. It is achieved with relaxing from the individual goals of different design phases. As the role of the software is often the clue between problem (algorithms) and solution (HW) domains, the software quality goals are usually derived directly from the system qualities.

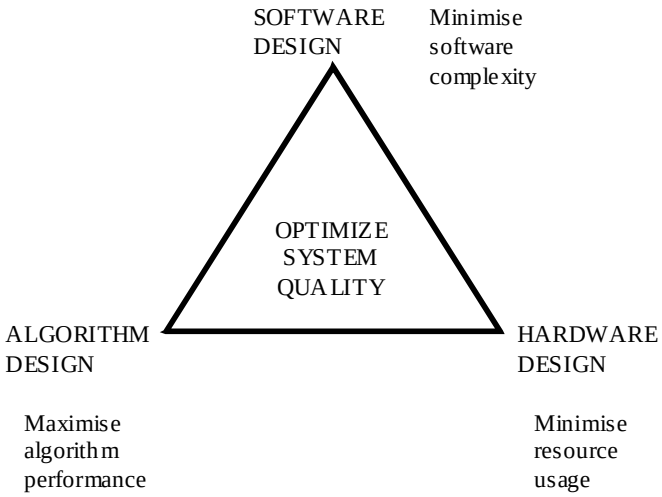


Figure 5-2. System quality optimisation.

5.2 Software architecture design flow

Software architecture is usually described with several views. Our approach is based on four views: logical, physical, process and development views. The software architecture views are illustrated in Figure 5-3. The logical view describes the required functionality in an implementation-independent way; the development view shows how the required functionality can be developed with today's software platforms and technology; the physical view maps the required functionality to the hardware platform; the process view shows the runtime functionality of the system. The physical and process views together create a runtime or execution view to the architecture. The physical view is mainly a summary of the hardware architecture design from the software point of view.

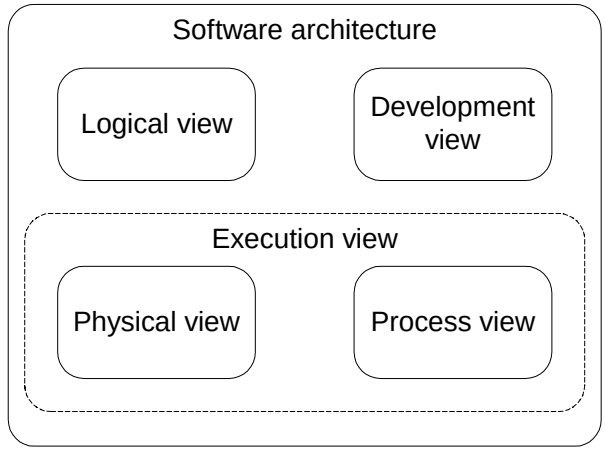


Figure 5-3. Software architecture views.

The logical view is represented by the functional specification in the system design. The physical view is derived from the results of the hardware architecture design. The process and development view designs have fewer dependencies on the other tasks in the DSP system development.

The process of developing the views is iterative and incremental. First, a conceptual version of the architecture is created. It defines the main concepts in the architecture without considering the implementation level details. When the main concepts are clear, the architecture is extended towards realisational architecture. For example, the organisational aspects are taken into account

thereupon. The logical view is defined mainly at the conceptual level. The development view is relevant only in realisational architecture. The physical and process views can first be considered at the conceptual level and after details are clarified, they are completed to realisational level. Each view may include one or several diagrams or structures that define its content.In this work, the tasks of software architecture design are divided into three phases that are illustrated in Figure 5-4.

- The first task in software architecture development is to *analyse the requirements*. A requirement analysis is performed in order to find out the responsibilities of the designed system and, especially, the quality requirements. If the system is simple, the results can be directly generated into a class model; however, more thorough methods are usually needed. The purpose of this phase is not to make a requirements specification for the system but to modify and enhance the requirements so that they can be used in the architecture evaluation.

- The second task is to design the architecture candidate, i.e. *architecture selection*. The functionality, logical view, is designed for the whole system, not for software alone. The physical view is derived from the hardware specification. The software specific views include the process and development views. The selection of architectural structures is based on the quality requirements.

- *Evaluation* of the overall quality of the candidate is the third task. Evaluation is based on the evaluation criteria defined in the first phase. If the candidate is not good enough, the architecture has to be refined. When the architecture candidate is acceptable, its description can be stored as an architecture specification.

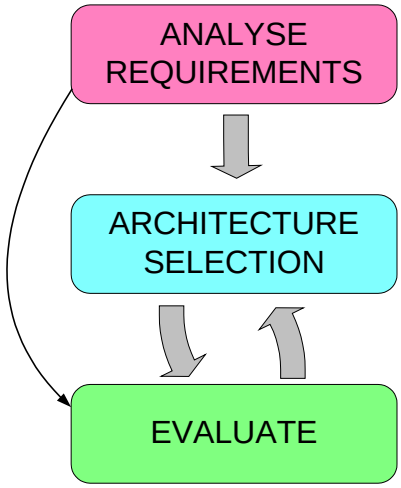


Figure 5-4. Architecture development process.

The software requirements analysis is based on the DSP system requirements specification and the requirements of the hardware platform and other environmental constraints. The DSP system requirements specification should include the specification of both functional and non-functional requirements of the product. If one wants to develop a family of products, it is of necessity that the requirements specifications of each product are available. Requirements are derived from product descriptions and from several standards for multimode products. The requirements are not specifically focussed on software or hardware but on the system. The main results of the requirement analysis are as follows:

- Use case diagrams. Use case diagrams show the responsibilities of the designed system from the user's point of view. A use case is a collection of possible sequences of interactions between the system and its users. The use case diagram is also a context diagram that shows the external connections to the system. An event table can be created from the external events to support the use case diagrams. Only the main use cases need to be defined for the purposes of architecture design.

- Evaluation profiles. The quality requirements are covered with evaluation profiles, which are groups of scenarios that define the coverage of the

quality evaluation. The type of the scenarios depends on the quality attribute it is created for.

The architecture selection includes several types of tasks for each view:

- Identification of the *problems*. The most critical design problems of the view, which are relevant at the architectural level, are defined. These problems or questions are derived from quality goals and external factors that affect the software architecture.

- Defining *strategies*. Design strategies show how the most critical design problems are solved in the view. They are used to clarify the design approach. Initial strategies are defined when starting the architecture design, but the list of strategies should be updated when other significant architectural decisions are made during the architecture refinement. An architectural strategy can mean for example a use of a style or pattern.

- *Modelling* is divided into a selection of architectural structures, validation, and mapping. Architectural structures consist of components, connectors, their topology and other design decisions specific to the view. Functionality can be validated using the same use cases developed in the requirement analysis. Scenarios are derived from these use cases in order to illustrate the interaction of components. Each view describes different aspects of the architecture. It is important that they are in conformity with each other. Mapping serves as a means by which this can be demonstrated.

- *Design rationale*. The design rationale explains refinements and alternative solutions. It is generated from an informal design history for future purposes. The alterations after an evaluation of the architecture are called *refinements*. The solutions that are considered but never actually selected are called *alternative solutions*. During the lifetime of a system, several changes are made. Most of them do not affect the architecture but sometimes the assumptions that are made of the system when designing the architecture are not valid anymore and therefore the architecture has to be modified. The design rationale can be of help therein. Firstly, it shows the assumptions that are made when selecting between different solutions; secondly, it can make

the alterations faster, and show the other solutions which have already been considered and the reason for their not being selected before.

The quality taxonomies are utilised in all the phases of the architecture design (see Figure 5-5). Response and stimulus sections are needed in creating the evaluation profiles and for the actual evaluation. Architectural parameters are the basis for selecting the architectural structures for the system.

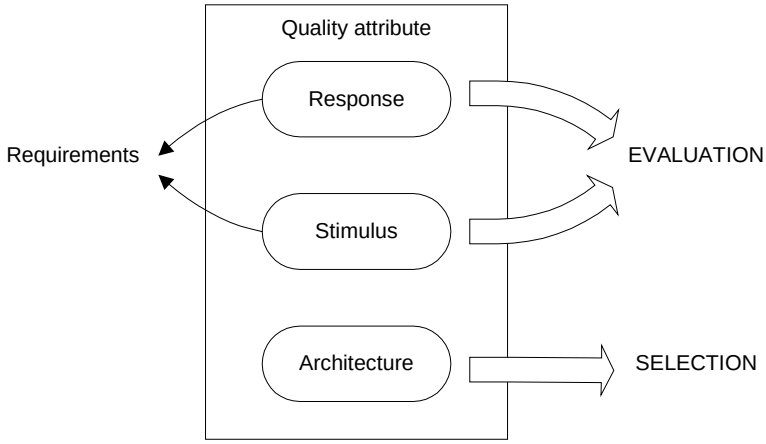


Figure 5-5. Usage of taxonomies in the architecture development.

The approach to software architecture is iterative. Firstly, a software architecture design process is a series of design decisions (see Figure 5-6). Two architecture alternatives may differ only in one aspect but a design decision is always based on the evaluation of the whole architecture candidate. Secondly, the software architecture for an embedded system is always dependent on the underlying hardware architecture. Therefore, there can be various alternative software architectures for each of the hardware architecture candidates. Finally, software architecture includes several views to the architecture. Because of these issues, the approach selected here is to divide the software architecture design problem into several subproblems and then study alternative solutions to these subproblems. Some specific subproblem can be used to separate the work into two or more architecture candidates. After that, several refined versions can be generated from each architecture candidate, using the results of the evaluations.

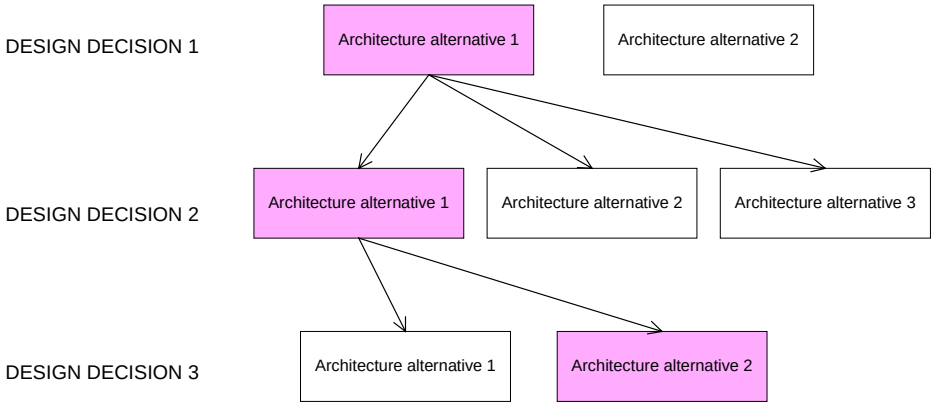


Figure 5-6. Generation of architecture candidates.

5.3 Multimode characteristics

A multimode operation sets certain requirements to the architectural views:

- The logical view should find the common functionality of the different modes. The logical view should include all the functionality that is defined in the product requirements and the control functionality that is needed for managing different modes. The view should define the different combinations of functionality that can be simultaneously active. The combinations are directly derived from the requirements.

- The physical view defines the different mappings from the configurations that were defined in the logical view to the hardware architecture.

- The process view defines the actual runtime configurations for the system.

- The development view should advance the reuse between modes by separating the common functionality to different domains that provide services to the mode-specific domains.

Figure 5-7 clarifies the relationships between hardware and software entities. *HW mapping* defines the location of the functionality in the hardware platform.

It maps the logical functionality into hardware blocks. If changes to mapping are possible during runtime, it is necessary to define more than one mapping.

For each processor defined in a HW mapping, there can exist one or more process mappings. *Process mapping* defines how the functionality defined for the processor is mapped to processes. If mapping can change during runtime then more than one mapping has to be defined.

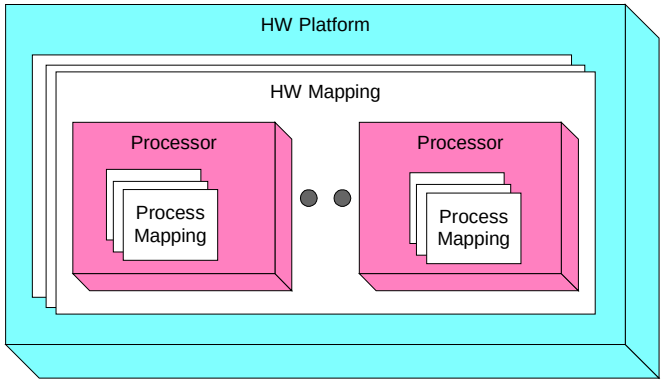


Figure 5-7. Relationship of hardware and software entities.

Figure 5-8 illustrates the generation of runtime configuration. *Runtime configuration* combines the individual process mappings into a configuration of the whole hardware platform. In case the configuration can change during runtime, more than one configuration has to be defined. The initial partition of functionality into processes starts with making one process mapping for each processor in each HW mapping. The refinement starts after this. The quality requirements are taken into account using strategies that optimise the process mappings. The quality is evaluated within the scope of the entire hardware platform. Runtime configurations are the results of the refinement process.

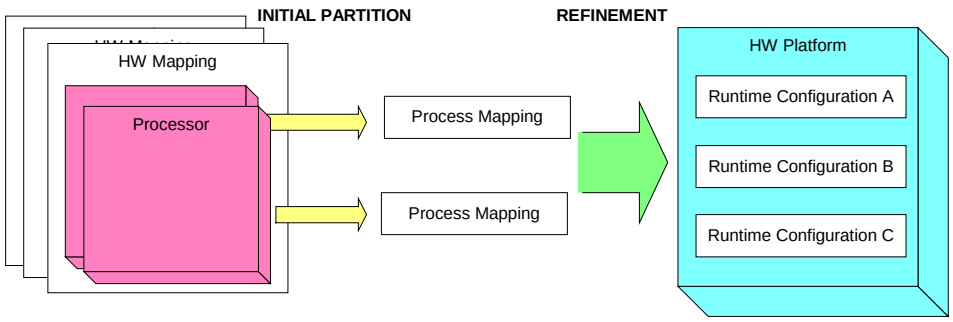


Figure 5-8. Generation of Runtime Configurations.

When more than one runtime configuration is defined, it means that the functionality has to be reconfigured when transferring from one configuration to another. Reconfiguration may have various meanings. For instance, embedded systems are usually constantly reconfigured; parameters are given new values, and hardware resources (memory, processor, ASICs) are reserved and released. However, the old methods may not be enough for future multimode systems. The reconfiguration that may be needed in a multimode system is due to the fact that in order to create affordable products, hardware resources should be spared especially in multimode terminals. This may require, for example, that the implementation platform for an algorithm is selected dynamically only after the algorithm is called.

If the system is changed so that it is stopped and new software is downloaded, it is called *static reconfiguration*. When the composition of the system functionality is changed on the fly, without reset it is *dynamic reconfiguration*. Dynamic reconfiguration has yet another feature; the change can be made either transparently without affecting the normal operation, or non-transparently. The *transparent dynamic reconfiguration* needs special techniques and is the most difficult to implement. The selection between static reconfiguration and dynamic reconfiguration is based on possible minimum downtime requirements. If there are none and there is no need for reconfiguration during the transmission, it is possible to use static reconfiguration.

The first task in designing reconfiguration is to define the reconfiguration unit in addition to defining what kind of reconfiguration is actually needed. A reconfiguration unit is the smallest unit of software that can be individually

replaced. In a multimode system, one example of a possible reconfiguration need is the change of mode during an active connection. For example, a WCDMA call is transferred to a GSM call when the WCDMA connection is lost. When the reconfiguration needs are defined and the reconfiguration unit is specified, the effects on the architecture can be analysed. Reconfigurability can affect the architecture in two ways. Firstly, it might cause additional functionality into the system. Secondly, it may set requirements to the structure and definition of the components. Architecture design can support reconfigurability by including the selecting of an architectural style that supports reconfiguration.

Multimode DSP software has the following kinds of reconfiguration characteristics:

- Mode-specific components. Most of the components are specific to one mode.

- Reused components. Some components are reused at least in two modes.

- Rearranging components. Connections between components are removed and added depending on the mode.

- Parameterised components. Components have parameters, which have different values depending on the mode.

- Components can be implemented in software or hardware.

- Dynamic loading. New functionality may be added to the system by dynamic loading.

- Transfer of state. When changing mode during a connection, the state in the old mode should be transferred and translated to the component in the new mode. In some cases, it may not be enough to set a component to its initial state - the state that it should have depends on the state the system was in when the reconfiguration was requested.

- Timing. Perhaps the most important thing is to control the timing of the mode changes. Mode change should be as seamless as possible.

- Dependencies. Traditionally, DSP software contains a large number of dependencies because it has to be optimised. The dependencies make the runtime reconfiguration harder.

- Modelling configuration, changes, and constraints. The reconfiguration system should include an easy way to model the current configuration, the changes and their constraints.

If a seamless transfer of services between networks is necessary, it means that two networks should be supported simultaneously, for a short period of time. This causes a temporal need for more processing power. One way to get the components of two systems to be simultaneously active could be to lower the requirements of the algorithms temporarily. In reality, it may be difficult to accomplish a full seamless transfer without a duplication of resources (Huotari 1999). In addition, the different requirements of the applications and the different capabilities of the networks require a careful consideration of what kind of mode changes are possible in which situations (Soininen et al. 2001).

6. Architectural views

The purpose of this chapter is to define the architectural views that are useful for designing DSP software. The aim is to define the aspects that are typical of the DSP software development in the views. The architectural decisions should be based on quality goals. The view development consists of finding such values to the architectural parameters defined in the quality taxonomies that the quality requirements are fulfilled.

The software architect does not define the architecture for himself. Architectural views can facilitate discussion between different stakeholders. Table 6-1 maps different interest groups to the views they should be mostly interested in. Some examples of the usage possibilities will also be given. Software architecture can be used in discussions between different types of developers, between developers and managers, and between product development and marketing.

Table 6-1. Stakeholders versus views.

Stakeholder	**View**	**Example usage**
Software developer	Development view	Discussion about the responsibilities of the modules with the other designers.
	Logical view	Training.
	Process view	Optimisation of performance.
	Physical view	Integration of software and hardware components.
Algorithm designer	Logical view	Discussion on requirements and the complexities of the components.
Hardware designer	Physical view	Discussion on hardware/software partition.
	Logical view	Discussion on hardware/software partition.
Marketing	Logical view	Discussion on the required functionality.
Project manager	Development view	Project planning and management.

Figure 6-1 illustrates how the quality taxonomies are utilised in the view development. Defining the critical questions in the architecture design affects the external environment and the quality measures. Architectural parameters point out the kinds of strategies that should be created for solving these problems. Consequently, problems and strategies are a basis for the design rationale and strategies are utilised when modelling the architecture. On the other hand, new strategies may be generated during the modelling. However, also these strategies have to be derived from the quality attributes.

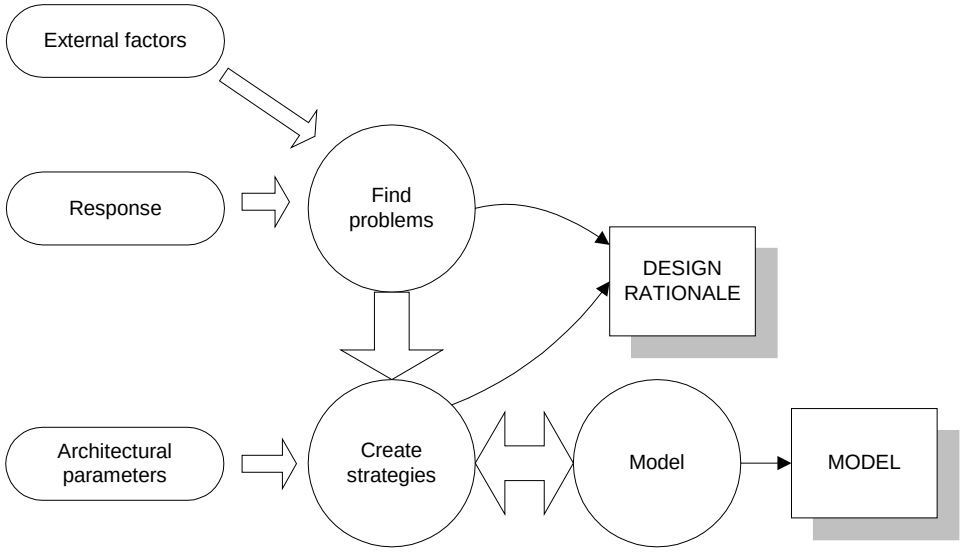


Figure 6-1. Taxonomies in the view development.

The definitions of architectural views for DSP software development have been earlier published in (Purhonen 2002).

6.1 Logical view

The logical view divides the required functionality into components known to the application domain. It describes the functional entities in the system and how they are connected to each other independently of the implementation. Although the logical view is a software term, the logical view of an embedded system should define all the functionality in the system, not just software. In addition,

the high-level logical view could be directly reusable between products in a product line. The logical view should answer the following kinds of questions:

- What are the functional entities that the system is composed of?
- What are the responsibilities of the functional entities?
- How are the functional entities related to each other?
- What are the complexities of the functionalities?

Usually, the logical view is not altered much after it is finalised because it is mainly based on the functional requirements. However, the requirements of the system are often finalised during the architecture development, or even later. Therefore, especially new control functionality or needs for a different partition of the functionality may be found out during the development of the other views.

The tasks belonging to the creation of the logical view are depicted in Figure 6-2. As all the other views, the logical view development starts with defining the problems and strategies. Class modelling finalises the work started in the analysis phase with the functional requirements. The goal in the class modelling is to find common behaviour. The class structure is created by first finding out the object candidates and then adding the relationships between them. The feature analysis and existing experience from senior designers, product documents and literature are utilised. The responsibilities of complex objects can be clarified with drawing state models. In the architecture design, classes should not be defined on a level that is too detailed. Nowadays, the classes are often used more like components such as those presented in (Hofmeister et al. 1999a). A component approach is easier to use in identifying the interfaces without going into any implementation level definitions such as functions.

The logical view has two dimensions: configurations and objects. A configuration table or equivalent is essential for a multimode system. It specifies the states the DSP system can have and defines the objects that are active in each state. Configuration tables are utilised when making the physical view. If a hardware platform is reconfigurable, there can be as many different hardware platform configurations as there are configurations in the configuration table.

Scenarios are used for both validating and understanding functionality. Scenarios are derived from the use cases defined in the requirement analysis. The scenario for a particular use case can be different in each different state that is defined by configurations. Scenarios can be made using MSC diagrams or UML sequence diagrams. If the definitions of the classes are left on an abstract level, one possibility in showing the responsibilities and the interfaces of the classes is to use scenarios. Thus, they would not be made only for validating functionality.

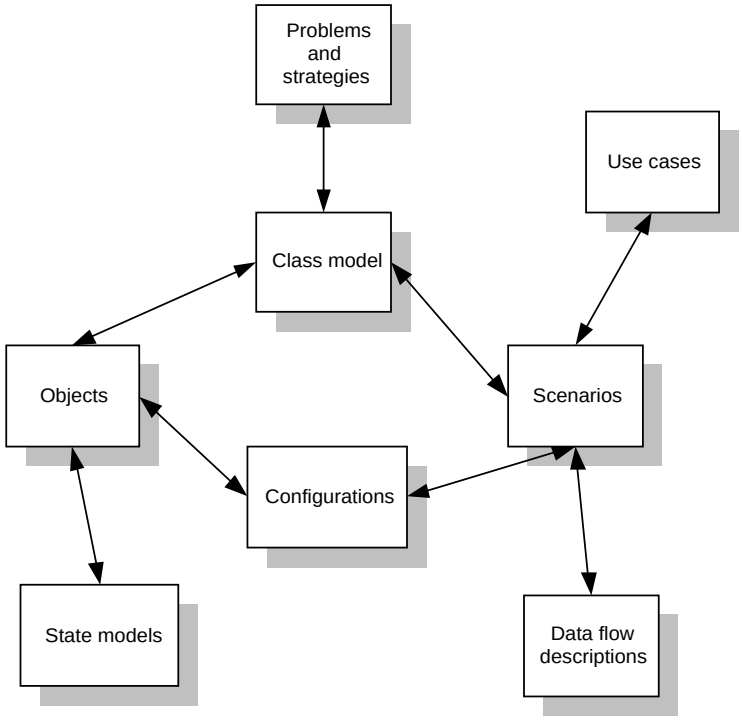


Figure 6-2. Logical view development.

A typical aspect of DSP software is that it includes both data processing and control objects. Class models are good at describing the relationships between classes; however, data processing, in particular, needs additional diagrams for clarification. The interaction between data processing objects can be abstracted to data flow descriptions. Data flow descriptions are composed of objects and data flows between them. Objects are derived from data processing classes in the class model. The results of the algorithm design are utilised. The main functionality of a DSP system is to process data. Data flow descriptions are used

for understanding the signal processing requirements in an implementation independent way. They can be later used as a base to the process view. The members of the data flows can be thought of as active objects.

A summary of the structures in the development of the logical view is presented in Table 6-2. The class model and the configuration table are the only ones that are necessarily needed to show the required functionality, whereas the others are optional. The other structures support the design of more complex systems.

Table 6-2. Logical view structures.

Structure	**Description**	**Used for**
Class model	Components: classes Connections: is inherited from, is part of	Describing functionality in the system. Supports reasoning about similar behaviour.
Configuration table	Two dimensions: configurations and objects	Describing the combinations of functionality that can be needed simultaneously.
Data flow description	Components: objects Connections: may send data to	Describing the connections between signal processing algorithm objects.
State model	Components: states Connections: transition to	Verifying the responsibilities of critical objects.
Scenarios	Components: objects Connections: may send data to	Validating functionality.

A refinement of the logical view is needed when the overall architecture does not fulfil the quality goals that are set for it. Quality goal violations may occur, if there are components that seem to be too complex, or the components have complex interrelationships. The complexity can be handled using strategies and styles and patterns such as in Table 6-3.

If some complex functionality cannot be divided logically, for example an algorithm, it should not be divided on this level. If its complexity is clearly higher than the rest of the functions then it is a good candidate for a hardware implementation. Usually, it is not economical to size the processor capacity according to a single functionality that needs a massive amount of processor power compared to others.

Table 6-3. Refinements for logical view.

Refinement need	**Refinement**	**Possible conflicts**
Complexity of some class is too high.	Strategy: Use functional decomposition and class division rules defined in literature and try to break complexity down.	Too many small classes may increase the overall complexity of the system. Sometimes data processing components are indeed complex, and this has to be taken into account in the implementation.
Complexity of interrelationships between classes.	Strategy: merge classes and reorganise them using decomposition rules.	If too many classes are merged, the complexity is transferred inside classes.
Depending on the requirements, different algorithms may be used for the same operation.	Policy pattern (Douglass 1998): a selection of different algorithms to implement the same black box behaviour.	Indirection can be the source of performance problems.
Depending on the workload, different implementation platforms can be used for an algorithm.	Interface pattern (Douglass 1998): separates interface from implementation.	Indirection can be the source of performance problems.

When the logical view has been prepared, the complexity estimates of each functional entity are created. The algorithms cause the majority of the workload

in the DSP systems. The expertise of algorithm designers is essential in estimating the complexity of algorithm components.

6.2 Physical view

The physical view is needed for describing the external systems that DSP software has to connect. The view serves as input information to the process view where additional objects are created for connections to external components. It is also one part of the runtime view, and therefore, if hardware mappings are changed during the execution, all the possible mappings should be modelled. The logical view and the physical view are actually common views between software and hardware architectures.

The physical view answers the following questions:

- What is the hardware/software partition?
- What are the external connections of DSP software?
- What are the resource usage estimates?

The physical view is composed of the results of the hardware architecture designers. They base their work on the functionality defined in the logical view and the work of algorithm designers. Software architecture designers provide them with comments, utilising the results of the process and development view design. The physical view design is based on the hardware mapping process that selects an optimal implementation platform for each logical component.

The physical view development is constrained by the following quality requirements:

- Processing capacity. Processing capacity requirements can change when the development progresses: new functionality is added, complexity of existing functionality changes, some functionality is removed etc.

- Power consumption. For example, if the current solution consumes too much power, one alternative is to change the hardware platform.

- Resource cost. Cost requirements should be taken into account when selecting hardware. If the cost of the hardware parts changes or the goal set for the total cost changes, the physical view may need to be modified.

The physical view has three dimensions: configurations, objects, and nodes. These dimensions are described with two types of structures: physical structure and mappings (see Figure 6-3). The physical structure describes the hardware components, such as processors and ASICs, and connections between them. A mapping table maps the functionality defined in the logical view to the physical structure. In simple cases, the physical structure and the mapping table can be combined in a deployment diagram where the logical view components are deployed into the physical view components. The functionality is validated by using scenarios.

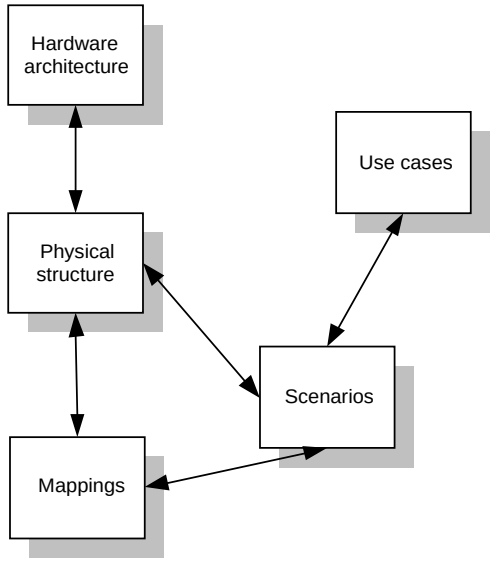


Figure 6-3. Physical view development.

Dynamic reconfiguration may be needed if different configurations reuse the same hardware resources. Although the need for dynamic reconfiguration is mainly decided in this phase, it is modelled in the process view. In order to be

able to study the resource usage, the object requirements should be compared to the capability of the hardware resource.

A summary of the architectural structures in the physical view are presented in Table 6-4.

Table 6-4. Physical view structures.

Structure	**Description**	**Used for**
Physical structure	Components: processors, ASICs, FPGAs, etc. Connection: communicates with	Describing the hardware architecture and context for software.
Mapping table	Three dimensions: configurations, objects, and node	Creating mappings from each object in the logical view to a hardware node in each configuration.
Scenarios	Components: nodes Connection: sends messages to	Validating functionality.

The hardware mapping process finds an optimal execution platform for each object. The physical structure affects performance through the execution times of individual objects. Hardware architecture dictates the maximum performance available by the software architecture decisions. If software architecture is estimated to have unacceptable performance, one solution is to refine hardware architecture. However, the refinement is part of the hardware architecture design so it is not discussed here.

In a multimode system, the goal is to optimise the performance of the whole product - not of one configuration only. Therefore, the development of the physical structure will need several iterations. At least in the final iteration, the results of the process and development views should also be taken into account when analysing the platform's capability to perform the proposed functionality. However, the main factor in performance is the algorithms, which take most of the processing power in DSP systems.

When the functional entities are mapped to their implementation platforms, it is possible to make execution time estimates from them. They are based on the complexity estimates that were done in the logical view. The other resource usage estimates that are most interesting from the software point of view are traffic between hardware units and memory usage. The resource usage estimates are done in co-operation with the hardware experts.

6.3 Process view

The process view is the second part of the runtime view of the architecture. It divides the functionality inside the processors defined in the physical view into independent execution threads. It also defines the internal communication between the executing components in a processor and the external communication to the other nodes.

The process view answers the following questions:

- What is the runtime behaviour of the system?
- What are the updated resource usage estimates?

The qualities that should be fulfilled by the process view is performance and, to some extent, reuse. Performance is used as a main factor in the evaluation, but reuse has to be considered from two points of views. On the one hand, some processes might be reusable assets; on the other hand, runtime reconfiguration is modelled on this level, if it is needed. If there are more than one mapping for a processor in the physical view, there may exist a need for dynamic reconfiguration.

The process view has four dimensions that are defined by a mapping table: objects, process structure components, processors, hardware mapping. If there are more than one processor in the DSP system, each of them creates a subsystem and a dimension to the process view. Development of the process view is depicted in Figure 6-4:

- The process structure is the static representation of the process view. It describes the independent execution entities of the system. The responsibilities of the processes can be sketched using state diagrams although a detailed design of processes is not part of the architecture design. If the process view is complex, it can be abstracted into a metamodel, which explains the basic concepts in the process view.

- The process view defines the mapping from the process structure components to the logical view components. Each processor has separate process mappings. In case the system has several modes that use the same resources, there can be more than one mapping per processor. The definition of runtime configurations combines the individual process mappings.

- Scenarios are used for validating functionality. In addition, they show the dynamic behaviour of the process view. Both control and data flows should be covered.

After software and hardware functions are separated, the hardware components become external components to software. Therefore, there will be new external events (interrupts) to software, which are derived from the physical view.

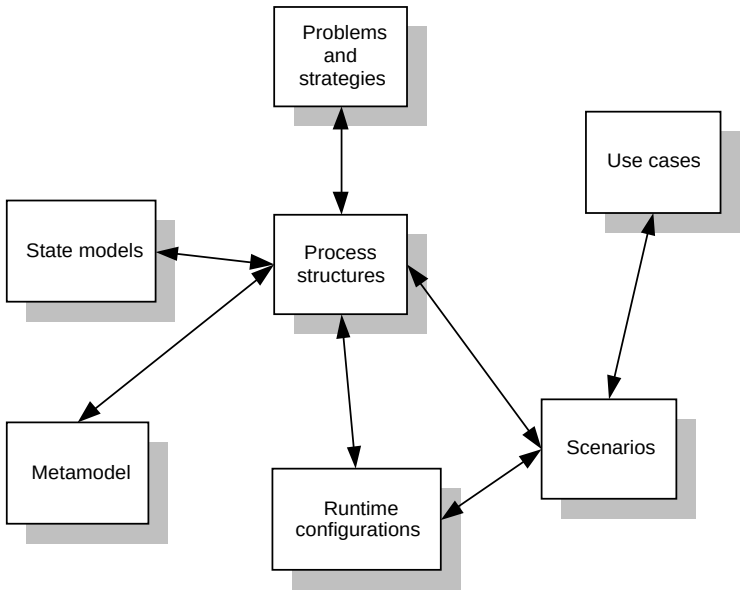


Figure 6-4. Process view development.

The structures of the process view are presented in Table 6-5.

Table 6-5. Process view structures.

Structure	**Description**	**Used for**
Process structure	Components: processes, threads Connections: messages, buffers, etc.	Describing how the system works during runtime. One for each processor in a hardware mapping.
Metamodel	Components: main concepts Connections: is related to	Understanding the main concepts in complex systems. Describes e.g. the used architectural styles.
State model	Components: states Connections: transitions to	Verifying the responsibilities of a complex process.
Mapping table	Four dimensions: objects, process structure components, processors, hardware mapping	Mapping the objects to process structure components, for each processor in a hardware mapping.
Scenarios	Components: processes, threads Connections: messages	Validating functionality.

The development of process structures includes the following tasks: definition of runtime components and connectors, and selection of topology and resource allocation policies

Runtime components are elements in the process view to which the functionality in the logical view (i.e. functional specification) is mapped. Examples of runtime component include process, thread, and data store.

Runtime connectors are used for connecting runtime components. In addition to connecting, connectors can provide the following services (Mehta et al. 2000): communication, co-ordination, conversion, and facilitation. Communication

denotes transmission of data among components, and co-ordination, respectively, transfer of control among components. Conversion connectors enable the interaction of heterogeneous components. They convert the interaction required by one component into that provided by another. Facilitation is an additional service when components can interoperate with each other but there is nevertheless a need to provide mechanisms for facilitating and optimising the interactions. There can be several types of connectors that provide these services (Mehta et al. 2000). The basic connector types can be combined to create more advanced connectors such as remote procedure call (RPC).

Topology means the way the connectors combine the components creating configurations. Dynamic systems have several configurations.

Decisions of *resource allocation* include defining policies for scheduling, memory usage, bus usage, etc. The performance modelling and analysis includes finding out the right process composition and periods and priorities. The goal in setting priorities and periods is to minimise the overhead of context switches and other support functions.

Architectural decisions that should be made in the process view are mainly derived from the performance taxonomy. When styles that include processes or threads are used, task division and priority assignment should be performed. Several authors (Gomaa 1993; Moon 1993; Jones 1991) have studied task division. In priority assignment, the prevention of priority inversion is essential - techniques include priority inheritance and priority ceiling (Klein et al. 1993).

Examples of refinement methods that can be used in creating the process view are presented in Table 6-6.

Table 6-6. Refinements for process view.

Refinement need	**Refinement**	**Possible conflicts**
There are real-time latency requirements with the production of final outputs and the topology consists of multiple processes arranged as concurrent pipelines.	Performance ABAS: Concurrent pipelines (Klein & Kazman 1999)	Potential pitfalls: priority assignment, completion times.
There are real-time performance requirements and multiple processes on a single processor share a resource.	Performance ABAS: Synchronisation (Klein & Kazman 1999)	Potential pitfalls: prioritisation strategy, sources of blocking, priorities used during the critical section.
Some process is too complex.	Task division strategies (Gomaa 1993; Moon 1993; Jones 1991).	When processes are added, context switches are increased.
Handling of data flows.	Pipes and filters (Buschmann et al. 1996).	Indirection can be the source of performance problems.
Handling of data flows, when the scheduling order cannot be defined beforehand.	Blackboard (Buschmann et al. 1996).	If the scheduling is very straightforward, blackboard probably causes too much overhead.

When the process view has been prepared, the resource usage estimates can be updated with values that are more accurate and that also contain the workload of control functions. The complexity estimates may need to be updated with the additional workload of control functions such as operating system calls.

6.4 Development view

The development view provides the architecture with the organisational view. It presents the actual modules that should be developed. Therefore, it facilitates project planning. Modules can be grouped into different levels of abstractions. Multimode DSP software has to fulfil similar requirements as the product-line software. Thus, although the development view is product-specific, it may take into account some product-line requirements.

The questions that the development view answers are the following:

- What are the actual software modules that should be developed?
- Can COTS components be used?
- Is it possible to reuse components from earlier projects?
- What are the reusable assets?

The development view may be constrained by the decisions that the organisation has made concerning COTS components and reuse of older software. Such can be the case, for example, when a decision has been made upon using a specific operating system. Additionally, if product-line architecture is to be built, the product-line specification informs one of what are the reusable assets in the product family. The qualities that restrict the creation of the development view are mainly product-line concerns and development time. Cost is considered when COTS component recommendations are given.

The development view is composed of a module structure and an associated mapping table (see Figure 6-5). Modules are the actual software components that will exist in the DSP system. The mapping table maps subsystems and components to the logical view. The execution architecture (physical and process views) is used for specifying the support components. The module structure is characterised by the following entities:

- Domains. Each group of functions is put into a particular level of abstraction. The criteria for the division of functionality can be, for example,

distance from the hardware. Domains should be strictly separated from each other, and in principal, domains should not know the internals of the other domains. It is the layered style that is traditionally used in the composition of domains. Sometimes it is necessary to make exceptions to the rules of this style, e.g. for reasons of efficiency. In such a case, the reason should be explicitly explained.

- Subsystems. When the structure of a domain is still complicated, it should be further divided into subsystems. For example, RTOS or libraries of COTS components can be subsystems inside domains.

- Components in the development view are the modules that should be developed. Relationships between the modules are *is_composed_of* or *uses.*

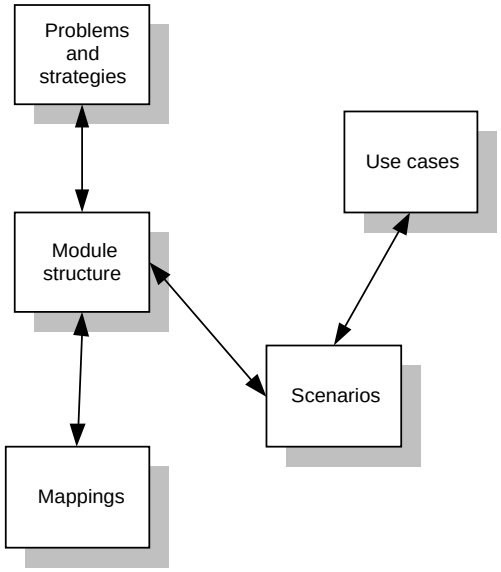


Figure 6-5. Development view design.

The structures in the development view are summarised in Table 6-7.

Table 6-7. Development view structures.

Structure	Description	Used for
Module structure	Components: modules, subsystems Connections: is a submodule of	Defining the modules that have to be developed or acquired as COTS components or reused from other products.
Mapping table	Three dimensions: objects, modules or subsystems, domains.	Mapping the defined modules and subsystems to the objects from the logical view and process view.
Scenarios	Components: modules Connections: sends messages to	Validation of functionality.

The patterns that can be used for refining the module structure are presented in Table 6-8. The module structure is usually based on the layered architectural style. In addition, modules are also classified by whether they are included in reusable assets or not. Some new modules can later be accepted as reusable assets in the product family.

Table 6-8. Refinements for development view.

Problem	Refinement	Possible conflicts
Architecture does not fulfil modifiability scenarios.	Modifiability ABAS: Layering (Klein & Kazman 1999) Minimising the effect of changes to one layer when a portion of another layer changes.	Can the system's performance afford the number of layers proposed? Is the layering strict or is there any bridging?
Architecture does not fulfil portability scenarios.	Wrapper. For example, hardware wrapper hides the changes in the hardware. (Schmidt 1999)	Indirection may cause performance problems.

The challenge in the development view is to know to what extent the architect identifies the components. The domains represent different types of expertise that are needed from the developers. At some point, the experts of each field should take the responsibility for the definition of the components. The architecture description of a domain should include the knowledge of the domain that the other domains need. Anything internal to the domain need not to be included in the architecture description. The development view is used especially in the project planning and integration. It is used for the creation of development time estimates for each module and subsystem.

6.5 Summary

The development of DSP software architecture is based on four views: logical, physical, process, and development. Each of them illustrates different aspects of the architecture. The relationships of the architectural views are illustrated in Figure 6-6. The logical view divides the required functionality into meaningful components. The complexity estimates of the functionality in the logical view are needed when searching for the optimal implementation platform in the physical view. The physical view provides the hardware/software partition when defining the hardware platform. In addition, it gives the initial resource usage estimates for the process view. The process view is used for validating the runtime operation of the system. The development view shows how the system is actually developed. The development view gets the division of required functionality from the logical view, additional hardware related components from the physical view, and support functionality from the process view.

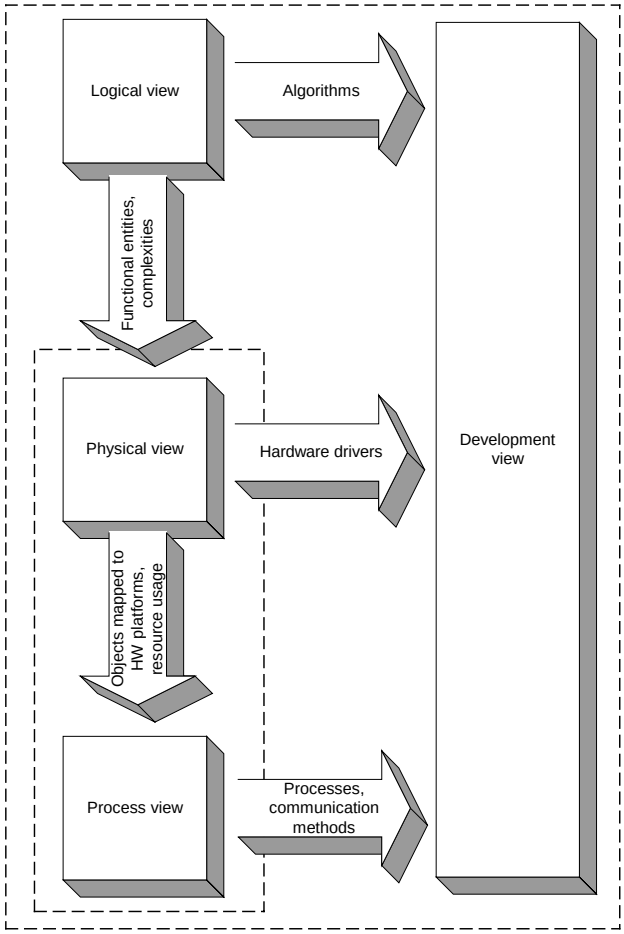


Figure 6-6. Relationships of architectural views.

7. Architecture evaluation

The purpose of this chapter is to present an evaluation strategy for comparing DSP software architectures. Architecture evaluation is based on the defined quality attributes.

7.1 Evaluation strategy

Products have to be evaluated from two perspectives: ability to fulfil the functional requirements, and ability to respond to the quality requirements. Classic testing often concentrates on functionality. In our case, the goal was to find tools for a designer to choose between architecture candidates. Because functionality is not architectural in nature (Bass et al. 1998), the basic assumption is that the required functionality exists in every case. Because of these reasons, this study concentrates on quality evaluation. To ensure that all the functions are unambiguously defined in an architecture representation, scenarios are derived from use cases when developing the views.

In this work, the evaluation is considered from the point of view of confidence building. It is assumed that the architect makes the evaluation, and when the results should be accepted, the review team makes the risk assessment. In order to prepare for the review, the architect may want to ensure that the result of the risk assessment will be acceptable so there is a need to define evaluation criteria that also cover the risks. However, it is often difficult for a designer to find out the exceptional situations; therefore, a risk assessment by an external review team is desirable. The two sets of criteria for evaluation may consequently overlap, but the starting point for the evaluations is nevertheless different.

Although this study concentrates on the tools of the architect, the same techniques that are used for evaluating the architectural decisions in the architecture design phase can be used for re-evaluation in the later phases of the product's life cycle. In addition to the internal changes in the product's requirements, the advances in the external environment such as advances in techniques and methods may require architecture re-evaluation. Consequently, the results of the evaluation done by an architect can be used:

1. To guide the design process. They show what should be done in order to make the design better.

2. To open the design process. They show why decisions are made. If a wrong decision has been made, it is possible to find out why the decision was made; thereupon, it will be easier to carry out due corrections. In addition to that, the wrong decisions can be detected earlier.

Quality attribute evaluation at the architectural level is a difficult task. In fact, the quality of the architecture will be found out properly only in the course of the following years when the architecture is used in the development and the products are released. Thus, the value of the architecture is assessed continuously during the different phases of its life cycle.

Because many properties of the system are still unknown in the architecture design phase, there may exist doubts about whether the evaluation based on the estimates is reliable enough. However, the purpose is not to give any exact values of the quality of the architecture but to make comparisons and find possible problems. The earlier the problems are found the better. If evaluations are done faithfully during development as soon as a problem is found out, the problem can be corrected by utilising the development design history. With the carefully designed evaluation criteria, the architecture can be continuously assessed when better estimates, such as execution time estimates or actual values, are available and the implementation advances. If architecture evaluation helps in avoiding problems and finding problems at least a bit earlier, it can be considered useful. In order to encourage the continuous assessment, the evaluation should be easy to perform.

The target is to define an evaluation strategy and to find the minimal combination of evaluation methods that can be used in order to be able to tell the difference between DSP software architecture candidates. The methods are selected by using the following requirements: easy to learn, fast to perform, and expensive tools not necessary. In addition, the evaluation procedures should be repeatable. Although it should be possible to manage without any additional tools, it is an advantage if there exists one. During the architecture development, it is the architecture designer, who mainly uses these tools. However, they can also be useful for proving for the review team that the selected architecture is the

best candidate, or, as testing tools during the later phases of the software development.

The problem with the existing evaluation methods is often that they are not meant for the architecture phase and they are too time-consuming to be used for architecture evaluation. Usually, evaluation methods need some sort of implementation of the system. Use of a prototype can be useful also in this case. However, because the underlying hardware is usually not ready when DSP software architecture is designed and the tools are not ready either, this is often not possible.

Another reason for simple tools is that DSP software is usually not very large compared to many other systems which are often used as examples of architecture design. Most of the work is done in the signal processing algorithms and the functionality is generally very restricted by the standards. Thus, it would not be cost-effective to spend too much time in evaluating the architecture candidates or individual architectural decisions.

The proposed architecture quality evaluation strategy is summarised in Figure 7-1. It is divided into three main phases: creation of evaluation criteria, actual evaluation, and architecture refinement. The results of the evaluation process are quality attribute profiles, evaluation result report, and the refined architecture. The evaluation strategy combines several of the methods and techniques used in the literature. The following chapters describe the phases in more detail.

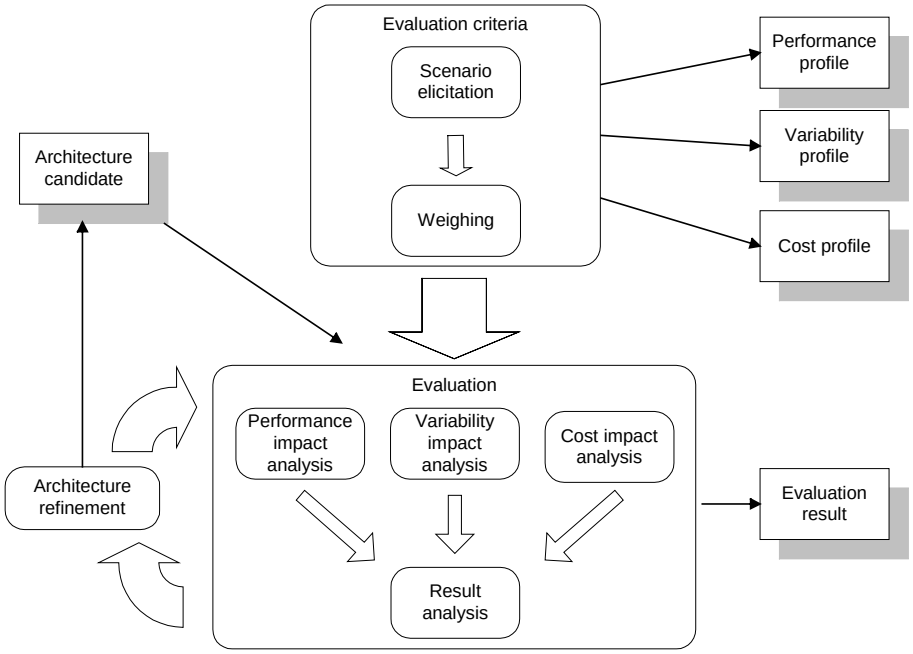


Figure 7-1. Quality evaluation strategy.

7.2 Creation of evaluation criteria

The evaluation is based on the quality attributes specified for DSP software. The preparation of evaluation criteria should be a teamwork of all the stakeholders of DSP software. Evaluation criteria are created in two phases:

1. *Scenario elicitation*. Scenario elicitation includes the creation of profiles for the three quality attributes. The selected scenarios should be as concrete as possible. Scenario elicitation is based on the quality taxonomies (see Figure 7-2). A stimulus section contains the variables that create the space where scenarios should be elicited. A response section is used for restricting the profiles to include only the most critical scenarios.

2. *Weighing*. After scenarios are defined, each of them is marked with weights. Weight shows the relative criticality or importance of the scenario. On the other hand, especially in variability scenarios it shows the likelihood of the

appearing of the scenario. Weight has two purposes. It is used in restricting the amount of scenarios in detailed impact analysis and in tradeoff analysis.

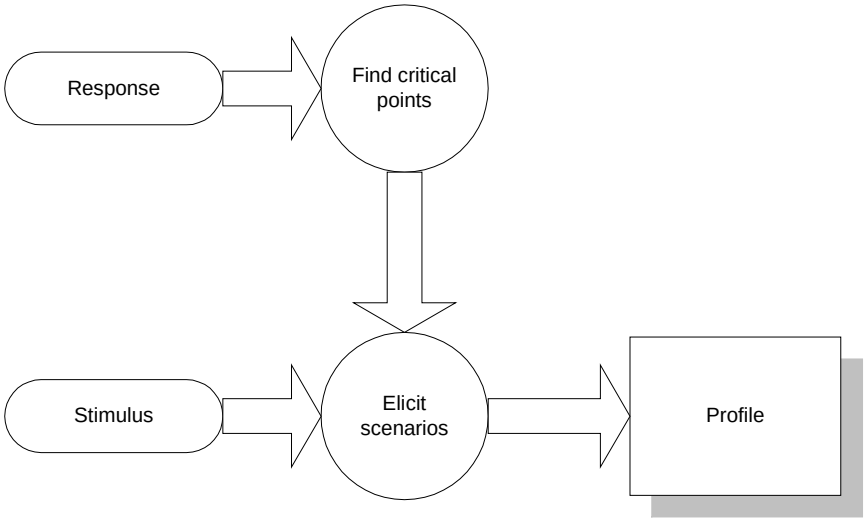


Figure 7-2. Scenario elicitation.

Performance profile is a group of worst-case usage scenarios. Creating a performance profile for a DSP software is a challenging task. The usage of several types of resources should be balanced so that worst-case situations can still be under control. Different products may have different types of preferences for resource usage. The art of evaluation is to find the critical scenarios - limiting the number down to as few scenarios as possible - to make the evaluation process fast.

The questions that should be asked when finding out the performance scenarios are derived from the taxonomy. For example, when is it possible to exceed latency requirement? Obviously in a real system, we cannot exceed resource utilisation limits, but in our case, those limits are not fixed in the beginning. Furthermore, when latency or throughput requirements are violated, the reason for this is probably that there has been resource usage problems; therefore, one scenario may serve as an answer to several questions.

Cost profile is dependent on how many different feature sets have been defined for the architecture. In addition, if the products are going to be developed incrementally, each version can be analysed separately. Cost profile is used

when trying to find the optimal feature set for a system or for comparing architecture candidates for a particular feature set. The profile should include the scenarios that are critical in terms of product and development cost.

Variability profile includes the change scenarios. Because it is impossible to detect and use all the possible scenarios that may exist, the scenarios are handled in groups. A number of representative scenarios should be created for each group. First, the basic scenarios are formed based on the sources of changes as shown in the variability taxonomy. After the most probable scenarios are found, some risk assessment scenarios should be thought of in each source group. The following categories of complex scenarios are used (Lassing et al. 1999): 1. Adaptations to the system with external effects. 2. Adaptations to the environment with effects on the system. 3. Adaptations to the macro architecture. 4. Adaptations to the micro architecture. 5. Introduction of version conflicts. After scenarios are elicited from each group, the possible duplicate scenarios can be removed, and it may be possible to combine some scenarios. The idea of groups and categories is to help to cover all the possible scenario types. It is possible that the final scenarios could be allocated to more than one group.

7.3 Impact analysis

The evaluation starts with creating an impact analysis of each quality attribute (see Figure 7-3). An impact analysis is based on the created profiles. In the impact analysis, the impact of each scenario to an architecture candidate is assessed.

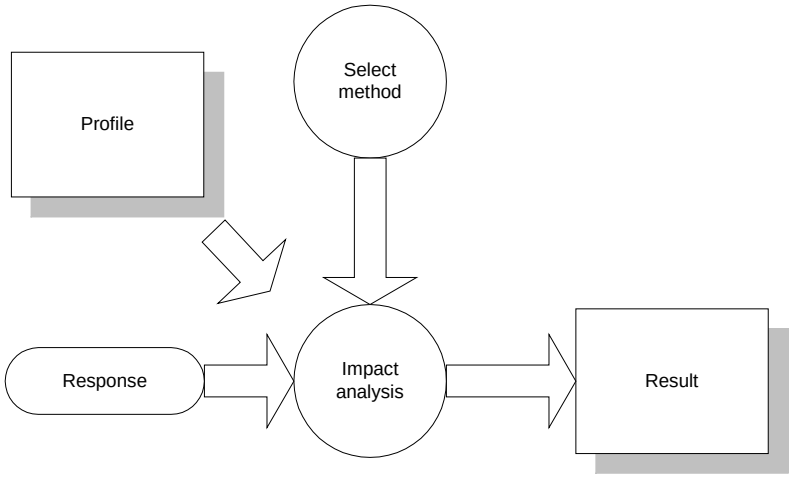


Figure 7-3. Impact analysis.

For the impact analysis, several estimates have to be derived from the architecture (see Figure 7-4). Complexity estimates are derived from the logical view, the physical view provides the initial resource usage estimates for each component, the process view refines the resource usage estimates, and the development view is used for estimating development times.

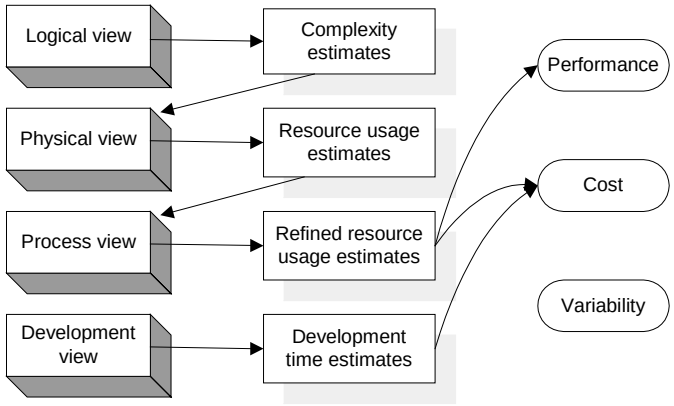


Figure 7-4. Estimates from the architectural views.

7.3.1 Performance

Analytic models provide a basis for relating quality attribute parameters such as queuing policies and execution time estimates to quality attribute measures. A schedulability analysis is appropriate for acquiring worst-case behaviour. Many of the results of the scheduling analysis are either directly applicable to a performance analysis of real-time systems, or offer valuable intuition. A queuing theory can be used to model a system as one or more service facilities that perform services for a stream of arriving customers.

Because the actual resource usage of the components is not yet known in the architecture design phase, estimates will be used in the analysis. These estimates can be replaced with real values as soon as they are available. There are worst-case execution time analysis tools available but they are usually for analysing implemented code, not architectural level descriptions. In fact, the actual values of the performance concerns are not even interesting because they depend on the implementation. It is more important to find out how architectural decisions affect the concerns and, specifically, what kind of conflicts there are between the concerns.

The performance measurements are made for each of the scenarios defined in the performance profile. For each scenario, the values of performance measures (latency, throughput, memory utilisation, processor utilisation, bus utilisation, power utilisation) are estimated and then compared to the requirements.

A performance impact analysis is made by using the process view. In addition to the profile that defines the test cases for the impact analysis, the following information is needed: resource usage estimates for runtime components, and resource usage restrictions defined in the physical view.

7.3.2 Cost

A cost impact analysis is divided into the following tasks:

- The *product cost* depends on the software/hardware partition and the software resource usage. The worst-case resource usage values can be derived from the results of the performance evaluation.

- *Effort* estimation is divided into module development times estimation and integration time estimation. Module development time depends on the complexity of the module, the size of the module, and the experience of the developer. Of these, architecture can affect the first two. Module development time estimates can be derived from the complexity estimates prepared for the functional entities. Integration time depends on the dependencies between the developed modules and on the overall complexity of the system. Metrics can be used to support this work.

- The amount of *personnel* needed for the changes depends on the skills of the staff and the amount of concurrency needed in the work.

- The *duration*, i.e., time-to-market, is counted from module development times and integration time. The duration depends on how much concurrency can be used in developing the system.

Every architecture evaluation should consider the cost consequences of the proposed design. The external factors, and not only the software, have to be also considered in the cost analysis. For example, software architecture affects the product cost only indirectly through resource usage. If software exceeds the resource usage limits, it should be considered whether it should be the software or the hardware that needs to be changed. If hardware is changed, this will affect the product cost.

7.3.3 Variability

A variability impact analysis is performed in two phases. First, each scenario is analysed individually. The second task is to reveal scenario interaction. The values for variability measures should be estimated for each scenario in order to be able to analyse the impact of the scenario. The variability measures are the cost of the change and the complexity of the architecture after the change.

When comparing two architecture candidates, it is important to compare their overall complexities. It is possible that some architecture fulfils all the possible change scenarios, but because of that, its overall complexity is higher and it is therefore harder to implement and test. Another architecture may be easy to change although it does not directly fulfil all the change scenarios.

Because exact values cannot be defined, only relative values are used for evaluating the impact of the scenario. In our case, we sum up the results of the four questions and give one qualitative value for the impact:

- Minimum impact: no change, or so small a change (e.g. only to one existing component) that there is no need to change the architecture.

- Low impact: several components are modified or new components are added, but they are of the same type than the old ones.

- High impact: changes to architectural views and structures.

- Maximum impact: major reforms to architectural views which means that completely new architecture is needed.

If there is a maximum impact for a relevant scenario, the evaluated architecture cannot be accepted. In case of minimum impact, there is no need for changes in the architecture. The most interesting cases are the high and low impact. All the high impact cases should be carefully studied and considered if there is a way to reduce the impact. In case of low impact, the weight of the scenario is taken into use. In case there are a large number of scenarios to be analysed, the scope of the detailed analysis should be narrowed. Using the weights, it is only the most important scenarios that have low impact that are studied in more detail.

The other part of impact analysis is to study the scenario interactions. Clustering means that several scenarios cause changes to the same architectural component (Bass et al. 1998). First, the scenarios can be variations of a same scenario, which is a good thing and means that the system's functionality is modularised properly. Secondly, the scenarios may be entirely different. High interaction among scenarios that are fundamentally different corresponds to low cohesion and suggests high structural complexity. High interaction among fundamentally

similar scenarios signals high cohesion. Additionally, if a group of similar scenarios affects many different components throughout an architecture, modularisation has not succeeded.

7.4 Result analysis

The main task of result analysis is to identify the quality conflicts. Quality conflicts are the features that are advantageous to one quality but unfavourable to another quality. After the quality conflicts are found out, the weights can be used to make compromises. A result analysis is divided into three tasks:

1. An analysis of the individual qualities based on the impact analysis results. In the event that the architecture does not fulfil the quality requirements, changes are proposed.

2. An analysis of conflicting requirements, using the impact analysis results. Each proposed change is analysed for its impact concerning the other quality attributes.

3. An analysis of the evaluation results, and making a decision upon whether the architecture candidate thus fulfils the requirements or not. If the candidate is not good enough, refinements are needed.

In this phase, the result takes the form of a decision upon whether the architecture fulfils the requirements or not. If not, then a decision should be made about the possible realisation of the proposed refinements. If the refinements to software are too costly to make, refinements to algorithm design or hardware architecture should be considered instead. This is because the ultimate goal is to optimise the whole DSP system, not just software or hardware separately.

7.5 Architecture refinement

The architecture refinement is performed in two phases:

1. An analysis of the required changes. The refinement of the architecture can be directed to software architecture, hardware architecture, or even to algorithm design. After the refinement responsibilities are divided between these three areas, the actual refinement can be performed separately.

2. Actual refinement. Each view has specific patterns, styles and other strategies that can be used to achieve better quality.

If initial assumptions for the architecture are changed, the architecture has to be re-evaluated. For example, any change to the system may affect its performance because performance is optimised to the particular set of operations. Therefore, after each refinement, the architecture should be re-evaluated. In addition, when some actual or better values for the execution time estimates or other estimates are received later on, the architecture should be re-evaluated.

8. Validation

8.1 Case study

The case study that is used to validate the architectural methods and techniques is a hypothetical personal mobile multimedia terminal called Multics terminal (Soininen et al. 2001). The case study was used as a source of ideas and as a validation vehicle in the Multics project. Although the whole terminal was specified in the requirements specification, the focus of the research was on the lower layer services and especially on the digital signal processing parts. This domain was called *Transceiver*. Both hardware and software architectures were considered. This work utilises the results of the software architecture research.

The goal was to create a terminal that could work optimally in any network. 'Optimally' meaning that it provides the best performance possible with the available resources at any time. In the initial requirements, Transceiver supports WLAN, GSM, and WCDMA standards. GSM and WCDMA are alternate standards but WLAN can be utilised simultaneously with either of them. The mode of operation depends on the user needs and the availability of the networks. The operation of Transceiver should be transparent to the user.

8.2 Requirement analysis

The functional requirements were analysed using use case-diagrams. The high level use case diagram for a Multics terminal is depicted in Figure 8-1. The use case diagram also shows some of the internal functionality, because telecommunication systems are traditionally divided into layers. The Radiolink actor refers to physical connection whereas Network actor creates a logical connection to the terminal.

The use cases Transmit messages, Receive messages, and Control transmission describe the responsibilities of the Transceiver. Control transmission was further specified as low-level use cases. The problem with use case diagrams is to find the right level of detail in which to define responsibilities. If there is too much detail, use cases restrict the architecture design too much, but if there are too few

details, the actual responsibilities remain unclear. Generation of use cases for a multimode terminal requires a good knowledge of the application area and understanding of the standards.

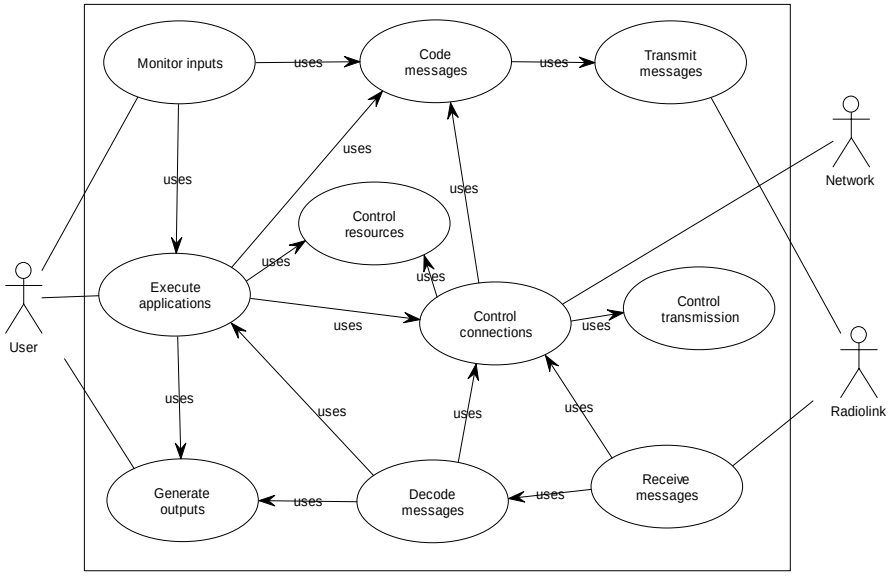


Figure 8-1. Multimode terminal use case diagram.

Evaluation profiles were defined for each of the quality attributes. The preparation of a performance profile needs an expert in the DSP domain both in hardware and software. Finding out the critical scenarios is a difficult task. The cost of the resources affects the type of the resource that is being used and the maximum possible value there can be for its usage. Therefore, there are two types of processes for creating performance profiles: one where resource usage restrictions are set beforehand, and another, more difficult one, where resource usage is balanced during the architecture evaluation. In this case, we did not fix any of the variables beforehand.

Table 8-1 shows two different types of example scenarios for performance profile. Scenario 1 is used for ensuring that the maximum available values are never exceeded for latency, processor utilisation and memory utilisation, in order to ensure reliable operation of the system. Scenario 2 is for user convenience. The most typical user scenario is considered to be GSM voice call,

and the power consumption should be minimised in that condition. Both of these scenarios have equal importance.

Table 8-1. Performance profile.

Scenario	**Description**	**Weight**
1. Simultaneous high-bit rate connections in WLAN and WCDMA	Scenario initiation: WCDMA transmitting Turbo coding WCDMA receive Turbo coding WCDMA cell search for handover WLAN receiver is activated Assumption: Whenever there is a possibility of WCDMA blocking resources from WLAN, this will happen. Scenario execution: TransceiverFrontEnd receives data and WLAN starts receiving 54 M/s. (Event = RF-DATA) In addition, WCDMA both receives and sends data (RF-DATA, BAS-DATA).	1
2. Plain GSM connection	Scenario initiation: GSM voice call is going on. Assumption: Naturally, both receiver and transmitter are working. Scenario execution: TransceiverFrontEnd receives data and GSM starts processing received data. (Event = RF-DATA). Protocols send also data to be transmitted (BAS-DATA).	1

Cost was estimated only for one feature set, which was defined in the requirements. In addition, the cost analysis is difficult if not impossible without the knowledge of the organisation that should develop the product. Therefore, the results of the cost analysis could be used only for comparing architecture candidates. With the proposed functionality, the following measures were estimated:

1. What is the hardware cost for the product (use estimates of maximum memory utilisation of different types of memory and processor type)?

2. What are the module development times? If the architecture allows using COTS components, then it is considered as an advantage.

3. What is the integration time?

4. How many people would be needed to develop this?

Variability profile consists of the scenarios of which some examples are presented in Table 8-2. Variability scenarios should be gathered from several people that represent different interest groups. In our case, this could not be done, but scenarios were gathered to different categories based on the experience and literature.

Table 8-2. Variability profile.

Source	**Scenario**	**Weight**
Changes in the external components	Change the type of a processor.	0.8
Changes in functional requirements	New mode in addition to GSM/WCDMA/WLAN is needed.	0.5
Changes in quality requirements	A cheaper version of the terminal is needed. The SW should be fitted to one processor instead of two	0.7

8.3 Architecture selection

This chapter describes the created views. The views are based on the object-oriented approach and UML notation. The evaluation of the constructed architecture is presented in Chapter 8.4.

8.3.1 Logical view

The critical problems studied for the logical view were:

1. How to enable multimode operation?

2. How to enable an optimal usage of resources?

The logical view is composed of class models, scenarios, state diagrams and data flow descriptions. A class model defines the types of objects that can be used in the system and the various kinds of static relationships between them. Figure 8-2 depicts the conceptual class model for the Transceiver. The class model allows a dynamic reconfiguration of connections between terminal applications and the networks.

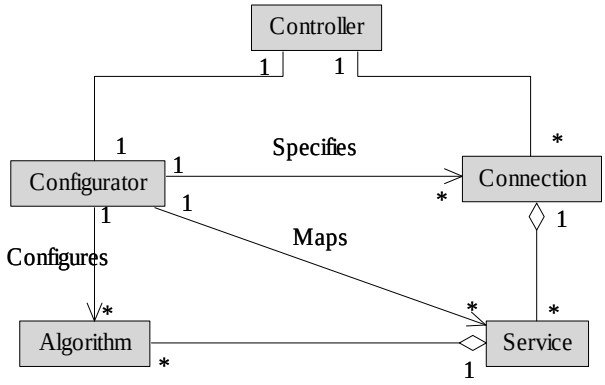


Figure 8-2. Transceiver class model.

The main strategies followed in the class model were:

1. Scalability: There can exist more than one simultaneous radio connections, to the same or different networks. The Controller is separated from the Connections having responsibility of the things that concern all the Connections. The functionality related to individual radio connections is then encapsulated in Connections and Services.

2. Adaptability: The data flow members i.e. Services are separated from their implementations, i.e. Algorithms. There could exist more than one mapping from a Service to Algorithms. Additionally, two different Services could use the same Algorithm. Services contain the specifications on what should be done and Algorithms contain the capabilities of what can be done.

3. Modifiability: Separation of concerns. The configuration issues are separated from the application issues. The management of resources is performed in one place and not distributed in every Connection. Controller handles external events and forwards them to Connections when necessary.

4. Extensibility: New types of algorithms and new types of modes can be added.

The internal behaviours of the classes are defined by state diagrams or algorithm descriptions. The collaboration of a group of objects is illustrated with interaction diagrams in Figure 8-3. An interaction diagram, scenario, captures the behaviour of a single use case. It shows the collaborating objects and the message flows between them.

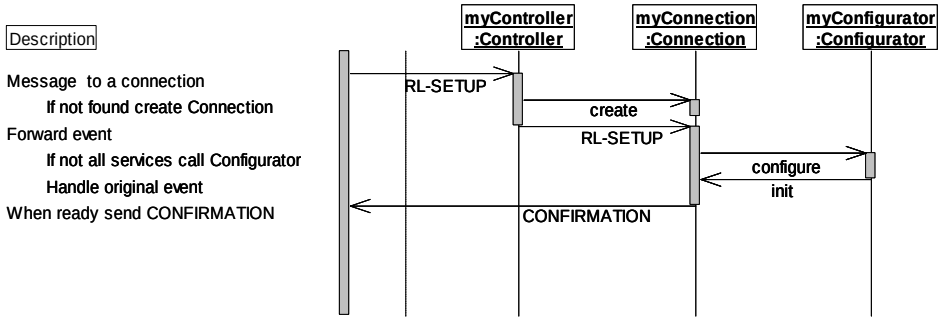


Figure 8-3. Interaction diagrams are derived from use cases.

Configuration tables have two responsibilities. Firstly, they define the states the system can be in, i.e. configurations. Secondly, they show the implementation alternatives for those configurations. We defined about 30 configurations for Transceiver that defined different combinations of WCDMA, GSM, and WLAN functionality that could be simultaneously active. First, the possible states for each standard were specified and then the required combinations were derived

from the user requirements. Data flow descriptions were used for illustrating how the configurations are actually formed. Data flow descriptions show how the chains that perform the data processing are actually created. The responsibilities of the control classes were verified with state models.

The class model is used especially for finding common behaviour. In addition, the logical view stays the same independently of the particular implementation platform of the components. The same logical view can be used for several products with different hardware platforms. Algorithms have different kinds of performance requirements compared to control code. When algorithms and control components are separated on this level, it is easier to take into account their diverse requirements. The logical view defines what kind of functionality can be simultaneously active. When configurability requirements are cleared in the logical view, the requirements of different configurations can be estimated and used in making the other views.

The logical view was found useful for discussing the required functionality. The contribution of algorithm experts is essential in this phase. However, UML and the object-oriented approach are not actually so easy to understand for those who are not already familiar with them. Therefore, only the basic features, which can be explained in the beginning of the process, should be used.

8.3.2 Physical view

The physical structure derived from the hardware architecture is illustrated in Figure 8-4. The physical model consists of two processors (Codec and Sync) and three hardware nodes. In addition, the external software (Protocols) and shared memory that can be accessed by all the nodes are presented as nodes. The physical model is, at the same time, a context diagram to the process view. It shows all the external connections to the Transceiver software.

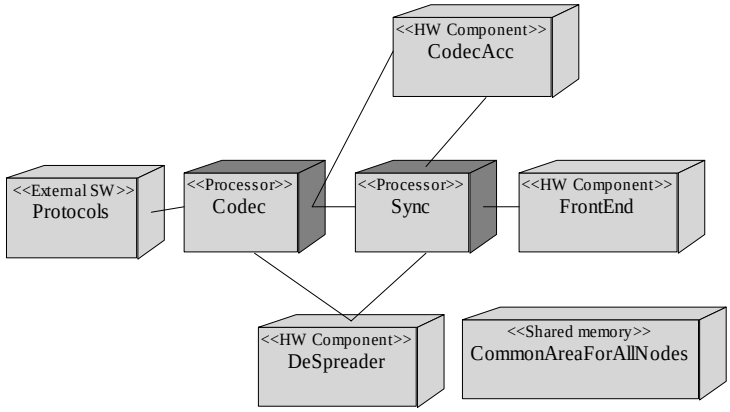


Figure 8-4. Physical model for Transceiver software.

8.3.3 Process view

Following questions are answered by the process view:

1. How are data processing tasks arranged?

2. How is control of data processing arranged?

3. How is data transfer between tasks performed?

4. What is the scheduling policy?

5. How are the operation parameters transferred to the data processing?

6. How is the multimode operation supported?

The questions are based on the study of the features of the hardware platform and the quality goals. The goal has not been to define independent problems but different aspects of the process view design; therefore, the solutions may also overlap.

Two candidates were prepared for the process view. The first version is based on the pipes and filters style (Buschmann et al. 1996) (see Figure 8-5). It is especially designed for systems that process streams of data. Each processing step is encapsulated in a *filter* component. *Pipes* implement the data flow between adjacent processing steps. The sequence of filters combined by pipes is called a *pipeline*.

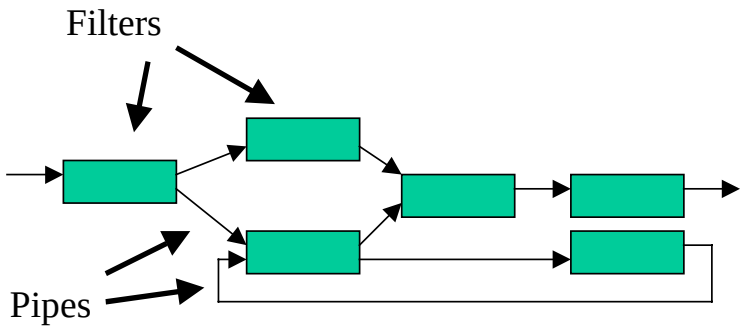


Figure 8-5. Pipes and filters style.

The blackboard style is applied in the second candidate. The idea behind the blackboard style is a collection of *independent programs* that work co-operatively on a common *data structure* (Buschmann et al. 1996) (see Figure 8-6). Each program is specialised in solving a particular part of the overall task. There is no predetermined sequence for their activation. Instead, the direction taken by the system is mainly determined by the current state of the system.

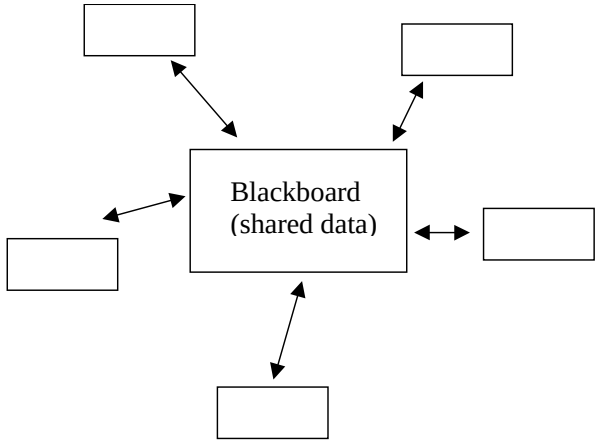


Figure 8-6. Blackboard style.

Two types of diagrams were developed for modelling the process view. The metamodel of the first process view candidate is depicted in *Figure 8-7*. The metamodel shows the main entities of the Transceiver from the runtime point of view. The data processing is performed in pipelines that are constructed from pipes and filters. Filters can be either software processes (DataProcess) or hardware components (HWComponent). Pipes are constructed from interprocess communication (IPC) and hardware drivers. A pipeline manager takes care of forwarding the configuration parameters to the pipelines. A resource manager controls the assigning of resources from nodes to the pipelines.

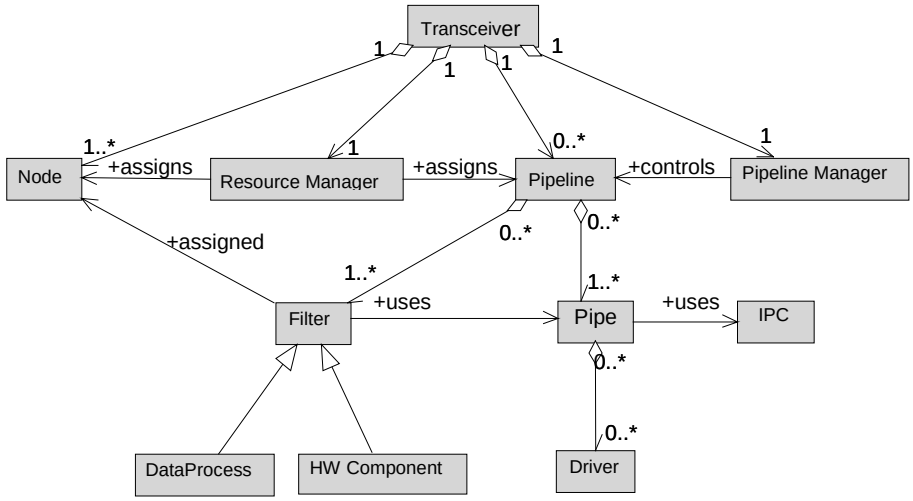


Figure 8-7. Process view metamodel.

The actual processes are shown in collaboration diagrams. All the control-related processes and the data processes that are closer to the higher-level software are situated in the Codec processor in Figure 8-8.

The difference between the two candidates was mainly in their way of handling the configurations. The first candidate has a more static approach. When a connection is requested, a pipeline is created for that connection. The pipes and filters are reserved for that pipeline until the release of that connection is requested. The second candidate differs from the first one in that the implementation of a service is decided upon when the service is called. The implementation depends on the current status of the system.

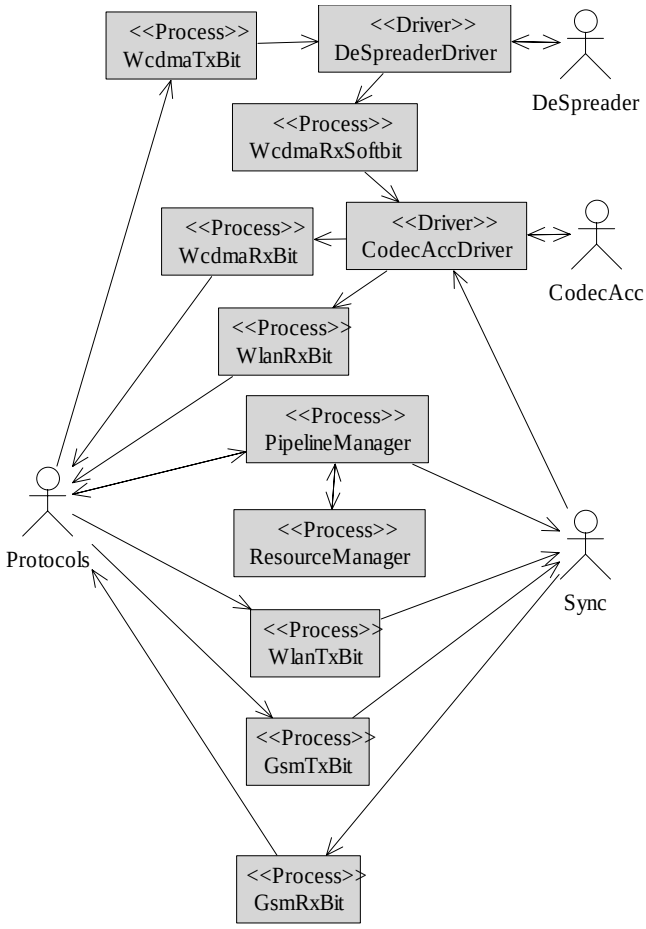


Figure 8-8. Collaboration of processes in the Codec processor.

The functionality of the process view was validated using scenarios similarly to the logical view. In order to keep the views in consistence, the process view components were mapped to the logical view components. The main strategies followed in the process view were:

1. Processor utilisation: Minimise context switches. If there is no specific reason to make a separate process for something then it is not done.

2. Latency: Critical functionality is mapped to a separate process. When some functionality needs a fast response, it is not combined with a less critical functionality.

3. Latency and memory usage: Avoid copying data and parameters. Shared memory is used for delivering control parameters and data.

4. Memory usage: Minimise the number of active processes. It is possible to remove unused processes in order to save memory space, if needed.

5. Power utilisation: Allow shutting down unused resources. It is possible to shutdown the unused nodes and also the unused blocks of memory.

6. Scalability: Allow a dynamic addition and deletion of connections and a separation of uplink and downlink.

8.3.4 Development view

Problems of the development view should be derived from the features of the development environment, i.e. hardware platform, development organisation, etc. Because there was no assumption of the development organisation in Multics, it could not be taken into account. The following questions are answered with the development view:

1. How is the complexity of hardware hidden from the application programmer?

2. How is control and data processing separated?

3. How should one handle the components of different modes?

4. How is the distribution hidden?

5. How are the layers separated from each other?

6. How are the modifications restricted to only few components?

7. What are the layering criteria?

The domains in the development view are illustrated in Figure 8-9.

- Applications. The application processes belong to this domain. They are independent of the underlying infrastructure. Therefore, they can be ported to different hardware and software platforms.

- Algorithms. This domain provides the algorithm library that is used by the Applications. The interface of the domain specifies the capabilities of the algorithms.

- Support. The communication methods and other operating system functionality are in this domain. This domain creates a software platform so that an application process does not need to know what is processed next and where.

- Devices. Hardware device drivers and interrupt handlers belong to this domain. It hides the actual composition of the hardware platform. Together with Support, it hides the infrastructure from the Applications.

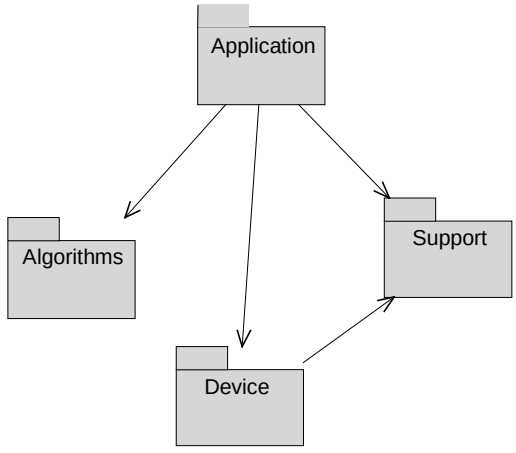


Figure 8-9. Development view.

The development view supports portability with using separate domains for hardware components and with using an operating system to hide processors. Extensibility is ensured with a specific domain to algorithms. Interfaces between domains are specified for facilitating modifiability. However, minimising the amount of sequential domains i.e. layers helps avoiding latencies.

8.4 Architecture evaluation

In the evaluation, two architecture candidates were compared. Their main difference was that they had different process views. The goal was to refine the architecture candidates, when necessary, according to the results of the evaluation. For convenience, the architecture candidate the process view of which was based on the pipes and filters style is called here the pipes and filters architecture and the other architecture candidate is called the blackboard architecture.

The performance impact analysis was performed using the execution time estimates from hardware architecture design for individual components. In addition, the time spent in control operations was estimated. RMA was used manually for analysing the performance. Estimates for processor utilisation, bus utilisation, memory usage, latencies and power consumption were prepared for each of the scenarios in the performance profile. The results were analysed in order to find out the points that needed changing. For example, the following changes were proposed to the original version of the pipes and filters architecture:

- Maximum power utilisation should be lowered for the Sync processor. Consequently, the following changes could be made: enable dynamic processes, remove inactive processes, rearrange processes, and shutdown the hardware parts when they are not active.

- Shorten latencies. The following changes are needed: add processing power, increase parallelism by using shorter periods, or rearrange the processes.

- Reduce memory usage of buffers. This means that the periods of processes should be shortened.

The cost impact analysis was difficult to perform because there was no estimation of the organisation that would develop the system. Therefore, relative values were used in the analysis. The results of the analysis could be used only for comparing the architecture candidates.

In the variability impact analysis, the impact of each change scenario was analysed. In addition, the scenario interaction analysis was used for covering the dependencies between the scenarios. The following types of changes were proposed for the initial version of the pipes and filters architecture:

- Both processors had their own drivers for hardware nodes. If processor allocation is changed, drivers, consequently, will need changing. There should be either one driver/node or one driver/HW component.

- Every time a new type of algorithm is added to the system, the selection of the algorithm has to be made in the data process. The Controller could give the pointers to the used algorithms and the conditions in the initialisation; or, one could use a dynamic inheritance pattern (Cohen 1996).

- Control process is too complicated. Therefore, it is divided into Resource manager and Pipeline manager.

In the tradeoff analysis, each of the proposed changes was studied and the impact of the proposed changes to the other attributes was analysed, Table 8-3. In accordance with the results of the tradeoff analysis, the following refinements were made for the original version of the pipes and filters architecture:

- Control process was divided into Resource and Pipeline managers to reduce the complexity of control.

- Dynamic processes were enabled for allowing updating for future purposes.

- One driver/HW node was selected for clear division of responsibilities. As a consequence there is no need for special assumptions of HW capabilities. It is assumed that HW components are fixed to the node they are now assigned.

The analysis of the pipes and filters architecture was partly redone after the refinements after which it better fulfilled the requirements.

Table 8-3. Examples of the result of the tradeoff analysis.

Result of impact analysis	**Change request**	**Tradeoff**
Performance		
Minimise power utilisation in some states by moving operations to one processor and putting the other processor in power-saving state.	Enable dynamic processes, remove inactive processes, re-arrange processes and shutdown the hardware parts when they are not active.	The use of dynamic processes in "a static way" does not cause any additional overhead and therefore can be accepted. The only problem is if such a RTOS can be found for DSP.
Shorten latencies.	Increase parallelism by finding out the optimum period.	Does not affect other quality attributes but causes increase in buffer utilisation if periods are lengthened.
Variability		
The same encoding is used in two processes.	Move the encoding to a separate process.	Too much overhead compared to the execution time of the encoding.
Control process is too complicated	It is divided into two processes. Resource management is separated to a different process.	Adds a little bit to the execution time but because it is not critical functionality it is ok.

In comparing the final designs, the pipes and filters architecture was considered to be the better alternative with the given requirements. The reason for this is the slightly better result in the cost evaluation and the fact that the performance of

the blackboard architecture is too sensitive to changes in process partition and periods. If an emphasis were laid on the importance of adaptivity so that the implementation platform of an algorithm should be selected for each call separately, the blackboard architecture would be a valid option.

The summary of the selected and alternative solutions to the design problems is presented in a design rationale. Part of the design rationale for the process view is shown in Table 8-4.

Table 8-4. Solutions to Process view problems.

Problem	**Selected solution**	**Explanation**	**Alternative solutions**
How are data processing tasks arranged?	Pipes and filters style	Strategy: Allow a dynamic addition and deletion of connections.	Proposed: blackboard style, if the system has to be able to select between different alternative implementations to a service during runtime.
How is the control of data processing arranged?	Main control and resource management are in separate processes.	Strategy: Use a separation of concerns	Removed solution: one Controller for everything because it would have been too complex to maintain.

The evaluation shows that even with inadequate knowledge of the system in the early phases of the development, the architecture evaluation can be used for comparing design decisions. The openness of the process enables the same tests to be repeated when more accurate knowledge is available. In the end, even the final, implemented product can be evaluated in order to find out whether the initial assumption applies and the selected architecture still fulfils its requirements. This evaluation was done manually; however, when there are larger evaluation profiles and several iterations of the architecture candidates, good tools are necessary.

8.5 Discussion

Two candidates for Transceiver architecture were developed. Although the two architecture candidates were mainly separated by the used architectural style in the process view, this is only one solution to one design problem. In fact, more than one architectural styles or patterns could have been used. Furthermore, in each view could be used different styles and patterns as long as they do not conflict with each other. Use case diagrams were utilised for validating the functionality with scenarios. They also served as a way to link the different views together. The design decisions were supported by the evaluation results. For supporting the future development of the system, the design decisions were explained in a design rationale. Furthermore, the alternative solutions that had been already considered were also documented. The architecture candidate based on the pipes and filters style proved to be the more appropriate base for a multimode Transceiver architecture. However, if other hardware architecture platforms, organisation and other environmental factors, and more detailed quality requirements were studied, refinements would be needed.

Software architecture design creates a bridge from the requirements to the detailed design. It is a basis for creating a system that fulfils its requirements, not only the functional but also non-functional ones. Furthermore, it facilitates a discussion on what the requirements actually are. The architecture design ensures that the major design decisions are made first, before detailed design, and not only after problems are found in testing.

The quality requirements specification should be emphasised in DSP software architecture development. It is the possible future needs, in particular, that should be considered. Without a proper understanding of the quality requirements, the comparison of design decisions becomes impossible. Generation of evaluation profiles would need the co-operation of different experts and stakeholders in order to be sure that everything is covered. However, it is important that only the critical scenarios are taken into the final profiles to keep the evaluation as fast as possible. Especially, the performance profile should include several scenarios, which reflect the different conditions in which the system is used.

Evaluation profiles enable the repetition of evaluations from the early architecture drafts to the implemented version of the system. When requirements change or new, more accurate values for estimations are received, or other assumptions do not hold any more, it is a signal that the architecture should be re-evaluated in order to find out if changes are needed.

The greatest advantage of the documentation of the architecture with views is that the different aspects of the design are explicitly shown somewhere. When the design decisions are explained, they can be truly followed and utilised in the later phases of the development. Furthermore, the experience gathered in one project can be used as a starting point in the following projects. This way although the people change in the organisation, the knowledge remains.

Design rationale covers the alternative design decisions. If there is a need to change the architecture, the already covered solutions are easy to find out and utilise, when possible. It is not necessary to write down the flow of each version of a candidate in detail. An informal design history (like a version list) could be enough for collecting all the changes done to the candidate. In the end, using the design history, a more formal summary of the significant design decisions and generations could be written down to the architecture document that is kept up-to-date and reviewed. Thus, the design history would not be reviewed or maintained otherwise but only when adding new “versions”.

The architecture design of complex systems is an iterative process and therefore, in the future, more emphasis should be put on developing easier and faster evaluation methods. For example, when there is more than a couple of performance scenarios, the evaluation would be much easier with a schedulability analyser

This work functions as a start in defining an architecture evaluation and refinement process for multimode DSP software. The purpose has been to give a general impression of the way quality requirements can be taken into account in the architecture design. A summary of the advantages of the proposed architecture development approach is presented as follows:

- Architectural views provide a way to manage complexity. An understanding of the overall system functionality grows. It is easier to find out the specific aspects of the system when the views are separated.

- The defined views support multimode operation. The logical view defines the different combinations of functionality that can be simultaneously active. The physical view defines all the possible mappings from logical components to the hardware architecture. The process view defines the actual runtime configurations. The development view supports the development of mode-specific and common components.

- In addition to the functional requirements, the architecture is derived from the quality requirements through applied strategies.

- Concurrent development is supported. Communication is easier when there is a common understanding of the system before detailed design starts. Development view defines the interfaces between different domains. An individual designer can find the module under work in the architecture and know its place in the system.

- Hardware/software partition is supported with two views. The required functionality is defined in an implementation-independent way in the logical view. The physical view is a communication media between hardware and software designers.

- The logical view provides the constraints and the possibilities for reuse. The actual work is done in the development view.

- The focus stays in the algorithm development. The logical view separates the control and the algorithms. In addition, the development view separates the algorithm development from the application development.

The pitfalls in the architecture development process are as follows:

- Use cases become the architecture design. If use cases go into too many details, they will start to restrict the design decisions more than is necessary.

- Not all the critical scenarios are covered in the evaluation profiles. The profiles are the main factor to the success of the evaluation.

- Too many scenarios are defined in the profiles. If the profiles become too large, the evaluation does not focus on the critical points, and furthermore, the evaluation becomes too cumbersome and time-consuming to be helpful.

- Academic discussion on notations and methods destroys the discussion on the actual contents of the views. The way views are described is not important as long as everybody in the project understands and accepts it. The guidelines concerning what diagrams are drawn and with what method should be established before the project starts.

- The architecture tries to solve all the problems at once. The architecture should concentrate on the most critical design decisions - detailed design is left to others.

- Evaluation of the architecture does not continue after the first version is released. There are many uncertain elements in the beginning of the project. There are many estimated values, in particular, that are used in the architecture evaluation. As soon as correct values or better values are received, the validity of the architecture should be re-evaluated.

- The detailed design is not in conformance with the architecture. The architect should make sure that the designers follow the guidelines that the architecture has given. If changes to the architecture are considered necessary in the detailed design, the architect should receive the change request. The architect then analyses the architecture if the change is really needed. The change to the architecture needs to be accepted before it can be done at the implementation level.

- Too much documenting frustrates the designers. Although there are certain things that should be written down, the purpose of architecture documentation is to provide a quick view of the properties of the system. The final formal documentation is different from what is done during the work. During the architecture design, the design decisions and evaluation results should be written down in an informal and fast way. A sort of a diary

should be kept. After the design starts to be more stable, the informal notes are generated into architecture documents and evaluation reports.

8.6 Future research

Because applying architecture design methods and techniques in DSP software design is a new subject, there is still much to do. The quality attribute taxonomies can be used as checklists in different points in the architecture development. In order to be reliable, they should be refined according to the experience that is received from several real-life projects.

In this work, the tools were not the main concern; in the future, however, it will be essential to find practical, useful tools for the evaluation process. Architecture evaluation will be used, as it should, only when its execution is fast and reliable. Therefore, the next step after this work will be to refine and polish the evaluation strategy and find tools that support it.

This research has concentrated on studying the qualities from the software point of view. However, the quality of a DSP system also depends on the hardware. Therefore, one interesting subject for future research is to broaden the quality analysis to cover both hardware and software. Traditionally, the hardware has guided the system design. However, more and more of the DSP systems are implemented in software. It would be about time for the software designers to take a more active role in the system design. One way to do this would be to take the ideas from the software architecture community to system architecture design.

9. Conclusions

This thesis has presented an approach to multimode DSP software architecture development. It was argued that the architecture development should be based on the quality requirements. Using the experience in DSP software architecture projects and literature, three quality attributes were defined for multimode DSP software. The different aspects of the attributes were clarified by using taxonomies. The architecture development process was divided into requirement analysis, selection of the architectural structures, and evaluation of the architecture candidates. The taxonomies were the starting point in all the three phases.

The approach is based on the utilisation of existing methods and techniques in the software architecture research field. During the late 1990's, software architecture research has evolved, and methods and techniques have been developed for each phase of the architecture development. This research combines the methods that are suitable for the multimode DSP software development. Digital signal processing software has long concentrated on saving the hardware resources and reaching the signal quality requirements. Now, when the processors and compilers finally enable the use of higher level languages in the implementation, and the complexity of software grows with multimode requirements, the focus has been turned also on the architecture design.

The research aimed at defining the architecture development approach that met the special requirements of the multimode DSP software. The research questions were the following:

1. What are the quality attributes of the multimode DSP software?

2. What are the architectural descriptions that should be used to specify multimode DSP software architecture?

3. How do quality attributes affect the selection of the architectural structures?

4. How are quality attributes used in the evaluation of architecture candidates?

The first question was answered by defining the quality attribute taxonomies. Based on the problem analysis, the critical quality attributes to multimode DSP software are performance, cost, and variability.

As an answer to the second question, four views are needed to define multimode DSP software architecture. The logical view divides the required functionality into implementation independent components. The physical view summarises the hardware architecture from the software point of view and also shows the external connections of the software. The process view shows how software works during runtime. And finally, the development view presents the actual modules that should be implemented. Each view can consist of several diagrams.

The third question is dealt with by means of quality taxonomies. The quality attribute taxonomies are used in every phase of the architecture development. Problems and strategies are defined on the basis of the taxonomies for each view. Strategies guide the selection of the architectural structures. The external environment and the quality measures affect the defining of the critical questions in the architecture design. Architectural parameters in the taxonomies point out what kind of strategies should be created for solving these problems.

The fourth problem, the evaluation, is supported by the taxonomies when creating the evaluation criteria and when specifying the actual measures that should be evaluated during the impact analysis. Evaluation profiles are created for each of the attributes. A stimulus section contains the variables that create the space where scenarios should be elicited. A response section is used for restricting the scenarios only to the most critical ones. It is possible to compare different architecture candidates by using the evaluation results; consequently, the one that best fulfils the requirements can be chosen.

The proposed architecture design approach is iterative and incremental. It is based on the existing methods and tools in the software architecture research field. The main idea is that without actually specifying the quality requirements, it is not possible to compare design decisions and to know that the developed system will be of the kind pursued. It was shown that even with an inadequate knowledge of the system in the early phases of the development, the quality evaluation can be used for comparing architectural decisions.

References

Abowd, G., Bass, L., Clements, P., Kazman, R., Northrop, L., Zaremsky, A. 1997. Recommended Best Industrial Practice for Software Architecture Evaluation, Technical Report CMU/SEI-96-TR-025. Software Engineering Institute, Carnegie Mellon University. 43 p.

Akao, Y. 1990. Quality function deployment: integrating customer requirements into product design. Cambridge: Productivity Press. 387 p. ISBN 0-915-29941-0

Allen, R., Garlan, D. 1997. A Formal Basis for Architectural Connection. ACM Transactions on Software Engineering and Methodology. Vol. 6, No. 3, pp. 213-249.

Alonso, A., de la Puente, J. A. 1993. Dynamic Replacement of Software in Hard Real-Time Systems. Proceedings of the Fifth Euromicro Workshop on Real-Time Systems, Oulu, Finland, June 22-24, 1993. Pp. 76-81.

Anthony, R.J. 2001. A Taxonomy of Transparency and a Dependency Graph. Proceedings of the IASTED International Conference, Applied Informatics, February 19-22, 2001, Innsbruck, Austria. Pp. 692-699.

Bachmann, F., Bass, L., Chastek, G., Donohoe, P., Peruzzi, F. 2000. The Architecture Based Design Method, Technical Report CMU/SEI-2000-TR-001. Software Engineering Insitute, Carnegie Mellon University. 56 p.

Barbacci, M., Klein, M.H., Longstaff, T.A., Weinstock, C.B. 1995. Quality Attributes. Technical report CMU/SEI-95-TR-021. Software Engineering Institute, Carnegie Mellon University. 56 p.

Barbacci, M.R., Ellison, R.J., Weinstock, C.B., Wood, W.G. 2000. Quality Attribute Workshop Participants Handbook, Special Report CMU/SEI-2000-SR-001. Software Engineering Institute, Carnegie Mellon University. 44 p.

Bass, L., Clements, P., Donohue, P., McGregor, J., Northrop, L. 2000. Fourth Product Line Practice Workshop Report, CMU/SEI-2000-TR-002. Software Engineering Institute, Carnegie Mellon University. 36 p.

Bass, L., Clements, P., Kazman, R. 1998. Software Architecture in Practice. Reading, Massachusetts: Addison-Wesley. 452 p. ISBN 0-201-19930-0

Bass, L., Kazman, R. 1999. Architecture-Based Development, Technical Report CMU/SEI-99-TR-007. Software Engineering Institute, Carnegie Mellon University. 36 p.

Bass, L., Klein, M., Moreno, G. 2001. Applicability of General Scenarios to the Architecture Tradeoff Analysis Method, Technical Report CMU-SEI-2001-TR-014. Software Engineering Institute, Carnegie Mellon University. 65 p.

Bellay, B., Gall, H., Hassler, V., Klösch, R., Trausmuth, G., Beckaman, H., Eixelsberger, W. 1997. Software Architecture through Architectural Properties, Technical Report TUV-1841-97-03. Technical University of Vienna. 21 p.

Bengtsson, P., Bosch, J. 1999. Architecture Level Prediction of Software Maintenance. Proceedings of Third European Conference on Software Maintenance and Reengineering, Amsterdam, Netherlands, March 1999. Pp. 139-147.

Bhattacharyya, S., Murthy, P., Lee, E. 1999. Synthesis of Embedded Software from Synchronous Dataflow Specifications. Journal of VLSI Signal Processing, Vol. 21, No. 2, pp. 151-166.

Bieman, J.M., Kang, B-K. 1998. Measuring Design-Level Cohesion. IEEE Transactions on Software Engineering, Vol. 24, No. 2, pp. 111-124.

Boehm, B., Abts, C., Chulani, S. 2000. Software Development Cost Estimation Approaches - A Survey, Technical Report USC-CSE-2000-505. USC Center for Software Engineering. 45 p.

Boehm, B., In, H. 1996. Identifying Quality-Requirement Conflicts. IEEE Software, Vol. 13, No. 2, pp. 25-35.

Bosch, J. 1998. Evolution and Composition of Reusable Assets in Product-Line Architectures: A Case Study. 1sr Working IFIP Conference on Software Architecture, October 1998. 13 p.

Bosch, J. 2000. Design and Use of Software Architectures, Adopting and evolving a product-line approach. Pearson Education Limited. 354 p. ISBN 0-201-67494-7

Bosch, J., Molin, P. 1999. Software Architecture Design: Evaluation and Transformation. Proceedings of the IEEE Engineering of Computer Based Systems Symposium (ECBS99), December 1999. Pp. 4-10.

Bot, S., Lung, C.-H., Farrell, M. 1996. A Stakeholder-Centric Software Architecture Analysis Approach. Proceedings of the Second International Software Architecture Workshop (ISAW-2), San Francisco, California, USA, Pp. 152-154.

Bratthall, L., Runeson, P. 1999. A Taxonomy of Orthogonal Properties of Software Architectures. Proceedings of the Second Nordic Workshop on Software Architecture (NOSA'99).

Briand, L., Carrière, J., Kazman, R., Wüst, J. 1998. COMPARE: A Comprehensive Framework for Architecture Evaluation. Technical Report IESE-046.98/E, November 1998. Fraunhofer IESE. 14 p.

Briand, L.C., Langley, T., Wieczorek, I. 2000. A Replicated Assessment and Comparison of Common Software Cost Modeling Techniques. Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000), Limerick, Ireland, June 4-11. Pp. 377-386.

Briand, L.C., Morasca, S. 1996. Property-Based Software Engineering Measurement. IEEE Transactions on Software Engineering. Vol. 22, No. 1, pp. 68-85.

Briand, L.C., Morasca, S., Basili, V.R. 1999. Defining and Validating Measures for Object-Based High-Level Design. IEEE Transactions on Software Engineering, Vol. 25, No. 5, pp. 722-743.

Briand, L.C., Wüst, J. 2001. Integrating scenario-based and measurement-based software product assesment. The Journal of Systems and Software, Vol. 59, No. 1, pp. 3-22.

Buschmann, F., Meunier, R., Rohnert, H., Sommerland, P., Stal, M. 1996. Pattern-Oriented Software Architecture - A System of Patterns. Chichester: John Wiley & Sons. 457 p. ISBN 0-471-95869-7

Cohen, S. 1996. Dynamic Inheritance in C++. C++ Report, Vol. 8, No. 4, pp. 30-37.

Coplien, J., Hoffman, D., Weiss, D. 1998. Commonality and Variability in Software Engineering. IEEE Software, Vol. 15, No. 6, pp. 37-45.

Dobrica, L., Niemelä, E. 2000. A strategy for analysing product line software architectures. VTT Publications 427, URL: http://www.inf.vtt.fi/pdf/. Espoo: Technical Research Centre of Finland. 124 p.

Douglass, B.P. 1998. Real-Time UML: Developing Efficient Objects for Embedded Systems. Addison-Wesley. 365 p. ISBN 0-201-32579-9

Drew, N., Dillinger, M.M. 2001. Evolution Toward Reconfigurable User Equipment. IEEE Communications Magazine, Vol. 39, No. 2, pp. 158-164.

Dueñas, J.C., de Oliveira, W.L., de la Puente, J.A. 1998. A Software Architecture Evaluation Model. Proceedings of the Second International ESPIRIT ARES Workshop, Las Palmas, February, 1998, LNCS 1429. Springer Verlag. Pp. 148-157.

Egyed, A., Medvidovic, N. 1999. Extending Architectural Representation in UML with View Integration. Proceedings of the 2nd International Conference on the Unified Modeling Language (UML), Fort Collins, CO, October 1999. Pp. 2-16.

Gacek, C. 1998. Detecting Architectural Mismatches During Systems Composition, Doctoral Dissertation, Center for Software Engineering. Los Angeles, CA 90089: University of Southern California. 195 p.

Gamma, E., Helm, R., Johnson, R., Vlissides, J.O. 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA: Addison-Wesley. 383 p. ISBN 0-201-63361-2

Gomaa, H. 1993. Software Design Methods for Concurrent and Real-Time Systems. Addison-Wesley Publishing Company. 447 p. ISBN 0-201-52577-1

Gupta, D., Jalote, P. 1993. On-line Software Version Change Using State Transfer Between Processes. Software-Practice and Experience. Vol. 23, No. 9. Pp. 949-964.

Hauptmann, S., Wasel, J. 1996. On-line Maintenance with On-the-fly Software Replacement. Proceedings of the Third International Conference on Configurable Distributed Systems. IEEE, Annapolis, Maryland, May 1996. Pp. 70-80.

Hofmeister, C., Nord, R., Soni, D. 1999a. Applied Software Architecture. Addison-Wesley. 397 p. ISBN 0-201-32571-3

Hofmeister, C., Nord, R.L., Soni, D. 1999b. Describing Software Architecture with UML. Proceedings of the First Working IFIP Conference on Software Architecture. Kluwer Academic Publishers. Pp. 145-160.

Huotari, A. 1999. Signal Processing in Multi-Mode Radio Terminal. Licentiate Thesis. Department of Electrical Engineering, University of Oulu, Finland. 69 p.

Jaaksi, A., Aalto, J.-M., Aalto, A., Vättö, K. 1999. Tried & True Object Development, Industry-Proven Approaches with UML. Cambridge: Cambridge University Press. 343 p. ISBN 0-521-64530-1

Jones, D.W. 1991. Solving Timing Problems in ADA. Proceedings of Miller Freeman, Inc., Embedded Systems Conference, Santa Clara, California September 24-27, 1991. Pp. 242-257.

Kaikkonen, T. 1996. Improving DSP Software Development Practices. Diploma Thesis. Department of Electrical Engineering, University of Oulu, Finland. 114 p.

Kalavade, A., Lee, E. 1993. A Hardware/Software Codesign Methodology for DSP Applications. IEEE Design and Test of Computers, Vol. 10, No. 3, pp. 16-28.

Kang, K.C, Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S. 1990. Feature-Oriented Domain Analysis (FODA). Feasibility study. Technical Report CMU/SEI-90-TR-21. Software engineering Institute. Carnegie Mellon University. 147 p.

Kang, K.C., Kim, S., Lee, J., Lee, K. 1999. Feature-Oriented Engineering of PBX Software for Adaptability and Reuseability. Software - Practice and Experience. Vol. 29, No. 10, pp. 875-896.

Karhinen, A., Ran, A., Tallgren, T. 1997. Configuring Designs for Reuse. Software Engineering Notes, Vol. 22, No. 3, pp. 199-208.

Kazman, R., Abowd, G., Bass, L., Clements, P. 1996. Scenario-Based Analysis of Software Architecture. IEEE Software, Vol. 13, No. 6., pp. 47-55.

Kazman, R., Klein, M., Barbacci, M., Longstaff, T., Lipson, H., Carriere, J. 1998. The Architecture Tradeoff Analysis Method. Proceedings of the 4th International Conference on Engineering of Complex Computer Systems (ICECCS98), Monterey, CA. Pp. 68-78.

Klein, M., Kazman, R. 1999. Attribute-Based Architectural Styles. Technical Report CMU/SEI-99-TR-022. Software Engineering Institute, Carnegie Mellon University. 82 p.

Klein, M.H., Ralya, T., Pollak, B., Obenza, R., Harbour, M.G. 1993. A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems. Massachusetts: Kluwer Academic Publishers. 712 p. ISBN 0-7923-9361-9

Kruchten, P. 1995. The 4+1 View Model of Architecture. IEEE Software, Vol. 12, No. 6, pp. 42-50.

Kukkohovi, M. 1996. Software Architecture for a Dual Mode Cellular Phone, Diploma Thesis. Department of Electrical Engineering, University of Oulu, Finland. 57 p.

Lassing, N. 2001. Architectural-Level Modifiability Analysis, PhD Thesis. Vrije Universiteit. URL: http://www.cs.vu.nl/~nlassing/. 236 p.

Lassing, N., Bengtsson, PO., van Vliet, H., Bosch, J. 2002. Experiences with ALMA: Architecture-Level Modifiability Analysis. The Journal of Systems and Software, Vol. 61, No. 1, pp. 47-57.

Lassing, N., Rijsenbrij, D., van Vliet, H. 1999. The goal of software architecture analysis: confidence building or risk assessment. Proceedings of the 1st Benelux conference on state-of-the-art of ICT architecture. Amsterdam, Netherlands: Vrije Universiteit. 6 p.

Lee, E.A. 1999. Embedded Software - An Agenda for Research. ERL Tehnical Report UCB/ERL No. M99/63. Berkeley, CA, USA: University of Berkeley. 16 p.

Lu, W.W. 2000. Compact Multidimensional Broadband Wireless: The Convergence of Wireless Mobile and Access. IEEE Communications Magazine, Vol. 43, No. 11, pp. 119-123.

Luckham, D., Vera, J. 1995. An Event-Based Architecture Definiton Language. IEEE Transactions on Software Engineering, Vol. 21, No. 9, pp. 717-734.

Lung, C-H., Bot, S., Kalaichelvan, K., Kazman, R. 1997. An Approach to Software Architecture Analysis for Evolution and Reusability. Proceedings of CASCON'97. Toronto, ON. November 1997.

Maccari, A., Saridakis, T. 1999. Software Architecture in Industry: Misuse and Non-Use. Proceedings of the Second Nordic Workshop on Software Architecture, NOSA'99. University of Karlskrona/Ronneby.

Magee, J., Kramer, J. 1996. Dynamic Structure in Software Architectures. Proceedings of ACM SIGSOFT'96: Fourth Symp. Foundations of Software Engineering (FSE4). Pp. 3-14.

Magee, J., Kramer, J., Sloman, M. 1989. Constructing Distributed Systems in Conic. IEEE Transactions on Software Engineering, Vol. 15, No. 6, pp. 663-675.

McCabe, T.J. 1976. A Complexity Measure. IEEE Transactions on Software Engineering, Vol. 2, No. 4, pp. 308-320.

Medvidovic, N., Taylor, R.N. 2000. A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Transactions on Software Engineering. Vol. 26, No. 1, pp. 70-93.

Mehta, M., Drew, N., Niedermeier, C. 2001. Reconfigurable Terminals: An Overview of Architectural Solutions. IEEE Communications Magazine, Vol. 39, No. 8, pp. 82-89.

Mehta, N.R., Medvidovic, N., Phadke, S. 2000. Towards a Taxonomy of Software Connectors. Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000), Limerick, Ireland, June 4-11. Pp. 178-187.

Mitola, J. 1995. The Software Radio Architecture. IEEE Communications Magazine, Vol. 33, No. 5, pp. 26-38.

Mitola, J. 1999. Technical Challenges in the Globalization of Software Radio. IEEE Communications Magazine, Vol. 37, No. 2, pp. 84-89.

Moon, M.F. 1993. Guidelines for Tasking Design. Embedded Systems Programming. Vol. 6, No. 10, pp. 28-34.

Moriconi, M., Qian, X., Riemenschneider, R.A. 1995. Correct Architecture Refinement. IEEE Transactions on Software Engineering, Vol. 21, No. 4, pp. 356-372.

Mouly, M., Pautet, M-B. 1992. The GSM System for Mobile Communications. France: Published by the authors. 701 p. ISBN 2-9507190-0-7

Nord, R.L., Cheng, B.C. 1994. Using RMA for Evaluating Design Decisions, Position paper. Proceedings of the Second IEEE Workshop on Real-Time Applications, IEEE Computer Society, Washington D.C. USA, July 1994. Pp.76-80.

Ojanperä, T., Prasad, R. 1998. Wideband CDMA for Third Generation Mobile Communications. Boston: Artech House Publishers. 439 p. ISBN 0-89006-735-X

Oreizey, P., Taylor, R.N. 1998. On the role of software architectures in runtime system reconfiguration. Proceedings of the International Conference on Configurable Distributed Systems (ICCDS4), Annapolis, Maryland, May 4-6, 1998. Pp. 137-145.

Oreizy, P., Medvidovic, N., Taylor, R.N. 1998. Architecture-based runtime software evolution. Proceedings of the International Conference on Software Engineering 1998 (ICSE'98). Kyoto, Japan, April 19-25. Pp. 177-186.

Oshana, R. 1998. Guidelines for DSP Development. Embedded Systems Programming. Vol. 11, No. 10, pp. 58-71.

Oviedo, E.I. 1980. Control flow, data flow and program complexity. Proceedings of the IEEE COMPSAC, November 1980. Pp. 146-152.

Perry, D.E., Wolf, A.L. 1992. Foundations for the Study of Software Architecture. ACM Sigsoft, Software Engineering Notes, Vol. 17, No. 4, pp. 40-52.

Peters, K. 1999. Migrating to Single-Chip Systems. Embedded Systems Programming. Vol. 12, No. 4, pp. 30-45.

Petriu, D., Shousha, C., Jalnapurkar, A. 2000. Architecture-Based Performance Analysis Applied to a Telecommunication System. IEEE Transactions on Software Enigneering. Vol. 26, No. 11, pp. 1049-1065.

Prieto-Diaz, R., Freeman, P. 1987. Classifying Software for Reusability. IEEE Software, Vol. 4, No. 1, pp. 6-16.

Purhonen, A. 2001. Quality Attribute Taxonomies for DSP Software Design. Software Product-Family Engineering, 4th International Workshop, PFE 2001, Bilbao, ES, Oct, 2001. van der Linden, F. (ed.). LNCS 2290. Germany: Springer-Verlag. Pp. 238-247.

Purhonen, A. 2002. Using architectural views in DSP software development. Proceedings of the IASTED International Conference, Applied Informatics, International Symposium on Software Engineering, Databases, and Applications, February 18-21, 2002, Innsbruck, Austria. Pp. 97-102.

Schmidt, D.C. 1999. Wrapper Facade: A Structural Pattern for Encapsulating Functions within Classes. C++ Report, SIGS. Vol. 11, No. 2.

Segal, M. E., Frieder, O. 1993. On-the-fly Program Modification: Systems for Dynamic Updating . IEEE Software, Vol. 10, No. 2, pp. 53-65.

Shaw, M., DeLine, R., Klein, D., Ross, T., Young, D., Zelesnik, G. 1995. Abstractions for Software Architecture and Tools to Support Them. IEEE Transactions on Software Engineering, Vol. 21, No. 4, pp. 314-335.

Shaw, M., Garlan, D. 1996. Software Architecture: Perspectives on an Emerging Discipline. New Jersey, USA: Prentice-Hall. 242 p. ISBN 0-13-182957-2.

Smith, C.U. 1990. Performance Engineering of Software Systems. Reading, Massachusetts: Addison-Wesley Publishing Company. 570 p. ISBN 0-201-53769-9

Smith, C.U., Williams, L.G. 1993. Software Performance Engineering: A Case Study Including Performance Comparison with Design Alternatives. IEEE Transactions on Software Engineering. Vol. 19, No. 7, pp. 720-741.

Soininen, J-P, Purhonen, A, Rautio, T., Kasslin, M. 2001. Mobile multi-mode terminal: making trade-offs between software and fixed digital radio. In: Del Re, E (Ed.). Software Radio: tehnologies and services. London: Springer-Verlag, 2001. Pp. 237-249.

Soni, D., Nord, R., Hofmeister, C. 1995. Software Architecture in Industrial Applications. Proceedings of the 17th International Conference on Software Engineering. Seattle, Washington: ACM Press. Pp. 196-207.

Srikanteswara, S., Reed, J.H., Athanas, P., Boyle, R. 2000. A Soft Radio Architecture for Reconfigurable Platforms. IEEE Communications Magazine, Vol. 38, No. 2, pp. 140-147.

Strike, K., El Emam, K., Madhavji, N. 2001. Software Cost Estimation with Incomplete Data. IEEE Transactions on Software Engineering, Vol. 27, No. 10, pp. 890-908.

Stuurman, S., van Katwijk, J. 1998. On-Line Change Mechanisms, the Software Architectural level. Proceedings of the ACM SIGSOFT Sixth International Symposium on the Foundations of Software Engineering (FSE-6), Lake Buena Vista, Florida, USA, November 3-5. New York: ACM Press. Pp. 80-86.

Tracz, W., Coglianese, L., Young, P. 1993. A Domain-Specific Software Architecture Engineering Process Outline. ACM SIGSOFT, Software Engineering Notes. Vol. 18, No. 2, pp. 40-49.

Varshney, U., Vetter, R. 2000. Emerging Mobile and Wireless Networks. Communications of the ACM, Vol. 43, No. 6, pp. 73-81.

Wermelinger, M. 1997. A Hierarchic Architecture Model for Dynamic Reconfiguration. Proceedings of International Workshop on Software Engineering for Parallel and Distributed Systems. Piscataway, NJ: IEEE. Pp. 243-254.

Weyuker, E.J. 1988. Evaluating software complexity measures. IEEE Transactions on Software Engineering, Vol. 14, No. 9, pp. 1357-1365.

Vigder, M.R., Kark, A.W. 1994. Software Cost Estimation and Control. Technical Report of National Research Council of Canada, Institute for Information Technology, NRC No. 37116. 69 p.

Vihavainen, K., Marttila, A. 1998. High-Level Design of Embedded DSP Systems. Technical report 2-1998. Tampere University of Technology. 60 p. ISBN 951-722-998-4

Wirth, N. 1971. Program Development by Stepwise Refinement. Communications of the ACM, Vol. 14, No. 4, pp. 221-227.

Withey, J. 1996. Investment Analysis of Software Assets for Product Lines. Technical Report CMU/SEI-96-TR-010. Software Engineering Institute, Carnegie Mellon University. 60 p.

Xu, J., Kuusela, J. 1998. Analyzing the execution architecture of mobile phone software with colored Petri nets. International Journal on Software Tools for Technology Transfer, Vol. 2, No. 2, pp. 133-143.

Zitzler, E., Teich, J., Bhattacharyya, S. 2000. Multidimensional Exploration of Software Implementations for DSP Algorithms. Journal of VLSI Signal Processing Systems, Vol. 24, No. 1, pp. 83-98.

Published by

Vuorimiehentie 5, P.O.Box 2000, FIN–02044 VTT, Finland
Phone internat. +358 9 4561
Fax +358 9 456 4374

Series title, number and report code of publication

VTT Publications 477
VTT–PUBS–477

Author(s)
Purhonen, Anu

Title

Quality driven multimode DSP software architecture development

Abstract

Traditionally, DSP software development has concentrated on optimising the algorithms. The future wireless communication systems create challenges to the DSP software. In order to handle the new requirements, more emphasis has been placed on software architecture. This thesis examines the way quality driven architecture development can be applied to multimode DSP software. First, the main quality attributes for DSP software are defined. Performance ensures that the timing requirements are fulfilled, with simultaneously minimising the resource usage. Cost attribute ensures that the development of the system is affordable. Variability is for evaluating how well the architecture can adapt to changes that are required to the system during its lifetime.

It is proposed that the DSP software architecture should be described with four architectural views. A logical view shows the required functionality in an implementation independent way; a physical view depicts the deployment of logical components to the hardware architecture and the interfaces that are relevant to the software; a process view is used for understanding the runtime functionality of the system; a development view describes how the system is actually implemented with today's software platforms and technologies.

The process of developing the architectural views is iterative and incremental. More details are added to the diagrams when the development continues. View development is a series of iterations between refinement of architectural structures and evaluation of the decisions made. An evaluation strategy is presented for comparing architectural decisions against quality requirements.

The results are validated with a case study of a future multimedia terminal that supports three systems: GSM, WLAN, and WCDMA. It is shown that the quality-driven development clarifies the design decisions so that it is easier to compare and refine architecture candidates.

Keywords
software engineering, quality, design methods, analysis methods, wireless systems

Activity unit
VTT Electronics, Kaitoväylä 1, P.O.Box 1100, FIN–90571 OULU, Finland

ISBN	Project number
951–38–6005–1 (soft back ed.) 951–38–6006–X (URL:http://www.inf.vtt.fi/pdf/)	E2SU00041

Date	Language	Pages	Price
September 2002	English	150 p.	C

Name of project	Commissioned by
	The National Technology Agency (Tekes)

Series title and ISSN	Sold by
VTT Publications 1235–0621 (soft back ed.) 1455–0849 (URL: http://www.inf.vtt.fi/pdf/)	VTT Information Service P.O.Box 2000, FIN–02044 VTT, Finland Phone internat. +358 9 456 4404 Fax +358 9 456 4374

VTT PUBLICATIONS

459 Hakkarainen, Tuula. Studies on fire safety assessment of construction products. 2002. 109 p. + app. 172 p.

460 Shamekh, Salem Sassi. Effects of lipids, heating and enzymatic treatment on starches. 2002. 44 p. + app. 33 p.

461 Pyykönen, Jouni. Computational simulation of aerosol behaviour. 2002. 68 p. + app. 154 p.

462 Suutarinen, Marjaana. Effects of prefreezing treatments on the structure of strawberries and jams. 2002. 97 p. + app. 100 p.

463 Tanayama, Tanja. Empirical analysis of processes underlying various technological innovations. 2002. 115 p. + app. 8 p.

464 Kolari, Juha, Laakko, Timo, Kaasinen, Eija, Aaltonen, Matti, Hiltunen, Tapio, Kasesniemi, Eija-Liisa, & Kulju, Minna. Net in Pocket? Personal mobile access to web services. 2002. 135 p. + app. 6 p.

465 Kohti oppivaa ja kehittyvää toimittajaverkostoa. Tapio Koivisto & Markku Mikkola (eds.). 2002. 230 s.

466 Vasara, Tuija. Functional analysis of the RHOIII and 14-3-3 proteins of *Trichoderma reesei.* 93 p. + app. 54 p.

467 Tala, Tuomas. Transport Barrier and Current Profile Studies on the JET Tokamak. 2002. 71 p. + app. 95 p.

468 Sneck, Timo. Hypoteeseista ja skenaarioista kohti yhteiskäyttäjien ennakoivia ohjantajärjestelmiä. Ennakointityön toiminnallinen hyödyntäminen. 2002. 259 s. + liitt. 28 s.

469 Sulankivi, Kristiina, Lakka, Antti & Luedke, Mary. Projektin hallinta sähköisen tiedonsiirron ympäristössä. 2002. 162 s. + liiitt. 1 s.

471 Tuomaala, Pekka. Implementation and evaluation of air flow and heat transfer routines for building simulation tools. 2002. 45 p. + app. 52 p.

472 Kinnunen, Petri. Electrochemical characterisation and modelling of passive films on Ni- and Fe-based alloys. 2002. 71 p. + app. 122 p

473 Myllärinen, Päivi. Starches – from granules to novel applications. 2002. 63 p. + app. 60 p.

474 Taskinen, Tapani. Measuring change management in manufacturing process. A measurement method for simulation-game-based process development. 254 p. + app. 29 p.

475 Koivu, Tapio. Toimintamalli rakennusprosessin parantamiseksi. 2002. 174 s. + liitt. 32 s.

477 Purhonen, Anu. Quality driven multimode DSP software architecture development. 2002. 150 p.

Tätä julkaisua myy
VTT TIETOPALVELU
PL 2000
02044 VTT
Puh. (09) 456 4404
Faksi (09) 456 4374

Denna publikation säljs av
VTT INFORMATIONSTJÄNST
PB 2000
02044 VTT
Tel. (09) 456 4404
Fax (09) 456 4374

This publication is available from
VTT INFORMATION SERVICE
P.O.Box 2000
FIN–02044 VTT, Finland
Phone internat. +358 9 456 4404
Fax +358 9 456 4374

ISBN 951–38–6005–1 (soft back ed.)
ISSN 1235–0621 (soft back ed.)

ISBN 951–38–6006–X (URL: http://www.inf.vtt.fi/pdf/)
ISSN 1455–0849 (URL: http://www.inf.vtt.fi/pdf/)