Liliana Dobrica & Eila Niemelä

A strategy for analysing product line software architectures

TECHNICAL RESEARCH CENTRE OF FINLAND ESPOO 2000

VTT PUBLICATIONS 427

A strategy for analyzing product line software architectures

Liliana Dobrica & Eila Niemelä
VTT Electronics

TECHNICAL RESEARCH CENTRE OF FINLAND
ESPOO 2000

ISBN 951–38–5598–8 (soft back ed.)
ISSN 1235–0621 (soft back ed.)

ISBN 951–38–5599–6 (URL: http://www.inf.vtt.fi/pdf/)
ISSN 1455–0849 (URL: http://www.inf.vtt.fi/pdf/)

JULKAISIJA – UTGIVARE – PUBLISHER

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

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

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

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

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

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

Technical editing Leena Ukskoski

Otamedia Oy, Espoo 2001

Dobrica, Liliana & Niemelä, Eila. A strategy for analysing product line software architectures. Espoo 2000, Technical Research Centre of Finland, VTT Publications 427. 124 p.

Keywords software product line, analysis techniques and methods, scenarios, quality attributes, spectrometer controller

Abstract

The purpose of the architecture evaluation of a software system is to analyze the architecture to identify potential risks and to verify that the quality requirements have been addressed in the design. This research report addresses the issue of how to perform an analysis of software product line architectures. Throughout the chapters we will try to present a way of thinking founded on an analysis at the architecture level of the quality attributes with the purpose to initiate and maintain a software product line considering the quality as the main driver in the product line development. The analysis strategy is exemplified with a spectrometer controller product line, a case study where the product line is initiated in a revolutionary style, which is such that the product line architecture and its components are elaborated to match the requirements of all expected product line members.

In this report, we will present our original contributions in developing this significant and, at the same time, new research domain. In order to be able to discuss an analysis strategy for a product line architecture, it is a considerable advantage to have a good knowledge of the state of art and practice in the software architecture domain. One of our contributions is to extract the main concepts that are common to software architectures and to present what is specific for the product line approach in software development. The study in the first part of the report gathers together, for the first time to our knowledge, all of the important published software architecture analysis methods and attempts to compare them. This survey shows the state of the research at this moment, in this domain, by presenting and discussing eight of the most representative architecture analysis methods. The study represents a step towards defining a general product line architecture analysis strategy. In practice, we will simplify our analysis approach and the last part of this report focuses on our experiences with the product line architecture analysis of the spectrometer controller PL.

Contents

List of abbreviations

ABAS	Attribute-Based Architecture Style
ADL	Architecture description language
ALPSM	Architecture level prediction of software maintenance
ATAM	Architecture tradeoff analysis method
BUFMAN	Buffer management
CASE	Computer assisted software environment
CIS	Command Interface Subsystem
CPU	Central processing unit
DMA	Direct memory access
DRAM	Dynamic RAM
EEPROM	Electric erasable programmable read only memory
EGY	Energy spectrum
ESAAMI	Extended SAAM by integration
ESM	Energy spectrum mode
GQM	Goal-Question-Metrics
HD	Hard disk
HK	Housekeeping
MCS	Measurement Controller Subsystem
MMS	Memory management subsystem
MSC	Message sequence charts
NFR	Non-functional requirements
OBC	On-board clock
OS	Operating system
PL	Product line

PLA	Product line architecture
PMS	Parameter management subsystem
QFD	Quality Function Deployment
RAM	Random access memory
RMA	Rate Monotonic Analysis
RMEM	Read memory
SAAM	Scenario-based architecture analysis method
SAAMCS	Scenario-based architecture analysis method with complex scenarios
SAAMER	SAAM improvement for evolution and reusability
SAEM	Software architecture evaluation model
SBAR	Scenario-based architecture re-engineering
SCV	Scope, commonality, variability
SEC	Single event characterization
SIXA	Spectrometer instrument X-ray array
SPE	Software performance evaluation
SRAM	Static RAM
ST	Star tracker
TIM	Time interval mode
TOPSA	Taxonomy of orthogonal properties of software architectures
WCM	Window counting mode

1. Introduction

During the recent years many research efforts have focused on ensuring that the quality of a software product is addressed at the architectural design level [9]. Architecture represents the first asset in an architecture-centric development process and from this point of view analysis can reveal requirement conflicts and incomplete design descriptions from a particular stakeholder perspective. There are well-known advantages of the introduction of the architecture in the life-cycle development of a single software product [18]. Because the product line software development is a new approach in software engineering research, the analysis of product line architecture should play an important role.

1.1 Software quality analysis at the architecture level

One of the major issues in the software systems' development today is quality. The purpose of the evaluation is to analyze the architecture for the identification of potential risks, verifying that the quality requirements have been addressed [49]. The role of architecture in a life-cycle of a software product is very important, but other stages of the development process (e.g. detailed design, implementation) are important, too. It is recognized that it is not possible to get an exact measure of the quality attributes for the final software product based on the architecture model [13]. This would imply that detailed design and implementation represent a strict projection of the architecture. The aim of analyzing the architecture of a software system is to predict the quality of the system before it has been built, not to establish precise estimates but the main effects of an architecture [35].

The idea to predict the quality of a software product from a higher-level design description is not new. In 1972, Parnas [55] motivated the use of modularization and information hiding as means of high-level system decomposition to improve flexibility and comprehensibility. In 1974, Stevens, Meyers and Constantine [63] introduced the notions of module coupling and cohesion to evaluate alternatives for program decomposition. During the recent years the notion of software architecture has emerged as the appropriate level to deal with software quality. This is because the scientific and industrial communities have recognized that the software architecture sets the boundaries for the software qualities of the

resulting system [8]. As one of the first decisions made, the software architecture must be checked against quality requirements. Recent efforts on systematization of the implications of using design patterns and architectural styles contribute, frequently in an informal way, to the guarantee of the quality of a software design [20, 25].

More formal efforts are concentrated on ensuring that the quality is addressed on the architectural level. Various communities from the software metrics, scenario-based and attribute model-based analysts have developed their own methods and techniques. The software metrics community has used module coupling and cohesion notions to define predictive measures of software quality [18]. Other methods include a more abstract evaluation of how the architecture fulfils the domain functionality and non-functional qualities [35]. Instead of introducing metrics for predictive evaluation, they describe examples for performing a more qualitative or quantitative evaluation. Analysis methods based on scenarios could be considered mature enough, since they have been applied and validated for the last six years, but the development of evaluation methods of attribute model-based architecture is still ongoing. Future research is needed to develop systematic ways to bridge quality requirements of software systems with their architecture.

1.2 Introduction to software product line

Nowadays a new initiative is coming onto the scene - the software product line. There are different topics to consider regarding a software product line [21]. Business, architecture, process and organization related aspects are the most important keys to effective product lines. A product line approach of software development is attractive to most organizations due to the focus on a strategic reuse of both intellectual effort and existing artifacts such as requirements, architectures and components (Figure 1). In the context of multiple software products created in one company, it is recognized that the effort and costs could be reduced if a product line is considered.

From the business point of view, strategic software reuse improves multiple factors, which influence the achievement of fast, efficient, predictable, low-cost, high-quality production and maintenance. We can enumerate the ability to take

advantage of new products and new technology faster; the significant decrease of time-to-market because off-the-shelf components and commercial-off-the-shelf components (COTS) are ready to use; higher employee productivity, with the emphasis not on coding but on reusing and integrating. Celsius Tech is one of many examples of success [8].

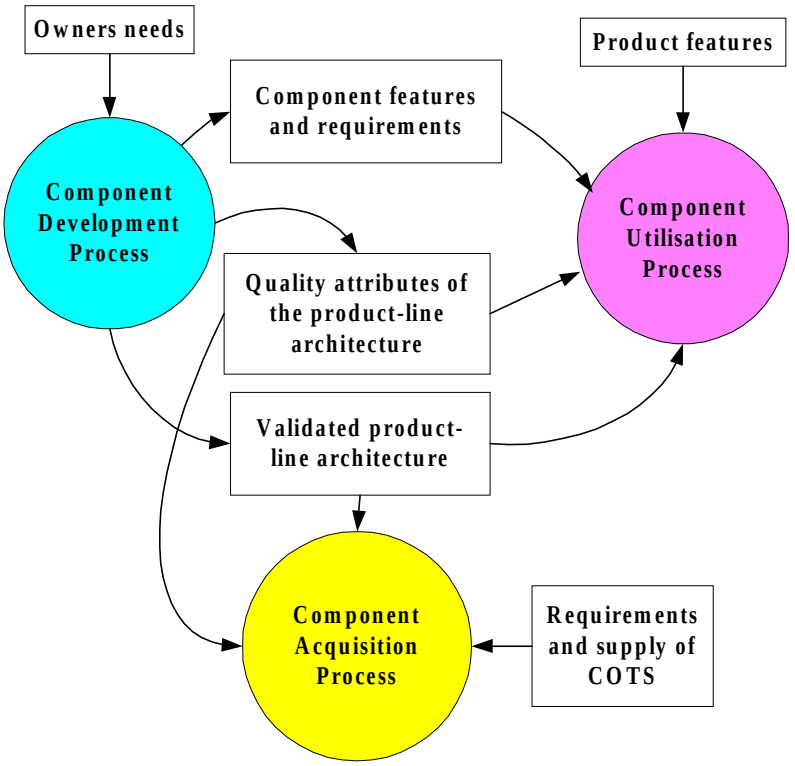

Figure 1. Relationships between product line architecture, quality attributes and software components.

The product line architecture is the main tangible element to consider in the context of software product line. All the PL members should share the same architecture. Under this new perspective the main question is what is the difference in analyzing PLA and the architecture for a single product? Are more architecture views or different evaluation techniques and methods needed?

The development process for a PLA is different than for one product [44]. The essential practice areas in the development of a single product include software

architecture modeling, representation, analysis, implementation in conformance with the specification, and traceability with requirements. In product line, additional practice areas such as generic and adaptable software architectures, reusable component design and development, traceability with common and variable requirements in a family of products are considered.

The goal of architectural analysis is to get measures of compliance with regard to requirement specification. In the case of product lines it is very important to identify, which are the relevant domain properties and how analysis techniques and methods could be applied to product lines. There are two categories of properties. The general one, like performance, satisfaction of real-time requirements, reliability, etc. and is specific to product lines. Among the specific points that deserve special attention are: *kinds of variation which can be covered by the architecture and properties that are preserved for all variants of an architecture, and stability of components' interfaces with respect to evolution in products.*

The open problem of a PLA analysis strategy is how to take better advantage of architectural concepts to analyze a software product line for quality attributes in a systematic way. The PLA must not only conform to quality requirements of each PL member, but it must also be generic and adaptable to the whole PL domain [19]. It is important to know how reusable and flexible PLA is to anticipated changes. The driving forces behind the development of a PLA must be reusability and modifiability. One objective of the evaluation is to estimate these two main structural qualities such as to maximize the reusability and to minimize possible changes in functionality required by various product members. It is also important to identify potential risks and to verify that the quality requirements of the PL domain have been addressed in the PLA design [41]. The analysis could be associated with the design in an iterative improvement of the PLA, when the PL is initiated in a revolutionary style, or for the re-engineering of an existent PL due to the PL evolution process [14].

1.3 Elements of software product line

A *software product line* is defined as a group of products sharing a common, managed set of features that satisfies specific needs of a selected market [8]. The products should be suitable to a market strategy and application domain. They share an architecture and are built from components.

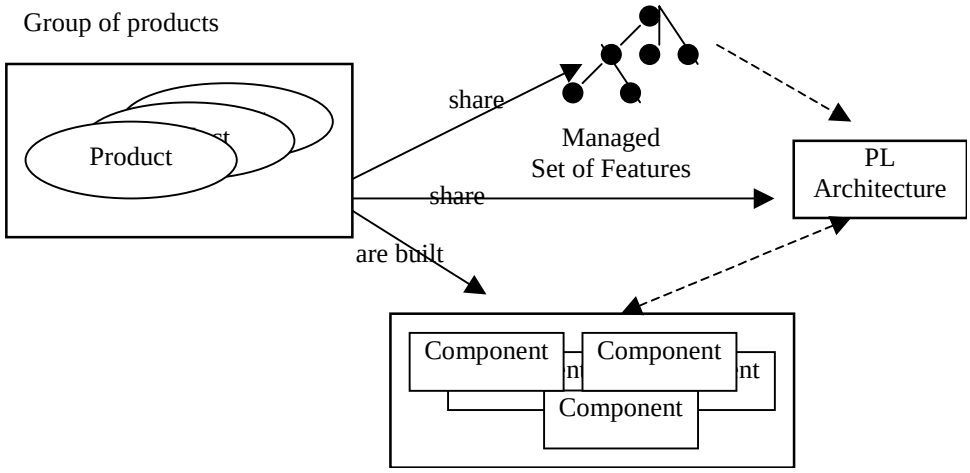

Figure 3. PL definition.

The PL definition includes three main concepts, which are feature, architecture, and component. Figure 2 describes the way they are related to give a unitary logic to the development of a PL. The common set of features represents the source of information for the shared architecture design, which is called product line architecture. From the viewpoint of decomposition and composition approaches of the architecture design methods, we can identify a bi-directional relation between PLA and the set of components [2]. In one direction of this relation, PLA influences component development considering the decomposition. The composition aspect acts in the opposite direction of this relation, and the set of components influences PLA development.

Some of the issues with PL are related to the process of initiation and how to deal with its evolution process.

Initiation. Software product line does not appear accidentally, but requires a conscious and explicit effort from the organization interested in using product line approach. Basically, one can identify two relevant dimensions with respect to the initiation process. The organization may take an evolutionary or a revolutionary approach to the initiation process. In each dimension, the product line initiation can be applied to an existing line of products or to a new product line that the organization intends to use. Each case has an associated risk level and benefits. For instance, in general, the revolutionary approach involves more risks, but then, higher returns compared to the evolutionary approach. [15]. Revolutionary approach to a new product line means that a product line architecture and components are developed to match the requirements of all the expected product line members, before developing the first product in a new domain.

Evolution. The research in software PL shows that the evolution of a PL is driven by possible changes in the requirements of existent members or the addition of new product members [14]. In the case of changes in the requirements of existent members, a typical way of handling this is to create two independent evolution cycles for each product. One cycle incorporates product-specific new requirements and the other is for the PLA as a whole, incorporating new requirements that affect all or most of the products in the PL.

A study of a PL evolution, which consists of two generations and four releases for each one, makes the definition of taxonomy of requirements and the analysis of evolution of PLA possible. Considering domain-specific software architecture, six categories of requirements have been identified. These are:

- The market needs which lead to the construction of a new product line;
- The addition of a new product to product line to improve the functionality;
- Improvement of the existing functionality with supporting new features like standards or user, requested;
- Extension of standard support incorporated in the new versions;

- A new version of hardware, operating system or third party component, which covers more functionality;

- Improvement of quality attributes.

PLA evolution may occur due to:

- A decision which states that a new set of products should be developed. In this case the PL could be spli using the existing PLA as a template for creating a new PLA or could be a derived PLA as a branch of the same PLA.

- A PLA component, which could be a new one with a new functionality, is changed, split or replaced to improve the quality attribute of PLA component.

- A new and changed relation between components as a consequence of the causes mentioned above.

2. Main concepts related to product line architecture

In this chapter we will present an overview of the main concepts that are relevant in the context of software product line architecture (Figure 4). The first part is concerned with the terminology related to the common denominator in software engineering development, which is considered the software architecture. The definition, description and design elements, views of software architecture, are introduced here. The second part describes the main software quality attributes and their relatioships to architectural views. The third part focuses on the particular concerns related to product line software development such as architectural styles and patterns. Initiation and evolution processes of software product line and also product line architecture specific description represent the main headlines of this third part. The fourth part focuses on the quality attributes especially critical in a PLA.

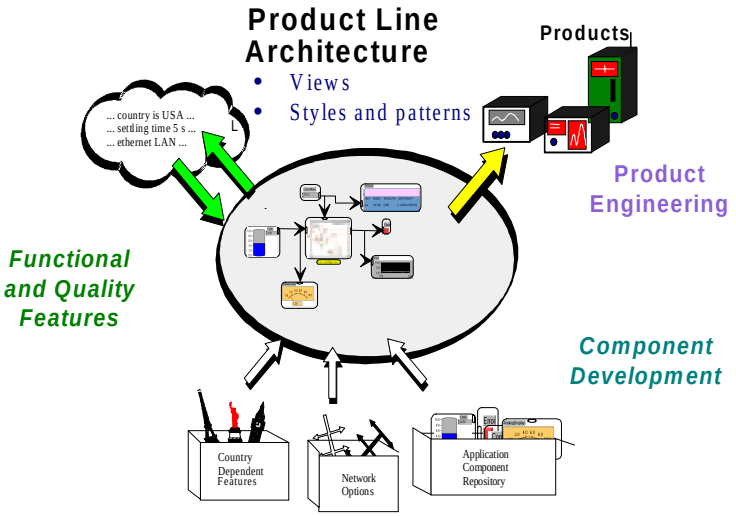

Figure 4. Main concepts of a software product line.

2.1 Definition of software architecture

A definition of software architecture is given in [8]. Here the software architecture of a program or computer system is defined as “the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them”. This definition focuses only on the internal aspects of a system and most of the analysis methods are based on this definition.

Another definition given by Garlan and Perry [56, 60] establishes software architecture as “the structure of components in a program or system, their interrelationships, and the principles and guides that control the design and evolution in time”. This process-centered definition is used when the evaluation model takes account of the presence of principles and guidelines in the architecture description.

The authors of [48] in their analysis method of flexibility have found that for the architectural analysis, the external environment is just as important as the internal entity of a system. Their opinion is that the definition of a software architecture should consist of two parts, namely of a macro architecture, which focuses on the environment of the system, and a micro architecture, which should cover the internal structure of a system.

2.2 Description of Software Architecture

2.2.1 Multiple views representation

The research in the architecture description is designed to address the different perspectives one could have of an architecture. Each perspective is described as a view. Although there is not yet any general agreement about which views are the most useful, the reason behind multiple views is always the same: *Separating different aspects into separate views help people manage complexity*. The information relevant to one view is different from that of others and should be described using the most appropriate technique for each view. Several models have been proposed that include a number of views that should be described in

the software architecture. For instance, the 4+1 View Model consists of the following views:

- the *logical view*, which is the object model of the design;
- the *process view*, which captures the concurrency and synchronization aspects of the system;
- the *physical view*, which describes the mapping of the software onto hardware and reflects its distribution aspect;
- the *development view*, which describes the static organization of the software in its development environment;
- the *scenario view*, or the +1 view, which illustrates the way the architecture satisfies the requirements and defines the relations among other views.

A similar set of views is distinguished in Hoffmeister's et al. (Figure 6). A *conceptual view* describes the system in terms of its major design elements and the relationships among them. A *module view* decomposes the system into modules and layers. An *execution view* describes how modules are mapped to the elements provided by the runtime platform, and how these are mapped to the hardware architecture. Lastly, a *code view* describes the mapping of runtime entities in the execution view to deployment components, the mapping of the modules from the module architecture to source components, and how the deployment components are produced from source components.

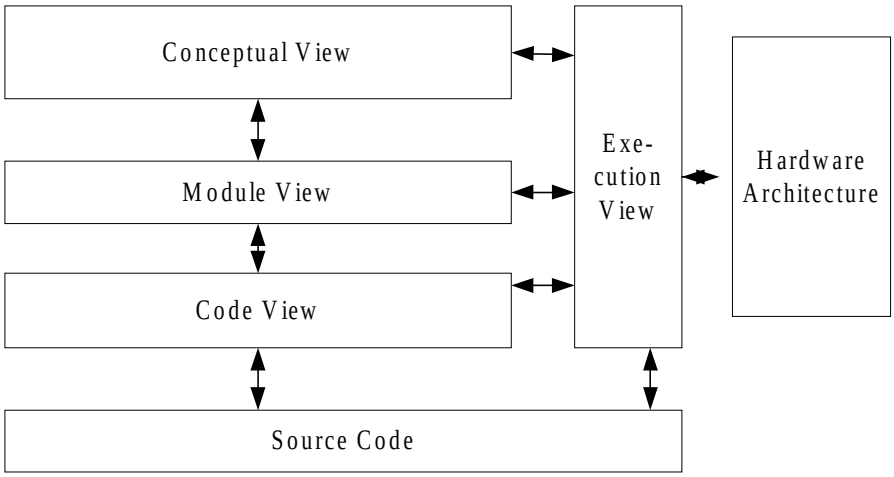

Figure 6. Views of architecture in Hoffmeister's et al. [27].

What the view models have in common is that they address the static structure, dynamic aspect, physical layout and development of the system. Bass et al. [8] introduce the concept of 'architecture structures' as a synonym to 'view'. In addition to four structures that are identical to the views in the 4+1 View model, the authors mention the *uses structure*, the *calls structure*, the *data flow*, the *control flow* and the *class structure*. In general, it is the responsibility of the architect to decide which view to use for describing the architecture.

An architecture modeling language represents software architecture. The main elements of a modeling language are components or elements and the relations among these. In order to predict any quality attribute based on the architecture model it is important to define a clear and precise terminology for these elements.

Components or elements. The elements of the architecture are almost always documented. Each element has a name, an interface, ports, and a behavior. The name serves the purpose of identification, the interface is the specification of what methods or services the architecture element provides for external use and subsequently must provide an implementation for. This information would be mapped into the header-files of C++ when implemented, but lacks some details compared to source code header files. For example, in the software architecture

documentation the signature of a method could be its name and potential other architecture elements parameter.

Static relations. The elements in the architecture have relations to other elements. These relations make up the form or the structure of the architecture. The relations must be documented by specifying, either in text or graphically, the related components (2 or more) and the relation type (association, aggregation, etc.). This information is hard to trace in the source code and will be obscured in the implementation of classes.

Dynamic relations. The dynamic aspect of the software architecture is important to determine its functions and qualities. The plausible way to document the dynamic aspect is to use *message sequence charts* for the method usage between architecture elements. Message sequence charts can help significantly in understanding the software without getting too deep into the details, which are left open for the detailed design and the implementation.

From the point of view of quality analysis at the architectural level, the possible representations could be very relevant in the quality prediction and effort estimation [33]. An evaluation method may need structures, which are concerned with:

- The decomposition of the functionality that the products need to support. Components are functional (domain) entities, and the connectors are ‘uses’ or ‘passes-data-to’ relations. This structure is useful for understanding the interaction between entities in the problem space, for planning functionality and for understanding the domain variability, and hence thereafter, the possibilities for creating a product line.

- The realization in a detailed design of the conceptual abstractions from which the system is built. Components can be packages, classes, objects, procedures, functions, methods, etc., all of which are vehicles for packaging functionality at various levels of abstraction. Relations include passes-control-to, passes-data-to, shares-data-with, calls, uses, is-an-instance of, etc. This structure could be crucial for understanding the maintainability, modifiability, reusability, and portability of the system.

- Logical concurrency. The components of this structure are units of concurrency that are ultimately refined to processes and threads. Relations include synchronizes-with, is-higher-priority-than, sends-data-to, can't-run-without, etc. Properties relevant to this structure include priority, preemptability and execution time. This structure is a key to understanding performance and is also important in reliability and security.

- Hardware including central processing units, memory, buses, networks, and input/output devices. Properties relevant to this structure include availability, capacity and bandwidth.

- Files and directories. This structure is important for managing and ensuring the administrative control of the system as it is fleshed out, including the division of work into teams (i.e. modularity) and configuration management.

2.3 The main software quality attributes

2.3.1 A software quality model

A software quality is defined in IEEE 1061 [30] and it represents the degree to which software possesses a desired combination of attributes (e.g. reliability, interoperability). Another standard ISO/IEC Draft 9126-1 [31] defines a software quality model. According to this model, there are six categories of characteristics (functionality, reliability, usability, efficiency, maintainability, and portability) which are further divided into sub-characteristics. These are defined by means of externally observable features for each software system. In order to ensure its general application, the standard does not determine which these attributes are, nor how they can be related to the sub-characteristics. The properties of the resulting system specify the quality features.

In a PL context, each software product member possesses a desired combination of quality attributes (e.g. performance, reliability, safety, security, etc.). The quality model of a PL domain must include the variability and commonality among these quality requirements and a PLA design should conform to these aspects. A quality-based PLA development must also prioritize the importance

of each quality attribute from both a domain and PLA perspective. The PLA perspective ranks reusability and modifiability high.

2.3.2 General software quality attributes

In this section the most important general quality attributes of software architecture are introduced. Generally, the quality of a software system is divided in two categories. The first one is the set of quality attributes that is observable at run-time, such as performance, functionality, usability, and the second one is the set of quality attributes that cannot be discerned at the run-time such as reusability or integrability [8]. The quality attributes of the latter set are also called non-functional qualities (NFR). Functionality is the ability of the system to do the work for which it was intended. It is orthogonal to structure, meaning that it is largely non-architectural in nature. The interest for the architecture is how it interacts with, and constrains, the achievement of other quality attributes (Table 1).

Non-functional qualities of a software system have a great impact on its development and maintenance, its general operability and its use of computer resources [51]. They have an equal impact on the software system, as have its functional properties. The larger and more complex a software system and the longer its lifetime, the more important its non-functional characteristics become.

Another form of expression for performance is the number of transactions per unit time or the amount of time it takes a transaction with the system to complete. In the case of distributed systems, performance is a function of how much communication and interaction there is between the components of the system. If all the components of the architecture are on the same processor then performance is a function of the amount of interaction by a subroutine invocation or it is referring to process synchronization. Performance analysis looks at the arrival rates and distributions of service request, processing times, latency and queue size if applicable.

Table 1. Main quality attributes of software architectures.

Quality attribute	***Description***
Performance	Responsiveness of the system, which means the time required to respond to stimuli (events) or the number of events processed in some interval of time.
Security	A measure of the system's ability to resist unauthorized attempts at usage and denial of service while still providing its service to legitimate users.
Availablity	Availability measures the proportion of time the system is up and running.
Reliablity	The ability of the system or component to keep operating over the time or to perform its required functions under stated conditions for a specified period of time.
Usability	Can be broken down into the following areas: • *Learnability*: How quick and easy is it for a user to learn to use the system's interface? • *Efficiency*: Does the system respond with appropriate speed to a user's requests? • *Memorability*: Can the user remember how to do system operations between uses of the system? • *Error avoidance*: Does the system anticipate and prevent common user errors? • *Error handling*: Does the system help the user recover from error? • *Satisfaction:* Does the system make the user's job easy?
Modifiability	The ability to make changes quickly and cost-effectively.
Maintainability	The ease with which a software system or component can be modified to correct faults, improve performance, or other attributes, or adapt to a changed environment.
Flexibility	The ease with which a system or component can be modified for use in applications or environments other than those for which it was specifically designed.
Scalability	The ease with which a system or component can be modified to fit the problem area.

Table 1. Main quality attributes of software architecture continues.

Quality attribute	***Description***
Portability	The ability of the system to run under different computing systems: hardware, software or combination of the two.
Reusability	Reusability means designing a system so that the system's structure or some of its components can be reused again in future applications.
Integrability	The ability to make the separately developed components of the system work correctly together.
Interoperability	A special case of integrability that measures the ability of a group of parts (constituting a system) to exchange information and use the one exchanged.
Testability	The ease with which software can be made to demonstrate its faults (typically execution based) testing.

Types of security threats could be: denial *of service* (by flooding the target with connection requests or queries), or *IP source address spoofing* (by assuming the identity of a host trusted by the target). Strategies against prevention, detection and response to an attack are: authentication server, network monitors; and "firewall", a system constructed on top of a trusted kernel that provides security services.

Availablity is equals to *mean_ time_ to_ failure/(mean_ time_ to_ failure + mean_ time_ to_ repair).* Designing components that are easy to modify and designing a component interaction scheme that helps to identify misbehaving culprits lowers *Mean_time_to_repair*.

Reliability is equals to *mean_time_to_failure*. *Mean_time_to_failure* is lengthened by making the architecture fault tolerant, by the replication of critical processing elements; by fielding a less error-prone system, which is addressed architecturally by a careful separation of concerns, which leads to better integrability and testability.

Modifications to a system can be divided in several categories of abilities. We can distinguish the ability to acquire new features, simplify the functionality of

an existent system, adapt to new operating environments, or restructure system services. A restructuring ability of the system leads to a decomposition of the system into modules, which in this way encourages the creation of reusable components. Maintanability is the same with modifiability, from the architectural point of view, but a fine distinction between the two terms has to do with the type of change being installed.

Reusability is a synonym to integrability if the system has been structured so that its components could be chosen from previously built products. It could be seen as a special case of modifiability.

Integrability measures the ability of parts of a system to work together. It depends on the external complexity of the components, their interaction mechanisms and protocols, and the degree to which responsibilities have been cleanly partitioned.

Testability refers to the probability that, assuming that the software does have at least one fault, the software will fail on its next test execution. It is related to the concepts of observability, to observe its outputs, and controllability, to control each component's internal state and inputs.

Some quality attributes, like reusability and modifiability, have similar architectural techniques for their achievement. It also happens that a similar overall purpose is achieved by multiple different quality properties, like portability and interoperability. Interdependencies and tradeoffs also exist between quality attributes. In case of conflict, an ordering priority between NFRs should be specified, or a preference of one NFR against another should be defined.

In the context of software product line modeling, variability is essential for building a flexible architecture. It is possible to anticipate some common and variable aspects in requirements of different product members and to construct a product member in such a way that it facilitates this type of variability. The driving forces behind the development of software product line are reusability and modifiability.

2.4 Mapping quality attributes and architectural views

A taxonomy of formally defined orthogonal properties of software architectures (TOPSA) that extends an architecture definition is given in [17]. The TOPSA space has three dimensions: *abstraction level* (conceptual, realization), *dynamism* (static, dynamic), and *aggregation level*. The TOPSA can facilitate discussions regarding software architecture during the development and evolution. This model makes a clear distinction between conceptual architecture and realization architecture, suggesting the creation of architecture models suited for different purposes and different stakeholders.

Syntactic architectural notations should be well understood by the parties involved in the analysis. Architectural descriptions need to indicate the system's computation and data components as well as all the relationships between components. The result of an architecture evaluation process depends on how well the description is made. TOPSA model and the architecture representation based on multiple views complement each other (Figure 8). Different views offer valuable examples for abstraction, dynamism and aggregation dimensions in TOPSA space.

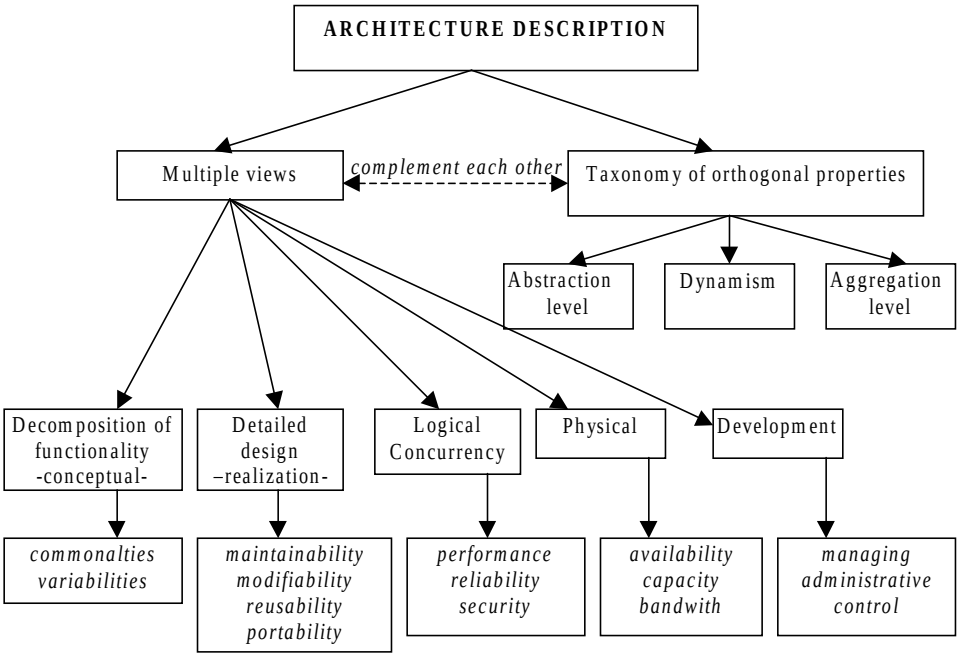

Figure 8. Architecture description and the relevance to the analysis of quality attributes.

Table 2 exemplifies the mappings of two quality attributes, performance and reusability, on the architecture views associated with a TOPSA space. For example, performance could be analyzed on the logical concurency view in a realizational plane of a TOPSA space.

Table 3. The mapping of quality attributes on architecture description.

Quality attribute	***Architecture View***	***TOPSA space (Abstraction, Dynamism, Aggregation)***
Performance	Logical Concurrency	Realizational
Reusability	Functional, Detailed design, Development	Conceptual/Realizational, Aggregation

An analysis method could exploit these relationships in the form of a defined set of rules, which states which view in a TOPSA space is the most appropriate for an analysis of a given quality attribute.

2.5 Architectural styles and patterns

An architectural style includes a description of component types and their topology, a description of the pattern of data and control interaction among the components, and an informal description of the benefits and drawbacks of using that style. In [60] and [20], several architectural styles are classified and described. Architectural styles are important since they differentiate classes of designs by offering experimental evidence of how each class has been used along with a fitness association for each software product property.

Architectural patterns are the building blocks of software architectures. An architectural pattern is different from an architecture style that is not predominant in the architecture. Architectural pattern is a defined/documented realization of a style or styles. The architecture of a complex system is likely to include instances of patterns composed in arbitrary ways.

A recent research report evaluates collections of architecture patterns in terms of both quality factors and concerns, and anticipations of their use [40]. The report introduces the feasibility of architectural patterns being ”pre-scored” to gain a sense of their relative suitability to meet the quality requirements of a software product. In addition to evaluating individual patterns, the report reveals that it is necessary to evaluate compositions of patterns that might be used in an architecture. Identifying patterns that are not composed well (the result is difficult to analyze or the quality factors of the result are in conflict with each other) should steer a designer away from “difficult” architectures, towards architectures made of well behaving compositions of patterns.

An *attribute-based architecture style* (ABAS) helps to move from the notion of architectural styles toward the ability to reason (whether quantitatively or qualitatively) based on quality attribute-specific models. The goals of having a collection of ABASes are to make architectural design more routine-like and more predictable, to have a standard set of attribute-based analysis questions, and to tighten the link between design and analysis. An ABAS is defined as having four parts:

- *Problem description*, which defines what problem is being solved by the structure.

- *Stimuli/Responses* are a characterization of the stimuli that the ABAS is designed to respond to, and the description of the quality attribute-specific measures of the response.

- *Architectural style* has a standard definition and the properties of the components or connectors that are relevant to the quality attribute.

- *Analysis* is a quality attribute specific model that provides a method of reasoning the behavior of component types that interact in the defined pattern.

Unfortunately, for the moment ABASes are not appropriate for any quality attribute. Quality attributes such as reusability or modifiability cannot be included, because there are no universal measures for them, only context dependent measures.

2.6 PLA specific description

Our definition of PLA is according to [8]. A PLA is a software architecture and a set of reusable components shared by a family of products.

The components are large pieces of software and are typically modeled as object-oriented frameworks that cover functionality common for the products in the product line and support the variability required for the various products.

An important aspect, which should be considered, is the component-based development of PLAs. A component-based architecture is a general solution for improving modifiability and reusability. Some quality attributes are domain-dependent such as performance, security and availability but reusability and modifiability are common for all product line architectures. At the conceptual level, PLA description is modeled by an abstract framework with component interconnections and a detailed representation of each contained component. The abstract framework could be considered a domain-specific architecture, which provides a global perspective of the group of products, while a component detailed representation is a micro-architecture, which represents a cluster of features.

Similar structures as for single product software architecture could be used for the representation of PLA. However, due to the presence of additional terms like scope, commonality and variability, it is possible that new views are to be introduced, such that facilitate the analysis of the PLA. Variability could be incorporated in each view of the architecture. For example, the conceptual view describes the components and their interconnection from the static point of view. In an abstract framework the variability could be expressed by not showing what is variable or by using specific relations and notations (aggregation, specialization, etc.) of the modeling language (UML for example). The usage of packages is a solution to express the diversity of specific component models. A package with specific concrete components is organized for each product member.

3. Methods and techniques for software architecture evaluation

In the previous chapter we presented the main concepts related to software product line architecture. In order to decide whether an architecture fulfils the quality requirements, it needs to be evaluated. In this chapter we will discuss a number of different approaches to architecture evaluation that we have found to be useful. The content of this chapter represents a surveillance study on the software architecture analysis methods. The role of the study is to put all these approaches in the same perspective by reviewing the state of art and practice in the research domain.

Generally, methods include a predefined and organized collection of techniques. In this perspective, we considered that it is important to present the classification of the evaluation techniques available at the architecture level, in the first part of this chapter.

The remainder of the chapter deals with the survey on the existent analysis methods. Different viewpoints that these methods reflect on the evaluation of the quality of software architecture make it very difficult to define a common framework of presentation. We will discuss the analysis methods trying to look for 1) their progress towards refinement over the time, 2) their main contributions, and 3) advantages obtained by using them. The discussion about the selected methods is focused on 1) discovering differences and similarities between eight available methods, and 2) making classifications, comparisons and appropriateness studies.

At the end of this surveillance work, we will draw some conclusions of the real level of the current research as well as the future work in this domain area defined by the presented methods. This study represents an important step towards defining a strategy for an analysis of software product line architecture.

3.1 Evaluation techniques

3.1.1 A classification of the evaluation techniques

Evaluation techniques, which could be included in architecture analysis methods, are portrayed in the reference papers [1][5]. These techniques are divided in two basic classes: *questioning* and *measuring techniques*. The first category generates qualitative questions to ask about an architecture and can be applied to evaluate an architecture for any given quality. Questioning techniques include scenarios, questionnaires and checklists.

Measuring techniques suggest quantitative measurements to be made on an architecture. They are used to answer specific questions and address specific software qualities, and therefore, they are not as broadly applicable as questioning techniques. This category includes metrics, simulations, prototypes and experiences. Metrics are quantitative interpretations placed on a particular observable measurement on the architecture, such as fan in/fan out of the components.

A comparison between the existent evaluation techniques is presented in [8]. Generality, level-of-detail and phase are included in a multi-dimensional framework of comparison. Regarding generality, the techniques could be general (questionnaire), domain-specific (checklists, prototype), or system-specific (scenarios). Level of detail (coarse-grained, medium or fine) indicates how much information about the architecture is required to perform the evaluation. There are three phases of interest to architecture evaluation: early, middle and post-deployment. The early phase evaluation occurs after initial high-level architectural decisions have been made, the middle phase can occur at any point of the iterative elaboration of the architecture design.

In terms of quantitative and qualitative aspects, both classes of techniques are needed for evaluating architectures. Various modeling and analyzing models expressed in formal methods are included in the set of quantitative techniques. Most often, qualitative techniques illustrate software architecture evaluations based on scenarios. Scenarios are rough, qualitative evaluations of architecture. Scenarios are necessary but not sufficient to predict and control quality attributes, and have to be supplemented with other evaluation techniques. For

example, including questions about quality factors in the scenarios enriches the results of architecture evaluation.

The essential rules for analyzing software architecture to determine if it exhibits certain quality attributes are described in [4]. These rules provide a context for existent evaluation techniques such as scenarios, questionnaires, checklists and measurements. One of the first main rules is the identification of the contract between the system and the environment. Conforming to this rule, scenarios represent a form to express the expectations and the obligations of the system and this technique defines what needs to be confirmed by the analysis.

3.1.2 Scenarios

Most of the considered architecture analysis methods use scenarios. The existing practices with scenarios are systematized in [38]. The usage of scenarios is motivated by the consensus and it brings to understanding of what a particular software quality really means. Scenarios are a good way of synthesizing individual interpretations of a software quality into a common view. This view is more concrete than the general definition of software quality [29], and it also incorporates the specifics of a system to be developed, i.e. it is more context-sensitive.

Scenarios are a postulated set of uses or modifications of the system. In analyzing a system, it is important that all roles relevant to that system (operator, system designer, modifier, system administrator and others depending on the domain) will be considered since design decisions may be made to accommodate any of these roles. Scenarios are typically one sentence long and could be more appropriately called *vignettes*. The modifications reflected in scenarios could be:

- a change to how one or more components perform an assigned activity,
- the addition of a component to perform some activity,
- the addition of a connection between existing components, or
- a combination of these factors.

The development of scenarios is based on the system requirements that are reflected in the architecture. Scenarios have to be sufficiently concrete to ensure the expressiveness of the analysis. In this regard, the analysis performed in [45] demonstrates that it does not seem to be possible to assess the reusability of architecture in general. Typical reuse situations for applications in the respective domain are depicted in a set of scenarios. The concentration on a specific set of applications and specific reuse scenarios allows eliciting information about the flexibility of software architecture and its constraints.

3.2 A survey on software architecture analysis methods

Being a new research domain, most of the structural methods for assessing the quality of software architectures have been presented in conference and journal papers. Although refinement and experiments for validating some of the methods are ongoing, they deserve our attention because they contribute to the development of this still immature research area.

The set of discussed methods includes:

- the scenario-based architecture method (SAAM) [34] and its three particular cases of extensions, one founded on complex scenarios (SAAMCS) [46], and two extensions for reusability (ESAAMI) [54] and SAAMER [52],
- the architecture tradeoff analysis method (ATAM) [37],
- scenario-based architecture re-engineering (SBAR) [9],
- architecture level prediction of software maintenance (ALPSM) [11],
- a software architecture evaluation model (SAEM) [24].

We use these acronyms during our study for an easier reference and identification.

3.2.1 SAAM

SAAM provides a systematic approach for performing architectural reviews. Its objective is to verify basic architectural assumptions and principles against the documents describing the desired properties of a software system. In the early phase of the system design, the correction of the architectural mistakes detected by the analysis is still possible without causing excessively high costs. Additionally, the analysis offers a contribution to assess the risks inherent to the architecture. Related to organizational aspects, the analysis allows the harmonizing of various interests of the involved stakeholder groups, thus setting up a common understanding of the architecture as a base for later decisions.

SAAM appeared in 1993, corresponding with the trend for a better understanding of general architectural concepts as a foundation for proof that a system meets more than just functional requirements [35]. The main activities of the method are introduced in [38] where the method is applied to evaluate different user interface architectures with respect to modifiability.

In [34] we found that SAAM has been seen as a canonical method for a scenario-based architecture analysis of computer-based systems. Scenarios are considered the foundation for illuminating the properties of a software architecture, and from a body of experience a stable set of six steps and dependencies between those steps' activities have emerged – this was called SAAM (Figure 10).

The first step consists in the *scenarios development*. In this step, stakeholders identify possible events that may happen in the life of the system. Scenarios should illustrate the kinds of activities that the system must support and the kinds of anticipated changes that will be made to the system.

The activity of the second step is the *description(s) of the candidate architecture(s).* This step is recommended to be carried out in parallel with the first one. SAAM is applied very early in the architecture design. Functionality, structure and allocation are the three perspectives defined for understanding and describing architectures. Functionality is what the system does. A small and simple lexicon is used for describing structures for a common level of understanding and comparing different architectures. The allocation of function

to components identifies how the domain functionality is decomposed in the software structure. The architectural components could be described either as modules in the sense of Parnas [55], or as cooperating sequential processes.

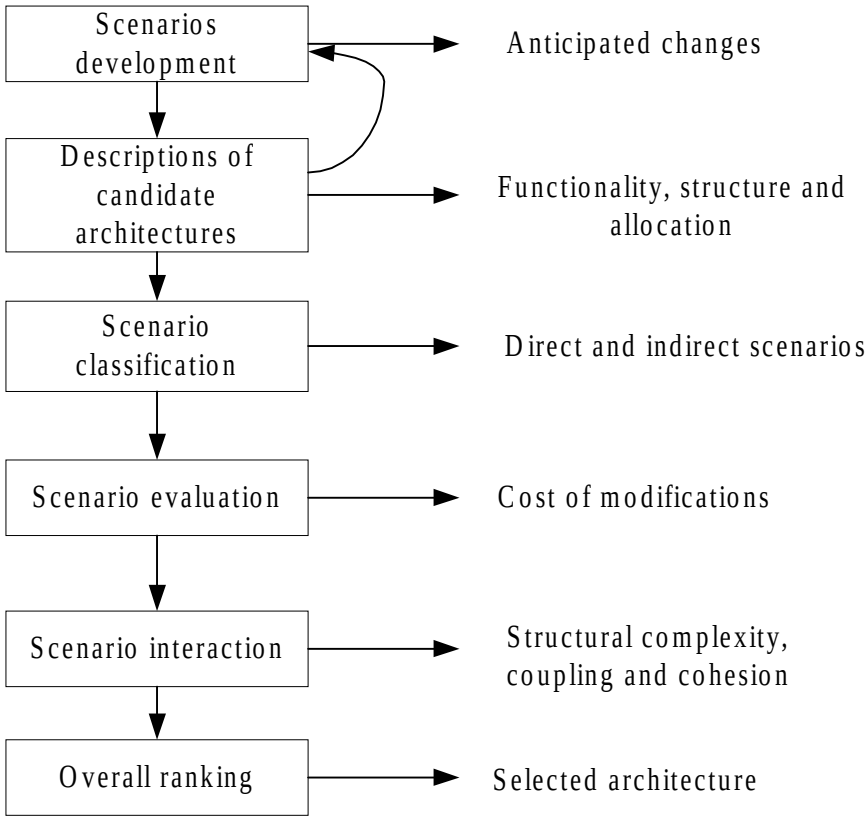

Figure 10. Six steps of SAAM.

The final version of architecture description together with the scenarios serves as the input for the subsequent steps of the method.

Scenario classification is the third step of the method. In this step it is important to decide whether a scenario requires modifications to the architecture. Scenarios that require no modifications are called *direct* and scenarios that do require modifications are called *indirect*. The purpose of the classification is to reduce the number of scenarios that is used as input for the next step in the method.

The next step performs *scenario evaluations*. The cost of the modifications associated with each indirect scenario is estimated by listing the components and the connectors that are affected. The result of this stage is a table that lists all scenarios (direct or indirect).

The table represents the input for the next step where *scenario interaction* is revealed. The purpose is to determine which scenarios interact, i.e. which ones affect the same component. High interaction of unrelated scenarios could indicate a poor separation of functionality. The amount of scenario interaction is related to metrics such as structural complexity, coupling and cohesion. Therefore, scenario interaction is likely to be strongly correlated with the number of defects in the final product.

SAAM can be used for both analyzing the quality attributes of a single software architecture and comparing the quality attributes of a number of architectures. The capability of the analysis method to evaluate the suitability of architecture with respect to the desired properties of a specific system can also be used for comparing different architectures. If the analysis is performed with the intention to choose among several architectural alternatives the analysis results of the considered architectures can be compared in the final step. To this end, scenarios and the scenario interactions should be weighted in terms of their relative importance. This weighting can then be utilized to determine an *overall ranking* of the candidate architectures.

SAAM cannot give precise measures or metrics of fitness. The aim of the architecture analysis is to guide the inspection of the architecture, focusing attention on potential trouble spots. The *result* of this method is a collection of small metrics (per-scenario analyses). This set of mini-metrics permits a comparison per a scenario-basis of a number of competing architectures.

SAAM is a mature method validated in numerous case studies. SAAM has been applied to evaluate a number of existing systems. The enumeration of the case studies includes global information system, air traffic control, WRCS (revision control system), user interface development environments, internet information systems, keyword in context (KWIC) systems, embedded audio systems, and visual debuggers.

3.2.2 SAAMCS

This method is introduced in [46] with the consideration of the complexity of scenarios as being an important factor for risk assessment. The contributions for extending SAAM are, on one hand, directed to the way of looking for the scenarios and on the other, to whereby their impact is evaluated. This method consists of three steps. The activities of these steps are the description of software architecture, identification of relevant scenarios, and evaluation of the effect of scenarios. As in SAAM, the first two steps should be performed in parallel. The goal of the second step is to find scenarios that may be complex to realize, and the third step applies a measurement instrument to evaluate the complexity of the scenarios.

The authors of the method introduce the idea that the systems within a domain are not isolated, but are integrated within an environment. As a result of this new supposition, the description of a software architecture of a system is divided into macro-architecture and micro-architecture. The measurement instrument, which is applied in the third step, includes factors that influence the complexity of the scenarios. In order to express the complexity of a scenario three different factors have been identified. These are:

- Four levels of impact of the scenario: no impact, affects one component, affects several components, affects software architecture.

- The number of owners involved in the information system.

- Four levels regarding the presence of version conflicts: no problem with different versions; the presence is undesirable but not prohibitive; creates complications related to configuration management; creates conflicts.

The method appreciates stakeholders' involvement and distinguishes the importance of the role of a scenario initiator. This role is the organizational unit that has most interest in the implementation of that scenario. Based on the initiator of the scenario, software architecture description and version conflicts, a list of classes of scenarios that are complicated to implement is provided.

3.2.3 ESAAMI

The conventional SAAM analysis in an architectural-centric development process considers only the problem description, requirements statement and architecture description. ESAAMI is a combination of analytical and reuse concepts and is achieved by integrating the SAAM in the domain-specific and reuse-based development process (Figure 12) [54]. Three factors influence the reusability of an architecture and are identified by the author of this method. These factors are: a common basis for a variety of systems in a domain, a sufficient flexibility to cope with variation among systems, and the documentation of properties to make them available for the selection of an architecture and its customization.

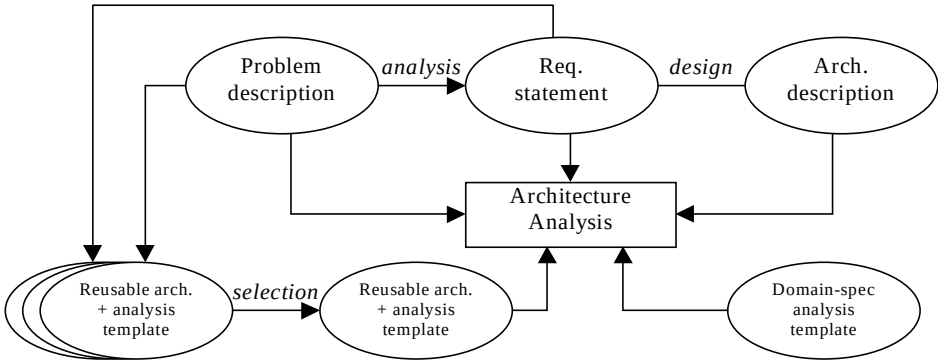

Figure 12. SAAM integrated in the domain-specific and reuse-based development [54].

A consistent basis for SAAM could be provided by reusable products which are collected in *analysis templates*. The reuse of analysis templates reduces the cost of the analysis and speeds up the process. The additional reusable products that can be deployed in the various steps of the analysis method are:

- *Protoscenarios*, which are generic descriptions of reuse situations or interactions with the system. These are intended to be used in the scenario elicitation step of subsequent architecture analysis after a selection and refinement process.

- *Evaluation protocols, proto-evaluations and architectural hints*, which are utilized during the step of scenario evaluation. These additional reusable assets are present in the protocols of the earlier evaluations in different projects, examples of descriptions of how the scenario can be performed using a set of abstract architecture elements, and hints associated to each scenario indicating which architectural structures would make the scenario convenient to handle.

- *Weights* established in different old projects in the same domain, thus making the results of the analysis comparable.

ESAAMI extends SAAM with two new techniques. One is based on reusing the domain knowledge by providing analysis templates representing the essential features of the domain. The degree of reuse is improved by concentrating on the domain. In this context, the analysis template is formulated on an abstraction level defined by the commonality of a large fraction of the systems in domain, and without referring to system specific architectural elements. The other technique is the specific knowledge about a reusable architecture. Thus, a reusable architecture is packaged with a tailored analysis template focused on the distinctive characteristics of the architecture. All these packages represent an input for the selection process of a reusable architecture. The selected one is a starting point for the architecture design of the new system and method steps.

From the practical viewpoint, the first step of this method is to select a reusable architecture to be deployed in a new system. It has to be ensured that the reusable architecture provides an adequate basis for the system to meet its requirements. SAAM estimates the effort for implementing scenarios that illustrate the requirements in the target system, predicting the effort required to realize a given part of a system's overall functionality. The selected architecture is then adapted and refined to meet the new requirements. The same set of scenarios is re-evaluated by SAAM to guarantee that the implemented system does not violate the initial design principles of the architecture, and that the initial assumptions about architectural properties of the system still hold. The results of this analysis are themselves part of the new-built system.

The author of the method signals the danger that the objectivity of the analysis may suffer due to the packaging of an analysis template together with a reusable

architecture. This problem result is similar to the well-known effects of solution-oriented as opposed to problem-oriented. A combination of the evaluation of architecture-related, domain- and project-specific scenarios is a solution to avoid this type of problems. In this way, it is recommended to take *project-specific properties* into consideration, while at the same time exploiting knowledge about the specifics of the reused architecture and the *system domain*.

3.2.4 SAAMER

From the point of view of two particular quality attributes, evolution and reusability, SAAM is extended in SAAMER [52]. The authors of SAAMER introduce a framework and a set of architectural views. The framework for information gathering and analysis consists of four activities: gathering information about stakeholders [16], architecture, quality and scenarios; modeling usable artifacts; analyzing; and evaluating. The method considers the following architectural views as critical for this type of software architecture analysis: *static, map, dynamic, and resource*. The static view integrates and extends SAAM to address classification and generalization of a system's components and functions and the connections between components. This classification and generalization of components and connections facilitates the estimation of the cost or effort required for changes to be made. Additionally, to further improve SAAM, two kinds of sources of information, the required changes and domain experts' experiences, are considered. Compared to SAAM where the risk is estimated by just counting the number of changes, both the information sources give a better suggestion about how the system could support each of the quality objectives or the risk levels for system evolution, or how to reuse across software domain systems.

An important point exposed by this method is that even if scenarios are considered the main drivers to evaluate various areas of architecture, the architectural views can also reveal deeper information. Scenarios describe an important functionality that the system must support, or recognize, where the system may need to be changed over time. Scenarios and the structural view are effective in identifying components that need to be modified, or are useful for preventive and adaptive maintenance activities. Analysis of scenario interaction is a critical step in SAAM. A high degree of scenario interaction may indicate

that a component is poorly isolated. However, the static view may show that this is just the nature of a particular architectural pattern. The dynamic view is appropriate to examine the behavior aspect to validate the control and communication to be handled in an expected manner. The mapping between components and functions could reveal the cohesion and coupling aspects of a system.

Furthermore, the method gives a practical answer to the question regarding when to stop generating scenarios. Two techniques are applied here. Firstly, scenario generation is closely tied to various types of objectives: stakeholder, architecture, and quality. Based on the objectives and domain experts' knowledge, the scenarios are identified and clustered to make sure that each objective is well covered. The second technique applied to validate the balance of scenarios with respect to objective is Quality Function Deployment (QFD) [16, 23]. From stakeholder objectives and architectural objectives to quality attributes, a cascade of matrices is generated to show the relational strengths. Finally, quality attributes are translated to scenarios to reveal the coverage of each quality attribute. An imbalance factor is then calculated for each quality attribute by dividing coverage by the priority of the quality. If the imbalance factor is less than 1, more scenarios should be developed to address the quality attribute in accord with the stakeholder, architecture and quality importance.

3.2.5 ATAM

The creators affirmed that ATAM has grown out of the work at the Software Engineering Institute on architectural analysis of individual quality attributes: SAAM for analyses of modifiability, performance, availability and security. The method was described in August 1998 [37] as being a spiral model of design and in May 1999 [33] as being a spiral model of analysis and design, which explains the recent evolution and progress of this method. The principal difference between other software analysis techniques is that ATAM considers the connections between multiple attributes. The objective of this method is to provide a framework to understand a software architecture's capability with respect to multiple competing quality attributes: modifiability, security, performance, availability, etc. [6]. The practice and reality have shown that

software systems need a trade-off among multiple software quality attributes, when their architecture is modeled and before they are implemented.

The method is divided into four main areas of activity [37]. These are the gathering of scenario and requirements, architectural views and scenario realization, attribute model building and analysis, and tradeoffs (Figure 14).

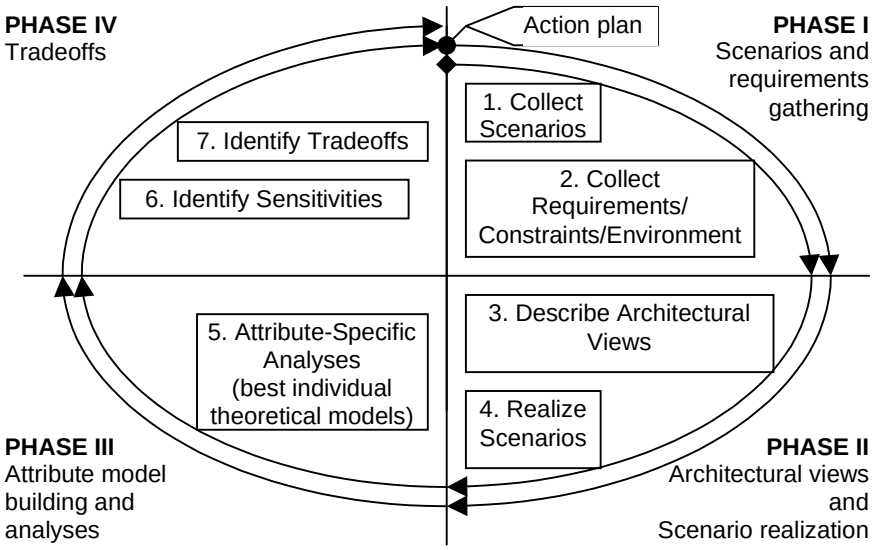

Figure 14. ATAM description.

Quality goals form the basis for architectural evaluation, but quality goals by themselves are not a sufficient basis to judge architecture for evaluation. Specific goals give a context and a meaning to quality attributes. So the first step in the architecture evaluation is to elicit the specific quality goals against which the architecture will be judged. The used mechanism is scenarios (Figure 16). Three types of scenarios are identified to probe the system from different architectural views, optimizing the chances of surfacing decisions at risk. These are:

- *Use cases*, which involve typical uses of an existing system and are exploited for the information elicitation.

- *Growth scenarios*, which cover anticipated changes in a system.

- *Exploratory scenarios*, which cover extreme changes that are expected to ”stress” a system.

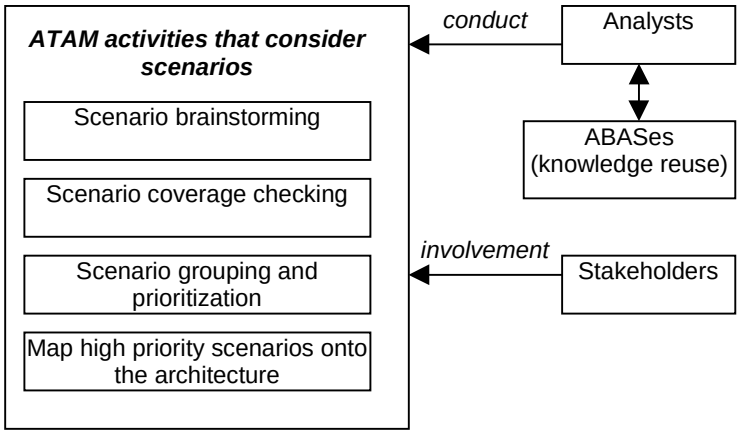

Figure 16. ATAM activities that consider scenarios.

We can identify a triple role of scenario usage in this method. This questioning technique helps to put in concrete terms as well as otherwise vague and unquantified requirements and constraints. Scenarios also facilitate communication between stakeholders because it forces them to agree on their perception of the requirement or constraint. Scenarios are also used to explore the space defined by the attribute model. Scenarios help to put the parameters of the models that are not part of the architecture into concrete terms. An existent taxonomy of each attribute is another base for ATAM. The taxonomies help to ensure attribute coverage and offer a rationale for asking elicitation questions. *Elicitation* questions facilitate the complete elicitation of attribute-specific information. In an analysis, the method also uses another type of questions. *Screening* questions guide or focus the elicitation on more ”influential” places of the architecture. These serve to limit the portion of the architecture under scrutiny.

ATAM considers *qualitative analysis heuristics*. Qualitative analysis heuristics are derived from ABAS and they are meant to be coarse-grained versions of the kind of analysis that is performed when a precise analytic model of a quality attribute is built. Asking these questions during an evaluation is more practical

than building quantitative models at that moment. These questions capture the essence of the typical problems or issues that are discovered by a more rigorous, more formal analysis.

Architecture view. The candidate architecture is generated based on requirements, scenarios and architectural design principles. The space of architecture is constrained by legacy systems, interoperability, and success failures of the previous projects. Architecture is a collection of functionality assigned to a set of structural elements, with constraints on coordination model - the control flow and data flow among them.

The architecture is described on the basis of five canonical or foundational structures, which are derived from Kruchten's "4+1 views" [42] (his logical view is divided into function and code structures). These five structures plus the appropriate mappings between them can be used to completely describe an architecture. During the analysis process ATAM requires several different views of the system: *a dynamic view*, showing how systems communicate, a *system view*, showing how software was allocated to hardware, and a *source view*, showing how components and systems were composed of objects. The architecture description should be annotated with *a set of message sequences charts* showing run-time interactions and scenarios. It is also important to understand the mapping between these views so that it can be determined how a change in one view will affect the representations and analytic models in another view.

If ATAM is applied during the design in an incremental improvement of the architecture, it does not require that all attributes be analyzed in parallel. The method allows the designer to focus on those attributes that are considered to be primary, and then introduce others later on. This can lead to cost benefits in applying the method, since what may be costly analyses for some secondary attributes need not be applied to architecture that was unsuitable for the primary attributes.

The method integrates the best individual theoretical model of each considered attribute in an efficient and practical way [28, 53]. Performance is one of these which has been analyzed for a long time [62]. Rate monotonic analysis (RMA) technique is part of performance analysis of real-time systems [39]. RMA is a

collection of quantitative methods that enable real-time system developers to understand, analyze, and predict the timing behavior of real-time systems. Individual models of the components, their interconnections, the available resources and the scheduling policies of the resources represent the basis for the RMA technique.

In ATAM, attribute experts independently create and analyze their models, then they exchange information (clarifying and creating new requirements). On the basis of this information, they can refine their models. The interaction of attribute specific analyses, and the identification of tradeoff have a greater effect on the system understanding and stakeholder communication compared to what any of those analyses could do on their own.

The process of analyzing architectural attributes forces to concretize the requirements and helps to uncover implicit requirements. The attribute analyses are interdependent: they depend, at least partially, on a common set of elements. Each analyzed attribute has implications on other attributes. The attribute interactions are discovered in two ways: using a sensitivity analysis to find tradeoff points and by examining the assumptions.

Tradeoff points and sensitivity points represent key decisions. Recognized from a knowledge base, unbounded *sensitivity points* are informally referred properties that have not yet been bound to the architecture. This information flags key decisions that have been made and those that have not yet been made. In a correct definition a *sensitivity point* is a property of one or more components (and/or component relationship) that is critical for achieving a particular quality. In practice, thus, changes to the architecture parameters affect significantly these modeled values. This can be obtained by using the stimuli and architectural parameters branches of attributes taxonomies. Tradeoff points are architectural elements to which multiple elements are sensitive. A *tradeoff point* could be defined as a property that affects more than one attribute and is a sensitivity point for at least one attribute.

During the architecture design the method provides an iterative improvement. In addition to the requirements typically derived from scenarios that are generated through interviews with the stakeholders, there are assumptions regarding behavior patterns and execution environments. Because attributes "trade off"

against each other, each assumption is subject to inspection, validation and questioning as a result of ATAM. When all these actions have been completed, the results of the analysis are compared to the requirements. If the system predicted behavior comes adequately close to its requirements, the designers can proceed to a more detailed level of the design or the implementation. In the event of the analysis revealing a problem, an action plan for changing the architecture, the models or the requirements, is developed. This leads to another iteration of the method in the development of software architecture. If only analysis is considered, the last step is to compare the results of the analysis to the requirements.

3.2.6 SBAR

In [9], a scenario-based method of the architecture re-engineering that focuses on multiple software qualities (reusability and maintainability) is presented. Various quality attribute research communities have proposed their own design methods for developing real-time [50], high performance [61, 62] and reusable systems [32]. All these methods focus on a single quality attribute and treat all others as having secondary importance, if any at all. SBAR considers these approaches unsatisfactory because a balance of various quality attributes is needed in the design of any realistic system. The contribution of this method is not only in the architecture design but also in the scenario-based evaluation of the software qualities of a detailed architecture of a system (Figure 18). A particularity of this method is that for assessing the architecture of the existing system, the system itself can be used. The goal of the evaluation method is to estimate the potential of the designed architecture to reach the software quality requirements.

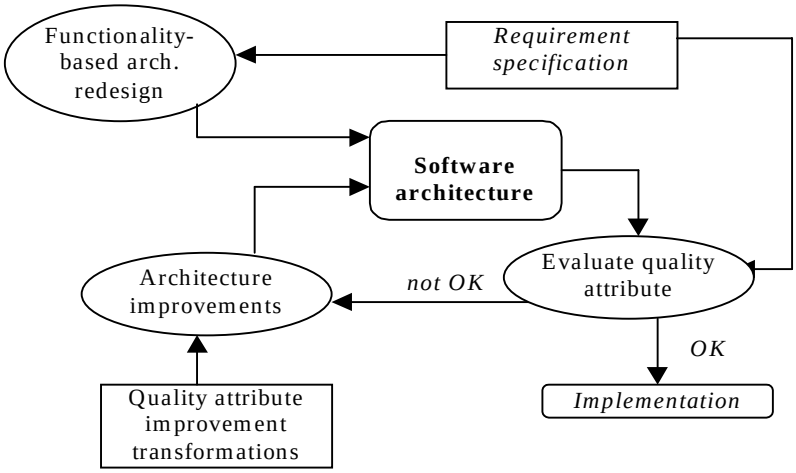

Figure 18. Reengineerining and architecture analysis by SBAR [9].

In SBAR, four different techniques for assessing quality attributes are identified: scenarios, simulation, mathematical modeling, and experience-based reasoning.

- *Scenarios:* This technique is recommended for the quality attributes of the development, such as maintainability and reusability, which are exemplified in the paper [9].

- *Simulation:* Simulation completes the scenario-based approach, being useful for evaluating operational software qualities such as performance or fault-tolerance.

- *Mathematical modeling*: Mathematical models allow a static evaluation of architectural design models. This technique is an alternative to simulation since both approaches are primarily suitable for assessing operational software qualities. To evaluate operational software qualities, the existent mathematical models or metrics developed by various research communities for high performance-computing [61], reliability [59], and real-time systems [50], could be used.

- *Experience-based reasoning:* This approach is founded on experience and logical reasoning based on that experience. Experienced software engineers often have valuable insights that may prove extremely helpful in avoiding bad design decisions and finding issues that need further evaluations. Although these experiences generally are based on anecdotal evidence, a logical line of reasoning can justify most of them. This approach is different from the other approaches. Firstly, the evaluation process is less explicit and more based on subjective factors such as intuition and experience. Secondly, this technique makes use of the tacit knowledge of the involved persons. For each quality attribute, the evaluator, in this case the designer, can select the most suitable approach.

Scenario-based evaluation of a software quality consists of defining a representative set of scenarios, analyzing the architecture and summarizing the results. The selected scenarios concretize the actual meaning of the attribute. Scenarios that capture typical changes in requirements may specify the maintainability. The performance of the architecture in the context defined by each individual scenario for a quality attribute is assessed by the analysis. Posing typical questions for the quality attributes can be helpful. The results from each analysis of the architecture and scenario are then summarized into overall results, e.g. the number of accepted scenarios versus the number of the not accepted ones.

The assessment process consists of defining a set of scenarios for each software quality, manually executing the scenarios for the architecture and subsequently interpreting the results. The assessment can be performed in a complete or statistical manner. In the first approach, a set of scenarios is defined; combined together, they cover the concrete instances of the software quality. If all scenarios are executed without problems, the quality attribute of the architecture is optimal. The second approach is to define a set of scenarios that makes a representative sample without covering all possible cases. The ratio between scenarios that the architecture can handle and scenarios not handled well by the architecture provides an indication of how well the architecture fulfills the software quality requirements. Both approaches obviously have disadvantages. A disadvantage of the first approach is that it is generally impossible to define a complete set of scenarios. The definition of a representative set of scenarios is

the weak point in the second approach, since it is unclear how one decides that a scenario set is representative.

3.2.7 ALPSM

An example of the prediction method of a software quality attribute at the architecture level is described in [11]. The method is related to the software maintenance community through the analysis of software change impact [12], but the main contribution consists of the architecture level where this prediction is performed.

ALPSM defines a maintenance profile, like a set of change scenarios representing perfective and adaptive maintenance tasks. A scenario describes an action, or sequence of actions that might occur as related to the system. Hence, a change scenario describes a certain maintenance task. Using the maintenance profile, the architecture is evaluated using scenario scripting, and the expected maintenance effort for each change scenario is evaluated. Based on this data, the required maintenance effort for a software system can be estimated. The method has a number of inputs: the requirements specification, the design of the architecture, expertise from software engineers and possibly historical maintenance data. This method consists of the following six steps:

1. *Identify categories of maintenance tasks:* formulate classes of expected changes based on the application or program description.

2. *Synthesize scenarios:* for each of the maintenance tasks, a representative set of scenarios is defined.

3. *Assign each scenario a weight:* the scenarios are assigned a weight based on their probability of occurring during a particular time interval.

4. *Estimate the size of all elements*: to be able to assess the size of changes, the size of all components of the system is determined. One of the three techniques can be used for estimating the size of components: using estimation technique of choice, an adaptation of an Object-Oriented metric or, when historical data from similar applications or earlier releases is

available, existing size data can be used and extrapolated to new components.

5. *Script the scenarios:* for each scenario, determine the components that are affected and to what extent they will be changed, this resulting in the size of the impact of the realization of the scenario.

6. *Calculate the predicted maintenance effort:* the total maintenance effort is predicted by summing the size of the impact of the scenarios multiplied by their probability.

This method analyzes maintainability by looking at the impact of scenarios. It uses the size of changes as a predictor for the effort needed to adapt the system to a scenario.

3.2.8 SAEM

The evaluation process of the quality requirements of the software architecture is rigorously formalized, especially in relation to metrics in the model described in [24]. The basis for software architecture quality evaluation and prediction of the final system quality is established in this model. For this purpose, a quality model based on standard software quality assessment process [31] is chosen, and a conceptual framework that relates quality requirements, metrics and internal attributes of the software architecture and the final system is proposed.

The elements required for quality evaluation of a software system, based on standard specification, are quality model, method for evaluation, metrics and supporting tools. SAEM gives a quality evaluation model based on data collection, measurement and analysis of the results.

The quality specification and the evaluation process are divided into external and internal ones. The external quality expresses the user view, and a specific quality evaluation model is applied to it. Internal quality expresses the developer view, and the evaluation process is adapted to this feature. In this sense, the specified quality requirements are mapped to internal attributes that will be present in the software architecture based on the experts' knowledge and company accumulated data. The internal quality attributes are composed by *special*

elements (such as functional elements or data elements) denoting quality characteristics, and *intrinsic properties* resulting from the development process (such as size, modularity, complexity, coupling, and cohesion). The author mentions the necessity to establish a relative importance between internal attributes and their values and recommends QFD [58] as a suitable technique for this purpose.

For quality evaluation, the goal of metrics is to find whether certain attributes meet the values specified in the quality specification for each software characteristics. The evaluation model assumes the existence of a previous internal quality specification, which defines the expected internal attributes with their values and their evaluation procedure.

Architecture development process constraints the internal attributes, so the result of the measurement process can improve architecture as a feedback. Internal metrics of the architecture indicates that the software satisfies external quality requirements of the product. An association between the internal characteristics and external quality characteristics is required (for example modularity and diagnostic function can be associated to maintainability). The architecture description language (ADL) model should have attached questioning or inspection techniques (such as software architecture models walkthrough) to detect the presence or absence of special elements. The intrinsic properties can only be detected by measuring techniques applied to the software architecture representation formalized through an ADL.

3.3 Discussion

The purpose of this discussion is to offer guidelines related to the use of the most suitable method for an architecture assessment process. The beginning part of discussion focuses on an appropriateness study. This study identifies the common goal and how this goal is interpreted by each of the analysis methods. The next part of this section contains comparison discussions, which identify differences and similarities between these methods. Several classifications of the methods are also established. Included evaluation techniques, the number of quality attributes, the stakeholders' involvement and when the method is applied in the architecture-based development process, are the main criteria of

classification. To maintain a pertinent discussion in exemplification, we consider only the most representative methods.

Common problems such as when to stop generating scenarios and how the scenarios' impact on a considered architecture is evaluated are identified in the analysis methods based on scenarios. Different proposed solutions are discussed in the next part.

The headlines of the last part of this section consider the special case of the evolution of ATAM from SAAM and how the existing knowledge is reused by the analysis methods.

3.3.1 Appropriateness study

Objective views are considered a basis for establishing which analysis method is most suitable for an architecture assessment process. Although each method has its particularity in the definition of its objectives (i.e. confidence building or risk assessment [47]), in all of them, we can identify a collective goal which is the prediction of the quality of a system before it has been built. In each method this goal is reflected under different angles and perspectives. The directions of these reflections are oriented to:

- guide the inspection of the architecture, focusing on potential trouble spots (SAAM),
- evaluate the potential of the designed architecture to reach the software quality requirements (SBAR),
- predict a quality attribute (maintainability) of a software system based on its architecture (ALPSM),
- establish the basis for the software architecture quality evaluation and prediction of the final system quality (SAEM), and
- locate and analyze tradeoffs in a software architecture, for these are the areas of highest risk in an architecture (ATAM).

3.3.2 Several classifications criteria of the analysis methods

3.3.2.1 Based on the included evaluation techniques

The common and particular characteristics of the goals drive to similarities and differences between all these presented methods (Table 4). At this moment we can establish a possible classification of the methods considering the techniques they use. From this point of view some of the methods are:

- only scenario-based, like SAAM,
- scenario-based and attribute model-based analysis technique, like ATAM,
- proposing different evaluation techniques depending on the attribute type, like SBAR,
- related to metrics, like SAEM.

The quality model of attributes for quantitative evaluation is treated during the evaluation process in two of the presented methods. However, from this angle we identify different approaches:

- SAEM, which is trying to define metrics based on the goal-question-metric (GQM) technique,
- ATAM, which considers that analysis techniques indigenous to the various quality attribute communities can provide a foundation for performing software architecture evaluation. It is not necessary to invent attribute-specific techniques and metrics, but to integrate existing techniques and metrics into systematic procedures or methods for architecture evaluation. ATAM provides flexibility in the integration of the best individual, theoretical model of each considered attribute.

Table 4. A summary of analysis methods.

Issue\Method	**SAAM**	**SAAMCS**	**ESAAMI**	**SAAMER**
The included evaluation technique	Scenarios	Scenarios	Scenarios	Scenarios
Attribute model	Qualitative using scenarios	Like SAAM	Like SAAM	Like SAAM
Number of quality attributes	Single	Single	Single	Single
Stakeholders' involvement	All	All	All	All
When to stop generating scenarios?	When the addition of a new scenario no longer perturbs the design.	Defines a framework diagram to discover all the complicated scenarios	Like SAAM	Uses a two-steps procedure
When is the method applied?	On the final version	On the final version	On the final version	On the final version
Scenarios impact evaluation	Relationship	Relationships, owners, versions	Like SAAM	Estimates the cost required for the change to be made
Reusability of the existing knowledge	Not used	Not used	Packages of analysis templates and reusable architectures	Not used

55

Table 3. A summary of analysis methods continues.

Issue\Method	**ATAM**	**SBAR**	**ALPSM**	**SAEM**
The included evaluation technique	Scenarios and attribute specific model	Depends on the attribute type: scenarios, mathematical modeling, simulators, objective reasoning	Scenarios	Metrics
Attribute model	Integrates existent qualitative and quantitative techniques	Qualitative	Built a maintenance profile based on scenarios	Different metrics based on GQM technique
Number of quality attributes	Multiple	Multiple	Single	A quality model
Stakeholders involvement	All	Designer	Designer	Not used
When to stop generating scenarios?	Uses a standard quality attribute-specific questions	Defines a complete set or a representative set of scenarios	Like SBAR	Not applied

Table 3. A summary of analysis methods continues.

Issue\Method	**ATAM**	**SBAR**	**ALPSM**	**SAEM**
When is the method applied?	On the final version or combined with the architecture design into an iterative improvement process	Combined with the architecture design into an iterative improvement process	During design to predict adaptive and perfective software maintenance (based on previous implementations)	On the final version
Scenarios impact evaluation	Like SAAM	Optimized or fulfilled	By estimating the size of the components and the extent to which they are effected	Not applied
Reusability of the existing knowledge	Uses a set of pre-packages analyses and questions including known solutions	Object reasoning is proposed but not exemplified	Not used	Not used

3.3.2.2 Based on the considered number of quality attributes

Some analysis methods are centered on the evaluation of a single quality attribute. However, for a better understanding of the strengths and weaknesses of a complex real system and its parts, performing a multi-attribute analysis is required. An important characteristic revealed by studying the analysis methods is the *number of quality attributes* a method focuses on. From this point of view we can distinguish:

- multiple quality attributes (ATAM, SBAR). For example, ATAM considers the architectural elements where multiple attributes interact,
- single quality attribute (SAAM), and
- a specific quality model (SAEM).

Reusability and modifiability were defined as the main drivers of a PLA. In the SBAR method, scenarios that are also used for modifiability in the SAAM method, are recommed for the analysis of reusability and maintainability.

3.3.2.3 Based on stakeholders' involvement

Although it is recognized that the *involvement* in the evaluation of all the *stakeholders* facilitates communication between them, not all the methods consider their presence as mandatory. ALPSM differs from SAAM in that it does not involve all stakeholders, and thus requires less resources and time, but instead provides an instrument to the software architects that allows them to repeatedly evaluate the architecture during design. Due to the need of a stakeholder's commitment, this method could be used in combination with SAAM. In SAAM and ATAM, architecture is evaluated in cooperation with the stakeholders prior the detailed design, but in SBAR, architecture is evaluated on a detailed design for re-engineering without any stakeholders' involvement, however posing typical quality questions at the same time.

3.3.2.4 When the method is applied

Considering the architecture-based development process, an important question is: when is the method applied?

- A common approach, which combines architecture analysis and design into an iterative improvement process; it could be identified in ATAM and SBAR. While SBAR includes guidelines about how to transform the architecture in order to meet certain quality requirements, ATAM concentrates on identifying sensitivities and tradeoff points. However, ATAM could also be applied for the evaluation to the final version of the architecture.

- SAAM, SAAMCS, ESAM, SAAMER are applied to the final version of the architecture, and therefore, they are appropriate in the evolution of a PLA.

- ALPSM is applied during the design to predict adaptive and perfective software maintenance.

SAEM is applied to the final version, but here it should be noticed that the evaluation model considers software architecture from two different viewpoints, developer's and user's. Therefore, the software architecture is either a final product, or an intermediate in the software system development process. The rigorous ambition of this model makes it hard to believe that it will be suitable for usage in an iterative and incremental software architecture design process.

3.3.3 Common problems and different solutions in scenario-based methods

Scenario-based assessment is particularly appropriate for qualities related to software development. Software qualities such as maintainability, reusability, modifiability, adaptability and portability can be expressed very naturally through change scenarios. As Poulin [57] concluded for reusability, no predominant approach for assessing this quality attribute exists. However, scenario-based evaluation depends on the objectivity and creativity of the

software engineers that define and execute them. In [1], the use of scenarios for evaluating architectures is recommended as one of the best industrial practices.

3.3.3.1 When to stop generating scenarios

A common problem of scenario-based methods is when to stop generating scenarios:

- SAAM considers that the set of scenarios is complete when the addition of a new scenario no longer perturbs the design.

- In SBAR two approaches are discussed. One is to define a complete set, which is generally impossible. The other is to define a representative set, which has the weak point of how to define which is the representative set. The last one is based only on the creativity and subjectivity of the software engineer.

- SAAMCS considers that the relevant scenarios are those which are, possibly, complex to realize. A two-dimensional framework diagram (5 categories of complex scenarios, 4 sources of changes) that may help to discover complicated scenarios is defined.

- SAAMER defines a practical two-steps procedure. In the first step, a coverage guarantee is obtained. The scenarios are identified and clustered based on the objectives and domain experts' knowledge, and the coverage is checked against the objectives of stakeholders, architecture, and quality. The second step validates the balance of scenarios with respect to the objective based on Quality Function Deployment (QFD) technique. The decision to develop more scenarios is made based on comparison against 1 of a calculated imbalance factor for each quality attribute.

- ATAM uses a set of *standard quality attribute-specific questions* to ensure proper coverage of an attribute by the scenarios. The boundary conditions should be covered. A standard set of quality-specific questions gives one the possibility to elicit the information needed to analyze that quality in a predictable, repeatable fashion.

3.3.3.2 Impact Evaluation

There are differences in the evaluation of the scenarios' effects on the considered architecture. The identified differences in the impact evaluation are:

- SAAM: Evaluates the effect of a scenario by investigating which architectural elements are effected by that scenario. The cost of the modifications associated with each indirect scenario is estimated by listing the components and the connectors that are affected and counting the number of changes.

- ALPSM: The effort needed to implement the scenario is predicted by estimating the size of the components and the extent to which they are effected.

- SAAMCS: Defines and uses a measurement instrument to express the effect of scenarios. The instrument indicates the impact of a scenario, whether multiple owners are involved and whether it leads to versions conflicts.

- SAAMER: A classification and generalization of the architectural elements facilitates the estimation of cost or effort required for changes to be made. The required changes specified in scenarios and domain experts' experiences suggest how the system could support each of the objectives or the risk levels for the systems evolution or reuse across applications.

- SBAR: The evaluation can be performed in a complete or statistical manner. The optimality of a quality attribute could be obtained using the former approach and the fulfillment of a quality attribute could be obtained using the latter one.

3.3.4 Methods evolution

A special case of qualitative and quantitative progress could be observed in ATAM. Considering the uses of scenarios, ATAM is based on SAAM. Unlike SAAM, which focuses on the architectural modifiability evaluation, ATAM focuses on finding tradeoff points in the architecture from the perspective of quality requirements on the product. In addition, ATAM prescribes formal or

informal analytic models for assessing the quality attributes of the system, but relies on the existence of such techniques for the quality attributes relevant to each case. In the case of a specific analysis of modifiability, ATAM builds an informal model like SAAM with inspection and review methods. Scenario interactions are interpreted as sensitivity points.

3.3.5 The reusability of the existing knowledge

Similarities at a coarse-grained level could also be identified between ATAM and ESAAMI. Both methods are based on SAAM. Considering the reusability of the existing knowledge, ATAM uses ABASes and ESAMI proposes packages of analysis templates and reusable architectures. However, when we talk about systematization of information there is no possible comparison. ESAAMI allows making available domain-respective architecture specific experience in an intuitive form, while ATAM is anchored in a very well structured knowledge base of quality attributes communities and architectural styles. ABASes provide a set of pre-packages of analyses and questions including known solutions to commonly recurring problems and known difficulties in employing those solutions. ATAM is based on a set of materials that describe many of the evaluation artifacts, like ABASes, a set of quality attribute-specific questions that aid the evaluator in probing an architecture and a set of questions that aid the analyst in gathering the information needed to build an analytic model of the quality.

We could also identify progress in individual attribute analysis techniques. For example, the generalization of the RMA technique in order to support future product evolution with the minimum possible effort is a new refinement direction defined in [3] dealing with family products. This issue could be considered and a reusable knowledge base could be adapted to these improvements.

4. PLA analysis strategy

The advantages of adopting a product line approach are, as described in chapter 1, decreased development and maintenance cost and time to market, and increased software quality. This chapter suggests a practical solution for the product line architecture development founded on an analysis of the software quality attributes. The beginning part is an introduction to the PLA analysis strategy. The remainder of the chapter focuses on the PLA analysis when the evolution of product line is considered. We will propose a measurement instrument, which determines if a new product should be added to an existent product line, and which components should be added to the domain architecture framework and which parts should be product-specific.

4.1 Introduction to the PLA analysis strategy

Product line architecture and reusable software components are suitable approaches for software systems, which are often re-engineered from existent ones [21]. From this point of view embedded systems are good examples. Although there are some similarities between embedded systems regarding quality attributes, there are also differences. If a quality attribute is important to one product line domain it does not necessarily mean it is important to another one. For example, availability is very important to switching systems, but generally it is not of the maximum level of priority. Power consumption and weight are features that may cause restrictions in the case of wireless products. In addition, the weight changes when different types of software that may exist in a product, are considered. Performance requirements, such as latency and throughput, have the highest value for digital signal processing software. An important quality is the cost, which is caused by the code size and data storage capacity. In order to be able to define a list of priorities of quality attributes, the embedded systems domain should be very well delimited. A practical approach is to define an appropriate domain based on the scope of the products, and analyze what is variable and common among the characteristics of all products in the line. The functionality and quality characteristics of the domain are important capabilities to be considered.

The domain analysis is important for the evolution of a software product line. Establishing the priorities among domain quality attributes is not enough if we are thinking about the architecture, which is one of the core assets shared by the product group. From this point of view the structural quality attributes such as modifiability, reusability, adaptability, portability etc. should be considered and a second list with these and their priorities could consequently be organized.

There is a common opinion among the researchers that the development process for a product line architecture is different from the one for one product. Practices areas essential to the development of a single product line architecture include software architecture design, representation, analysis, implementation in conformance with the specification, and traceability with requirements. In product line approach, additional areas like generic and flexible software architectures, reusable components design and development, traceability with common and variable requirements in the group of products, are considered.

PLA is mainly a design when the product line is initiated based on the existing products and evolves as long as new products are added to the line. Figure 20 describes the context of PLA analysis strategy, which considers the process of initiation and evolution of a PLA. A PLA analysis strategy takes account of the assessment process of the two lists of quality attributes.

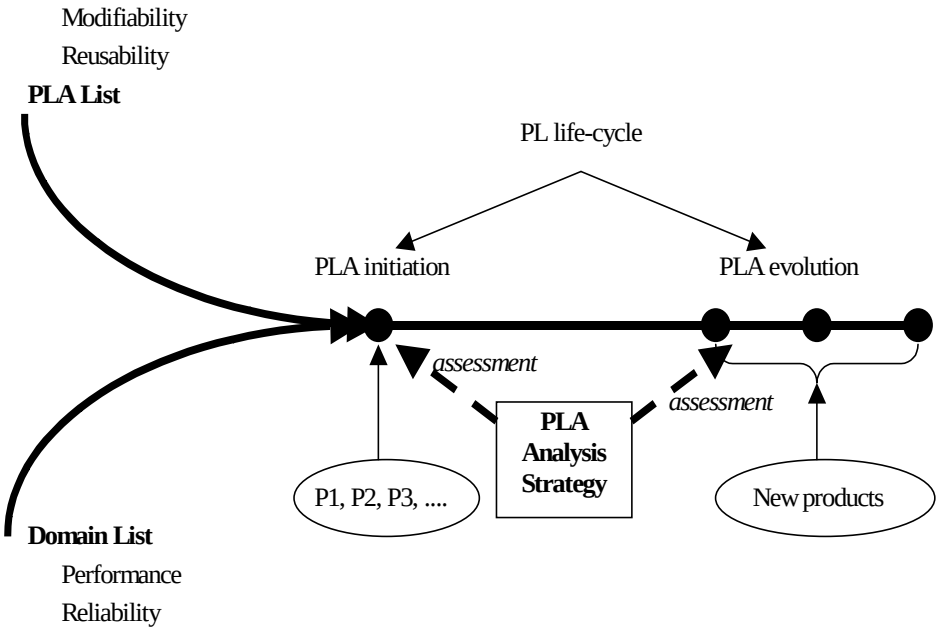

Figure 20. The context of PLA analysis strategy.

4.2 The measurement instrument

The measurement instrument is defined by a taxonomy for quality attributes, which is organized with respect to three main elements (Figure 22):

- The priority in a domain or PLA list. The presence of this element in the taxonomy is necessary due to the costs required by an analysis method at the architectural level.

- Architecture views, which are relevant for the specific quality attribute.

- Appropriate methods to be applied for quality attribute analysis.

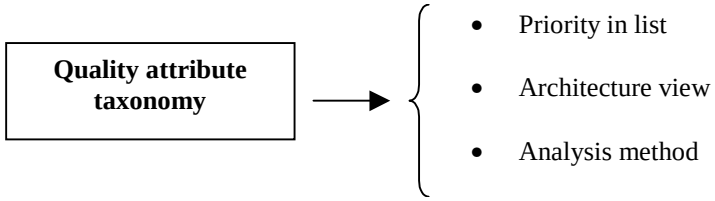

Figure 22. The main elements of a quality attribute taxonomy.

The priorities of the quality attributes in a domain are established based on the experts' knowledge and stakeholders' objectives (Figure 24). Quality function deployment (QFD) [23] is a suitable technique to show the relational strengths from stakeholders' objectives and architectural objectives to quality attributes.

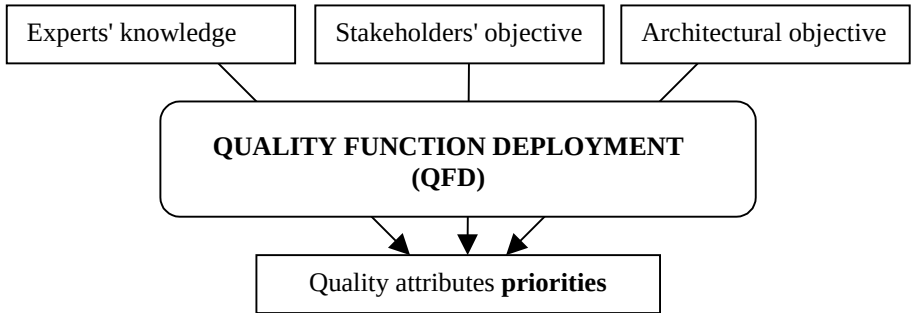

Figure 24. The role of QFD in establishing quality attributes priorities.

The list of priorities is important for the evaluation process, which considers an appropriate analysis method for each quality attribute. At this moment various architecture analysis methods such as scenario-based architecture analysis (SAAM), architecture tradeoff analysis (ATAM), or scenario-based architecture re-engineering (SBAR), are available. Our study about the existent state-of-art research in the domain reveals that the methods are distinguished taking into account the included evaluation techniques (qualitative or questioning, like scenarios; quantitative or measuring, like metrics, etc.), the number of considered quality attributes and their interaction for tradeoff decisions, the stakeholders' involvement, and how detailed the architecture design is at the moment when the method is applied to the architecture-based development process. Figure 26 exemplifies the most appropriate methods for analyzing several quality attributes.

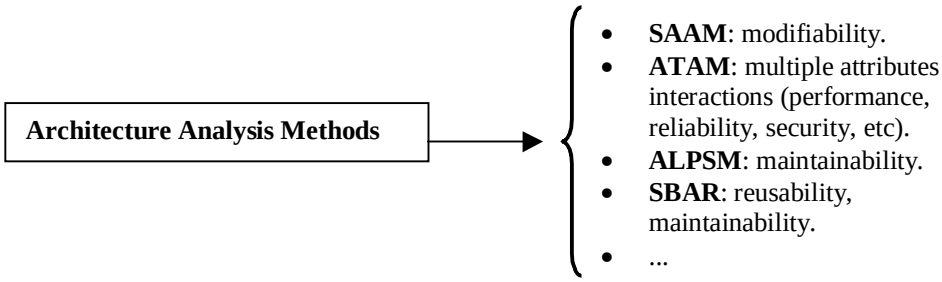

Figure 26. An example of quality attributes and the appropriate analysis methods.

An important issue in PLA assessment, which can be confirmed by the experience of the companies that are working in this area, is listening to the stakeholders. A product line environment requires new roles that take part in the analyzing process. These are product line architect, builder of generic (core) assets, builder of product from generic assets, product line maintainer and marketer/founder of the product line.

4.3 The evaluation procedure

The measurement instrument is applied on the architecture of a new product. The quality attribute with the first priority in a list is first analyzed with respect to the appropriate architecture view and appropriate method. If the results of the analysis are not acceptable, then the new components of a product should not be added to the PLA nor the new product to the product line. Otherwise the next quality attribute from the list is analyzed in isolation and then by considering the interaction with the first one. The process is repeated for all the attributes in the list. If the obtained results are acceptable, the new components of the new product should be added to the PLA and the product to the product line. The steps of the evaluation procedure could be described in the following pseudo-code:

Initialization

counter := the number of the quality attributes in the domain list;

1. *The quality attribute with the first priority is analyzed with respect to the appropriate architecture view and appropriate method.*
2. *If the results of the analysis are not acceptable the new components of a product should not be added to the PLA.*
3. *Else*

```
counter:=counter-1;
repeat          // repeat for all the attributes in the list
                the next quality attribute from the list is analyzed
                as in the fist step in isolation;
                considering the interaction with the previous ones;
                if the results are poor
                break ;
                else
                counter:=counter-1;
until (counter=0)  // end repeat
```

4. *If (counter=0)* // all attributes of the list and the obtained results are good *the new product is added to the PL.*

In the case of quality attributes with the same priority they should be analyzed in isolation and also considering the interactions between them. In order to decide for the PLA core development and maintenance, this procedure could also be improved and refined. From this point of view, the other list of priorities should also be considered. In this case a special attention should be paid to the collections of components in the architecture of a new product that are critical for achieving a particular quality attribute, or architectural elements to which multiple quality attributes are sensitive. A deeper level of architecture analysis could influence the decision for adding new components to the PLA.

5. PLA analysis method

The potential risk of PL development is minimized if the interactions of quality attributes are considered in the analysis method [41]. Not only modifiability, but also other structural or run-time quality attributes are important to the PL software development. However, reusability and modifiability are the main quality drivers for the PLA design. The analysis of these two quality attributes could be combined with other run-time quality requirements (performance, reliability, security, etc.) of the PL domain.

In the previous chapter we discussed the PLA analysis strategy and introduced, in general terms, a measurement instrument and a procedure to apply this instrument. The content of this chapter focuses on the specific reusability and modifiability properties of PLA. At the end, we will describe a method which could be applied for a PLA analysis.

5.1 Reusability and modifiability considerations

5.1.1 Reusability

In [20], the reusability in software architecture development is divided into two categories. It is considered that reusability has two major aspects – software development *with* reuse and software development *for* reuse. The *with* reuse aspect requires the construction of software architecture that allows 'plug-in' prefabricated structures and code components. The system is composed of existing components by adapting them to the needs and implementing 'glue' components to connect them. If the *for* reuse aspect is considered, the produced components are potentially reusable in future projects as part of the current software development. Parts of the software system under development are taken and reused in other systems without any modification.

Software reuse can be classified as fine-grained reuse, which reuses only a small amount of code segments at the programming level, and coarse-grained reuse, which reuses design concepts and/or a large amount of codes. Promoting the granularity of reuse is important, when increasing the scale of software reuse. A

coarse-grained reuse at the architecture level could be separated in technologies and domain knowledge.

Reusable entities in software architectural technologies can be divided into several categories, such as architectural style, architectural representation, architectural pattern, etc. [8, 20, 25].

The reusability of the existing knowledge includes packages of analysis templates associated with reusable architectures [40, 54]. The architecture-specific experience must be structured in a knowledge base to provide a set of pre-packages analyses and questions including known solutions to commonly recurring problems and known difficulties in employing those solutions. A PLA analysis knowledge base should be organized in three important sets:

- A set of materials that describe many of the evaluation artifacts,
- A set of quality attribute-specific questions that aid the evaluator in probing a product architecture,
- A set of questions that aid the analyst in gathering the information needed to build an analytic model of the quality attribute.

5.1.2 Modifiability

Modifiability is one of the important structural requirements for a PLA. According to [8], modifiability is the ability to make changes quickly and cost effectively. Modifications to a system can be categorized into:

- *Extensibility*, which is the ability to acquire new features,
- *Deleting unwanted capabilities*, to simplify the functionality of an existing application,
- *Portability*, for adapting to new operating environments, and
- *Restructuring*, which means rationalizing system services, modularizing, creating reusable components.

On the other hand, each of these categories could be analyzed standalone, if its identity has been identified in a set of well-defined sub-characteristics, possibly in a quality model form.

5.2 Method description

Scenario-based assessment is particularly appropriate for qualities related to software development, which are specific to product line architectures [26]. Software qualities such as maintainability, reusability, modifiability, adaptability and portability can be expressed very naturally through change scenarios. However, scenario-based evaluation depends on the objectivity and creativity of the analyst who defines and executes them. In [1], the use of scenarios for evaluating architectures is recommended as one of the best industrial practices. Our strategy is based on the SAAM, but improved through the introduction of guidelines for PLA analysis. Our method considers product line specific techniques such as commonality analysis, which systematically models the required similarities and differences among PL members. The analysis method consists of five important steps, which are:

- deriving change categories from the problem domain,
- scenario identification,
- PLA description,
- evaluation of the effect of the scenarios on the architecture elements, and
- scenario interaction.

Next chapters describe each step in detail.

5.2.1 Deriving change categories from the problem domain

The first step consists of defining several categories of changes derived from the problem domain of the PL.

Figure 28 presents five categories of the change scenarios derived from the problem domain of the spectrometer controller PL system used as an example in this research.

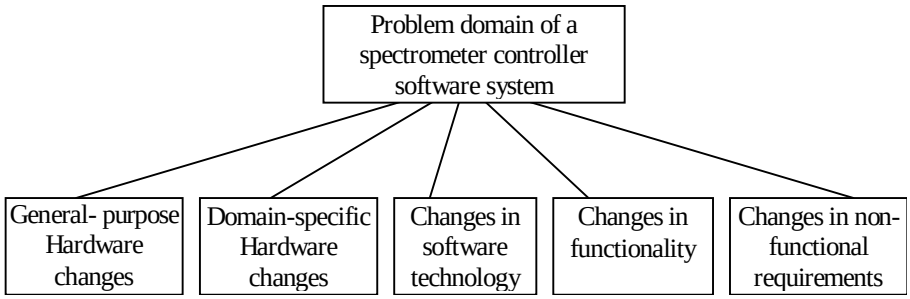

Figure 28. Deriving change categories from problem domain.

It may be possible that a change scenario related to one of these categories requires other changes in the other categories. It is recommended to follow this approach in the scenario development activity. The approach operates when the problem domain is organized so that it is easy to identify the main sources for the addition of subsequent features in the domain. An exemplification is detailed in the next section.

5.2.2 Scenario identification

The second step of the method consists of the identification of the scenarios. In this activity we distinguish possible changes that may happen in the life of the PL based on the derived categories. Scenarios should illustrate the kinds of anticipated changes that will be made to the PLA due to the PL scope.

A common problem of the scenario development activity is when to stop generating scenarios. Possible solutions could be suggested:

- The set of scenarios is complete, when the addition of a new scenario no longer perturbs the design.

- Try to identify a complete set of scenarios - but this is generally impossible.

- Delimit a representative set of scenarios, which has the weak point in how to define which is the representative set. This solution is based only on the creativity and subjectivity of the analyst, or it requires a domain knowledge base organized in the three important sets characterized previously.

- Consider various criteria for the relevant scenarios. For example, scenarios which may be complex to realize. A two-dimensional framework plan (categories of complex scenarios, sources of changes) may help identify complicated scenarios.

- Try to apply a procedure for identification. For example, a two-steps procedure, where in the first step a coverage guarantee is obtained. The scenarios are identified and clustered, based on the objectives and domain experts' knowledge, and the coverage is checked against the objectives of the stakeholder, architecture and quality. The second step validates the balance of scenarios with respect to the objective, based on a Quality Function Deployment (QFD) technique [58].

- Use a set of *standard quality attribute-specific questions* to ensure proper coverage of an attribute by the scenarios. The boundary conditions should be covered. A standard set of quality-specific questions allows the possibility of extracting the information needed to analyze that quality in a predictable, repeatable fashion.

If it is considered that the architecture is a good one, it is not necessary to generate scenarios to verify the functional requirements. Otherwise these should also be considered when verifying functionality. For analyzing the modifiability we must look for possible changes in the problem domain defined by the product requirements. Most of the anticipated changes characterize the product line variability that the PLA must harmonize.

5.2.3 Description of the PLA

Another required activity is the *description of the PLA.* This activity is considered the third step, but it could be performed in parallel with the previous one. The method is applied to analyze the conceptual view or a more detailed functional representation of the architecture, when not all the necessary structures have been designed and the architectural representation may not include some concrete domain requirements.

A specific of the PLA architecture is that it contains abstractions of the problem domain but also concrete components, which could be common or variants of different family products. PLA combine the generics of the domain with the concreteness of each product and one of its views must reflect the diversity and the range of this diversity.

The minimum of required views for a PLA representation is:

- A conceptual view considered as being a functional decomposition of the PLA into subsystems. The relationships between components are based on *pass-to control*, *pass-to data* or *uses*.

- A more detailed functional description, where the main objects are packages, components, ports, and protocols. Components, protocols and connectors distributed in abstract or concrete packages are defined for understanding and describing PLA based on features of the problem domain. The relationships are association, specialization, generalization, etc. Considering the dynamic aspect, statechart diagrams and message-sequence charts (MSC) are also part of this view.

- A diversity view, the particular PLA view as it has been introduced previously based on the research experiences on the case study.

For a common level of understanding, a small and simple lexicon could be used in describing structures.

5.2.4 Evaluate the effect of the scenarios on the architecture elements

The evaluation of the scenarios' effects on the analyzed PLA view may consider several issues. We can identify some of these in the following enumeration:

- A classification and generalization of the architectural elements facilitates the estimation of cost or effort required for changes to be made. Determine if the influenced architectural elements are members of abstract or concrete PLA packages.

- The effect of a scenario is estimated by investigating which architectural elements are affected by that scenario. The cost of the modifications associated with each change scenario is predicted by listing the components and the connectors that are affected and counting the number of changes.

- The evaluation can be performed in a complete manner, if the set of identified scenarios is complete. If all scenarios are executed without problems, the quality attribute of the architecture is optimal.

- The evaluation can be performed in a statistical manner, if a representative set of scenarios has been considered. The ratio between scenarios that the architecture can succeed with and scenarios not succeeded with well by the architecture provides an indication of how well the architecture fulfills the software quality requirements.

In case the analysis is performed at a time when PL has been already developed and multiple releases exist, it is possible to define and use a measurement instrument to express the effect of scenarios. The instrument must indicate not only the impact of a scenario considering both the flexibility in space (multiple variants in products) and time (multiple versions of variants), but also whether multiple owners are involved. The analysis of flexibility in time could indicate whether it leads to versions conflicts.

The objective of the evaluation is to get a prediction of the quality of the PLA with respect to the anticipated variability in functional or non-functional characteristics of this product line.

5.2.5 Scenario interaction

The result of the effects evaluation may represent the input for this last step where *scenario interaction* is revealed. The activity is to determine which scenarios interact, meaning that they affect the same component. High interaction of unrelated scenarios could indicate a poor separation of concerns.

6. Case study

Earlier in this report, we have discussed theoretical aspects related to software product lines and architecture assessment. An analysis strategy for product line architectures is very hard to discuss at an abstract level. Instead, one needs a concrete example of a product line initiated in a revolutionary approach in software development. In this chapter we will present a case study of a product line architecture that will be used throughout the rest of the chapters. In the beginning, this chapter introduces the scope of the concrete software product line. The products that are planned to be part of the product line need to exhibit sufficient commonality and it should be possible to handle the variability in a structured fashion. In order to describe the product line architecture, we will first present a brief analysis of the common and variable features of the product members. The last part exemplifies the available views of the product line architecture representation.

6.1 Scope of the product line

Our case study is a PLA representation of scientific on-board silicon X-ray array (SIXA) spectrometer control software [43].

SIXA is a multi-element X-ray photon counting spectrometer. It consists of 19 discrete hexagonally-arranged circular elements and specific domain hardware architecture. The SIXA measurement activity consists in the observations of time-resolved X-ray spectra for a variety of astronomical objects.

The instrument is programmed and operates using a set of commands sent from the ground station to the satellite. The role of a software spectrometer controller is to control the following measurement modes:

- Energy Spectrum (EGY), which consists of three energy-spectrum observing modes: Energy-Spectrum Mode (ESM), Window Counting Mode (WCM), and Time-Interval Mode (TIM).

- SEC, which consists of three single event characterization observing modes: SEC1, SEC2, and SEC3.

Each measurement mode could be controlled individually. A coordinated control of the analog electronics is required when both measurement modes are on.

A striking similarity among the requirements of each measurement controller makes the initiation and development of a PL possible. The PL domain is structured in packages of features based on the requirements specification. The common and variable aspects of features are mapped onto common or specific packages. In order to initiate the PLA we consider the first software product members to be the following:

- EGYController, which includes specific features of a standalone control of EGY measurement mode;

- SECController, which includes specific features of a standalone control of SEC measurement mode;

- SECwithEGYController, which includes specific features of coordinated control.

6.2 Analysis of the commonalities and variabilities of the PL requirements

A brief analysis of the commonalities and variabilities of the PL requirements is synthesized in Table 6.

Table 6. Common and variable features of the spectrometer controller PL.

<table>
<tr><td colspan="3">Common Features - Hardware Platform:
• DPU: M68000 processor (8MHz, 1 wait state), VME-bus architecture.
• 4MBytes SRAM for program variables and storing spectra.
• 128 kbytes EEPROM for saving program.
• 64kbytes for parameters.
• SRAM and EEPROM error corrected: single bit SRAM errors are hardware corrected, double bit errors and single bit EEPROM are indicated to software.
• opto-coupled serial links to a ground support equipment (EGSE).
• satellite interface unit (SIU).
• Analog control, analog electronics, detectors.
• Watch dog timer (WDT) to prevent the processor from causing damage to the instrument if it breaks out of control.</td></tr>
<tr><td colspan="3">Variable features</td></tr>
<tr><td>EGY Controller</td><td>SECController</td><td>SECWithEGYController</td></tr>
<tr><td>-</td><td>1. A 40 Mbyte (or perhaps bigger) hard disk to store scientific data. For reliability reasons there are two hard disks (Conner CP2040) with their own SCSI bus.
2. WDT function for SEC DPU to enable the software to continue observations from the next target after an error and maintain data on hard disk that was stored there before the error.
3. Hard disk fault tolerance.</td><td>1. If SEC is on it has control over the analog boards, otherwise EGY DPU has control;
2. Duplicate HW elements:
• DPU,
• VME buses (one bus for each DPU),
• 2 independent duplicated interfaces to the data transfer bus of the satellite (A1, A2, B1, B2),
• 2 SIU and
• 1 EGSE</td></tr>
</table>

Table 4. Common and variable features of the spectrometer controller PL continues.

Common Features - HW/SW Interfaces

- SIU interface contains a special dual port buffer memory for the transmitted and received messages. It generates 1 Hz SYNC pulse interrupts from the space craft clock.
- SRG-bus interface defines the messages between SIXA and the ground. It is used for routing several logical data flows. It has two separate buses.
- A BIUS sends ground commands to SIXA through SRG-bus.
- Housekeeping data.
- SRG-bus control commands could be sent individual or broadcasted.
- Science data transmission from science data files to ground.
- READ_Memory file structure depends on the EEPROM structure. The coding data types in ESM/SEC/READ_Memory files are specified.
- Parametrized common features:

1. File structure is function of *(file_id, file_header, number of target data blocks).*
2. File_header contains: *(file header length=423 words, calibration data, measurement programs).*
    3. Calibration data (76 words) contains ((slope, offset) for each of the 19 detectors, number of targets).
    4. Measurement progam (23 words) contains *([target (coordinates, number)], [status bits (SEC1,2,3, TIM,WCM,ESM)], [(start, end) time], number of events, exposure time, sampling time, T1, T2, [window 1,2 (low , high)]).*

Table 4. Common and variable features of the spectrometer controller PL continues.

Variable features – HW/SW Interfaces		
EGYController	**SECController**	**SECwithEGY Controller**
• Target data in ESM file contains ESM,WCM,TIM spectra for each repetition. • Number of spectra = number of repetition* number of modes selected. • ESM and WCM spectra lengths depend on the detectors number (19 respectively 7).	• The number of the events of the measurement program is significant for SEC only. • Target data in SEC file - maximum SEC spectrum size is less than 4 Mbytes.	• Contains two separate bus groups - group A (EGY) and group B (SEC). • BIUS duplicates ground commands to both DPUs • Broadcasting order: A1, A2 ,B1, B2; 200 microseconds' delay between transmissions
Common Features - Ground commands • Ground commands are divided into regular, service, and special. • A PARAM command determines the mode of SIXA operation for each target and contains measurement programs of up to 6 targets. If more targets will be observed then multiple PARAMs are needed. • The number of targets, which can be observed during one observation period is equals to SIXA_MAX_NUMBER_OF_TARGETS.		

Table 4. Common and variable features of the spectrometer controller PL continues.

<table>
<tr><th colspan="3">Variable Features – Ground commands</th></tr>
<tr><th>EGY Controller</th><th>SECController</th><th>SECwithEGYController</th></tr>
<tr><td>-</td><td>Add:
SetDiskFull - a new command in service commands (abnormal situation) group,
A new parameter related to disk in ChangeParameter command,
Tests for disk in StartDiagnosis command and
Hard disk diagnostics report in housekeeping data structure.</td><td>Considers:
StartDiagnosis ground command is for only one DPU_id.
BIUS multiplies (double) a start calibration command.
In order to preserve the consistency of data both DPUs have the same detector parameters and measurement table parameters.
Add
DPU-id field to WriteMemory command.
Identify
SEC or EGY in HK report.</td></tr>
<tr><td colspan="3">Common Features - Functional modes:
Essentially the same state behavior.
Common to EGY and SEC: The initialization of energy window parameters in WCM and SEC_2 for detector elements depends on the detectors number.
Finish EGY or SEC modes:
FOT command given by BIUS.
An exceptional condition is fulfilled (ERB, BAT, ECO, SFI, BGC).</td></tr>
</table>

Table 4. Common and variable features of the spectrometer controller PL continues.

Variable Features – Functional modes		
EGYController	**SECController**	**SECwithEGYController**
1. ESM,WCM **AND** TIM modes. 2. Measurement function • Calculate exposure time (ET) for ESM, WCM and TIM. • The SIXA crystal influences the accuracy of ET. 3. *Finish EGY:* • last repetition is completed (full spectra collected for all modes).	1. SEC1, SEC2 **OR** SEC3 mode. 2. Add TestDisk to the *power_up tests* and modify POWER_UP_TEST_TIME 3. *Diagnostic function* in STANBY state is completed with hard disk specific tests. 4. Measurement function • Calculate the sampling time (ST) of the window counters • The SIXA crystal influences the accuracy of ET. 5. *Finish SEC:* • event limit is reached • T2-time of the last SEC repetition expires.	1. Modify TEST_SRG bus_interf in *power up tests* suite. 2. *Housekeeping function:* The interface from the processors to the analog control is such that the SEC DPU can read the voltage and current values if both DPUs are powered. Therefore the EGY DPU will set the voltages and currents to zero in its HK-reports when both DPUs are on. If only EGY is on then it will report voltages and currents normally. 3. Measurement function • two SIXA crystals 4. *Start:* a SEC-mode starts simultaneously with the energy (ESM; WCM and TIM) modes for a target.

Table 4. Common and variable features of the spectrometer controller PL continues.

Common - Performance		
1. SRG-bus response times to all control commands is 200 microseconds. 2. Maximum average Event arrival rate from detector rate is 10 000ev/sec. The SEC1 mode is critical - the absolute limit is 100 000ev/sec. 3. Memory refreshment cycle due to errors due to galactic cosmic rays. For HM62256 (32kb) has been estimated 15er/device-day. Three devices are needed for one word. Consequently the average is less than 2 errors/hour/32 kwords. A refreshment Cycle of 1 hour for the memory is adequate. 4. Time of a self test is 1 minute or 15 (acc.to specialists) 5. Resource requirements: program code may not exceed 64k, max 32k of parameters.		
Variable Features		
EGYController	**SECController**	**SECwithEGYController**
-	It must be possible to read science data from the disk and send it to the SRG-bus simultaneously with the specified SRG bus science data rate of 800 kb/sec.	The reunion of EGY and SEC.

Table 4. Common and variable features of the spectrometer controller PL continues.

Common Features - Safety

Safety requirements detail those precautions necessary to prevent the on board software from causing damage to SIXA experiment or to its environment.

- Guarding against faulty operator inputs: CPAR and WMEM commands may cause damage if they include erroneous values. Therefore some checks for these commands are necessary. Checks for CPAR include parameter number, value and index. Checks for WMEM refer that the checksum is correct.
- Guarding against excessive power consumption: the latching relays on TCU consume power excessively if the length of the AON or AOFF pulse significantly exceeds 10 milliseconds.
    - SW must limit the length of AON/AOFF pulse from a DPU to LATCHING_RELAY_CONTROL_ TIME.
    - For safety AON and AOFF bits must be cleared in a startup program.
    - SW must give an AOFF pulse after a power up reset.
- Guarding against unintentional EEPROM programming. - processor breaks out of control due to a SEU. There is a hardware protection before programming of a word. A EEPROM-programming SW function must be designed so that before each EEPROM write, it checks that a WMEM command has really been issued.
- Guarding against detector overheating. There is a HW protection. SW monitors the detector voltage and shuts high voltage off, if temperature rises.
- Guarding against constant SRG-bus reservation by a constant long transmission. A watch dog timer expiration will generate a reset that initializes SIU by HW.
- Guarding detector against high voltage. SIXA detector may be destroyed if high voltage is switched on when detector temperature is above SIXA_DETECTOR_MAX_TEMPERATURE. There is a HW protection circuit.
    - For maximum security, the SW monitors the detector temperatures and does not try to switch HV on in case the temperature is above this limit.
    - Two delays ANALOG_ON_TO_HV_ON and HV_OFF_TO_ANALOG_OFF.
    - Delay ECO command to switching power off is HV_OFF_TO_ANALOG_OFF.

If watch dog timer expires and resets SIXA during observation when both analog electronics and HV are on. SW sequence after power up: switch HV off; wait HV_OFF_TO_ANALOG_OFF time, switch analog off.

Table 4. Common and variable features of the spectrometer controller PL continues.

Common Features - Reliability

Represent software features for ensuring a reliable operation of SIXA in presence of HW faults and errors in HW operation.

- WDT: A HW reset caused by WD is determined by SW by reading a flip flop, which is set due to this reset.
- Memory fault tolerance.
    - EEPROM: exception vector table, startup program, program copies, parameter copies, runtime EEPROM fault tolerance.
    - RAM: storage area, runtime RAM fault tolerance.
- SRG-bus fault tolerance: two busses for each DPU: A1,A2, B1, B2.
- Hard disk: 2 HD and 4Mb RAM.
- High voltage source fault tolerance: two sources.

The first step for a PLA design is the definition of the product line domain. The domain should be stable. A possible method adopted for establishing domain delimitation is FORM. This method is based on defining and clustering products features (i.e. capability, operating environment, domain technology and implementation technique features) based on domain requirements. The result is a features tree, which is a useful input for the next step in software development, the architecture design. In the FORM method, features are structured in four categories:

- capability features, which represent high level behaviors of the software, like services, operation and non-functional features;

- operating environment, which represents environment-specific information that is used for defining computational function in the domain; includes hardware and software related features.

- domain technology, which represents computational functions in the domain specifically for the way of implementing services and operations.

- implementation techniques, are similar to domain technology, but are more generic and may be used in other domains.

Various relations exist among these features, such as generalization, aggregation, utilization, and mutual dependency. Features themselves could be mandatory, optional and alternative.

The resulting features model is very complex. Clarifying the domain boundaries and standardizing domain technology must take place before this modeling process.

6.3 PLA representation

PLA design is represented by different views, which have to be considered in the analysis of its main structural qualities. Among these the conceptual view and detailed functional decomposition view could be enumerated. These are described in the next sections. We also introduce a diversity view, from the necessity to represent the common and variable elements of different product members in a unitary structure.

6.3.1 Conceptual view of the PLA

Figure 30 shows the conceptual view of the spectrometer controller PLA. This view is the result of a functional-based decomposition and it includes relations between different functional modules of the PL. The architectural components are large, functional (domain) entities, and the connectors are *"uses", "command" or "passes-data-to"* relations. This structure is useful for understanding the interactions between entities in the problem space, for planning functionality and for understanding the domain perspective, and hence thereafter, the possibilities for creating a product line.

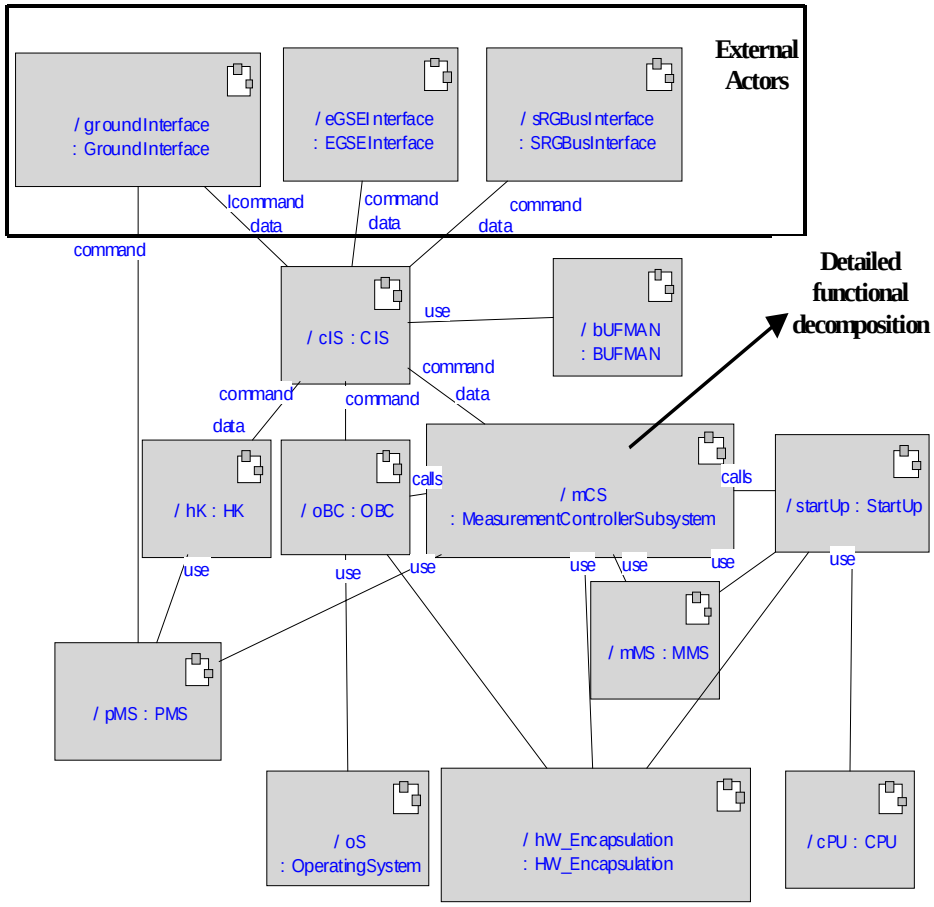

Figure 30. Conceptual view of the spectrometer PL.

The architectural elements are:

1. Measurement Controller Subsystem (MCS) which has the main role in controlling acquisition and dumping science data. It requires services provided by other components of the system.

2. Housekeeping (HK) forms the housekeeping reports and sends them to a command interface, when requested by command from the command interface subsystem. It uses services provided by PMS.

3. Command Interface Subsystem (CIS) hides the hardware buses' interfaces from the rest of the software. It delivers a command from the interfaces to the appropriate subsystem and routes the response to the command to the interface from which the command originated.

4. On-board clock (OBC) maintains an on-board clock used for time-stamping spectra in data files. It includes services for timing the start/stop of spectra and targets. It also provides other timing related services.

5. Memory Management Subsystem (MMS) provides services for handling the storage RAM, program and parameter EEPROM areas, memory refreshment and memory error exception.

6. Parameter Management Subsystem (PMS) provides services for initiating, changing and reading the on-board parameters in EEPROM.

7. Start-up program (StartUp) implements the power up and watchdog timer start-up functionalities.

8. Communication buffer management (BUFMAN) provides services for allocating/deallocating transmit buffers.

9. CPU-specific services (CPU) provide highly optimized high speed assembly language services (high speed word copy, interrupt enable/disable).

10. Hardware encapsulation modules provide low-level services for controlling specific hardware (analog electronics, watchdog timer).

The operating system (OS) forms its own subsystem, which has not been included in the list because it is totally application independent. OS has been encapsulated to provide an easy change of the operating system if necessary.

6.3.2 Detailed functional decomposition structure

In a detailed functional decomposition structure the main elements are packages, components, ports, and protocols. The static relations between components are association, specialization, generalization, etc. Considering the dynamic

relations, statechart diagrams and message-sequence charts (MSC) are also part of this view.

The detailed design of the conceptual view of the measurement controller component is presented in Figure 32. In this view abstract components of the PLA are included.

- The MeasurementControl component, which is responsible for starting and stopping the operating mode for data acquisition according to the commands received from the command interface and according to the events generated in other parts of the software.

- DataAcquisitionControl component collects events (science data) to the spectra data file during the observation of a target. The component includes as well as hides data acquisition details.

- Data File Management provides interfaces for storing science data - opening/closing/writing the data files, hiding data storing details. The component also provides interfaces for controlling the transmission of the stored data to a command interface.

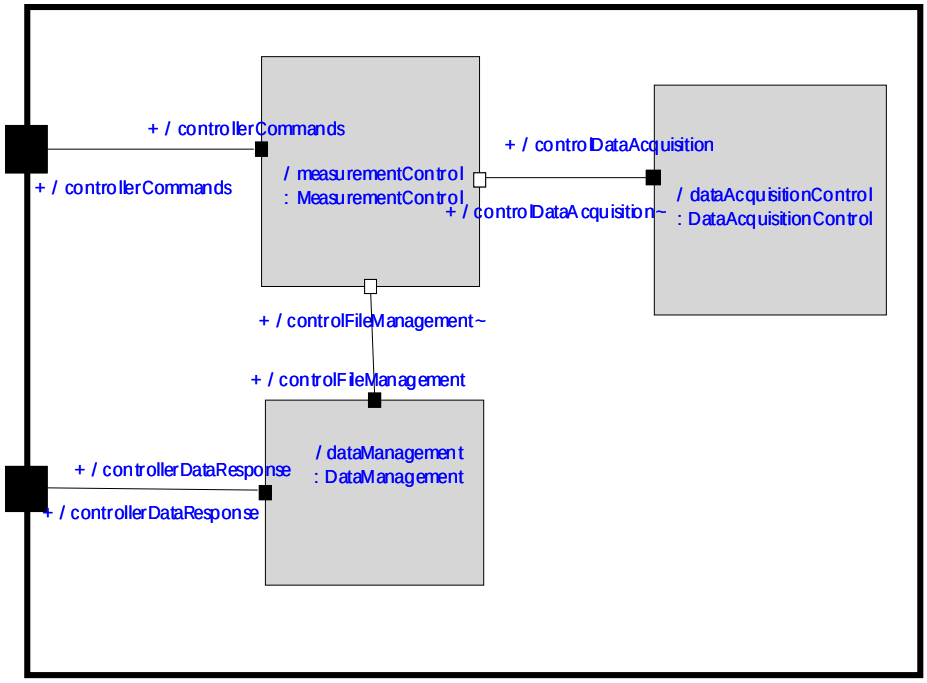

Figure 32. The detailed functional decomposition of the abstract components of PLA.

6.3.3 The diversity view of PLA

The diversity view of a PLA is introduced from the necessity to represent the common and variable elements of different product members in one view. This view is the one which makes the difference between PLA and the architecture of one product. This view must describe the stability of the PL domain, because the architecture is one of the reusable assets of the PL.

Generally, the boundary of the PLA must be defined before the beginning of the activity of its detailed design. We can identify an inclusion relation between domain architecture, product line architecture and a single product architecture. A product line architecture is included in a domain architecture and includes single product architecture. From this point of view a PLA must represent part of

the abstract features of the domain, but it should also be easy to identify the concrete features of the architecture of one PL member in its views.

The diversity view of a PLA for the design of an on-board X-ray spectrometer control software is presented in Figure 34. This view associates the generic with the concrete. Looking top-down, AbstractSpectrometerFeatures encapsulated in the measurement component are decomposed in three abstract components: MeasurementControlDataAcquisitionControl, and DataManagement.

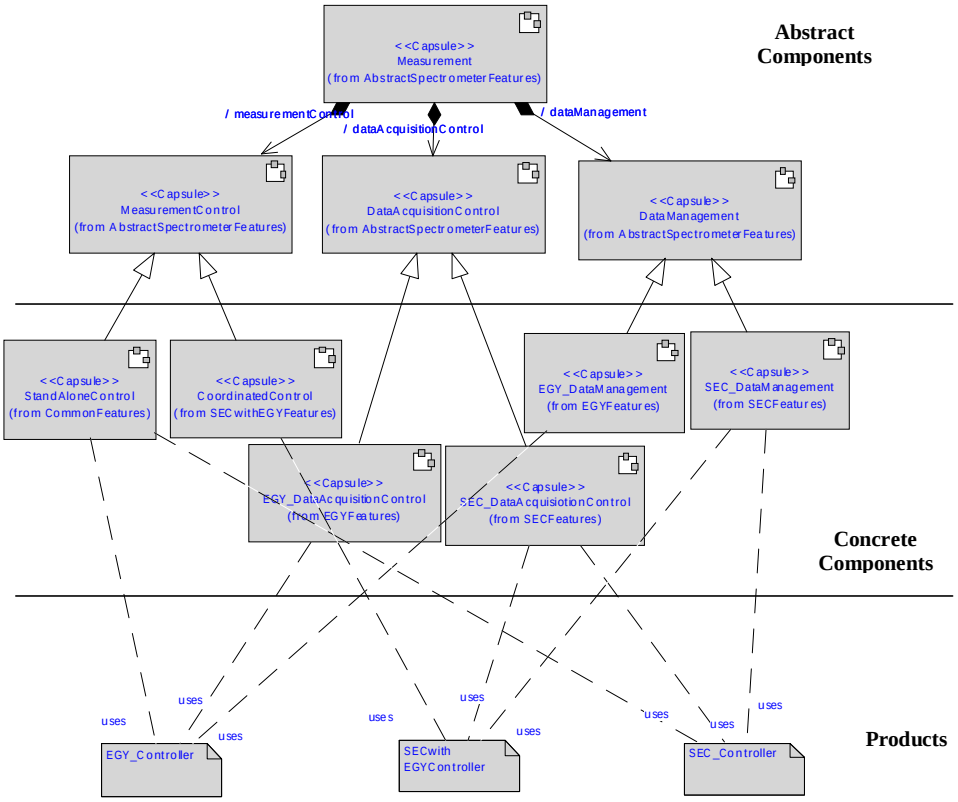

Figure 34. PLA Diversity view.

In each component, abstract features of the corresponding subdomains are collected, which are subsets of the Measurement domain abstract features. For

each product, which is a member of the family, each of the three abstract components is specialized in a concrete component. For example, MeasurementControl is specialized in StandAloneControl and CoordinatedControl, DataAcquisitionControl is specialized in EGY_DataAcquisitionControl and SEC_DataAcquisitionControl, and DataManagement is specialized in EGY_DataManagement and SEC_DataManagement.

The diagram includes the *uses* relation which is directed from products to concrete components, providing, in this way, the information on the reusability of each component. The same information could also be represented in tabular form, as in Table 8. The products of the group are vertically distributed and the components are dispersed horizontally. Each cell t_{ij} of the table is marked if product P_i uses component C_j. For example, two products, EGYController and SECController, use the StandAloneControl component.

Table 8. Reusability of the concrete components in a PLA.

Component→ Product↓	Stand Alone Control	Coordi-nated Control	EGY_Data Acquisition Control	SEC_Data Acquisition Control	EGY_Data Manage-ment	SEC_Data Manage-ment
EGY_Controller	x		x		x	
SEC_Controller	x			x		x
SECwithEGY Controller		x		x	x	x

7. Experiences from spectrometer controller PLA analysis

In the previous chapter we introduced our concrete example of a product line initiated in a revolutionary approach in software development. Regarding the PLA assessment, we studied and discussed many aspects at an abstract level. In this chapter we will discuss our experiences of theoretical thinking on the case study introduced previously. From a practitioner point of view, the PLA analysis strategy for reusability and modifiability will be exemplified in detail. Finally, we will conclude with the results of our practice.

7.1 Reusability strategy

One important benefit from the PL approach is reusability. A reuse-based software development process requires specific methods adapted for this context. It is well known that a method is considered as a predefined and organized collection of techniques and a set of rules which state *by whom, in what order, and in what way the techniques* are used to achieve some objectives. Rules are also included in the technique. However, here they define how the representation of the system is derived and handled using a conceptual structure and a related notation. Software reuse could be approached from the composition perspective. A component could be unrestricted to a code component, but it could also be some other artifact, such as software schema or software architecture. In software development other reusable elements could also be seen: domain models, and analysis and design documents.

Reusability is a difficult property to assess. This quality attribute could be seen as a balance between generality and specifics and is the main driver for product line development. First, the architecture and its components should be general because they should be applied in other similar situations. Secondly, the architecture should provide concrete functionality that supplies a considerable advantage when reused.

To evaluate this attribute in the case of one software product, scenarios are the only approach for the analysis. In this regard, typical reuse situations in a

domain are depicted in a set of scenarios. The concentration on a specific set of software products and on specific reuse scenarios allows eliciting information about the flexibility of software architecture and its constraints. The architecture is analyzed with respect to each scenario and the result assigns the effect to a measure, which could be expressed as unchanged, slightly changed, and new components. Also, the effect of a scenario could be the ratio of components reused *as-is* and the total number of components.

The product architecture is analyzed by asking the typical question for reusability: *'How much of the software can be reused?'* in the context of each reuse scenario. The effect on the architecture is evaluated in a statistical manner. This means that every scenario is assigned a quote number of affected components in the scenario, divided by the total number of components in the current architecture. The result should be as close as possible to one (as many of the components as possible should be reusable *as-is*).

A product line needs a strategic method. The method must not simply consider the reusability of a single product and then re-engineer its architecture for each other product. It is also recognized that better results are obtained if one quality attribute is not analyzed in isolation, but its interactions with other quality attributes are considered instead. Furthermore, a product line concept brings along new aspects like commonality and variability, and their interaction should be considered, too.

Considering the aspect of time, we can identify two important parts in the life cycle of a PL. One is the initiation moment of the PLA and the other is its evolution. PLA is initiated from a set of existing product features and the variability in space is thereby present. The other one is the evolution, where variability in space is multiplied by the variability in time.

Our strategy could be applied for the initiation of a PLA. The technique is based on the abstract features of all the considered products. To achieve both precision and abstractness, we will specify the reusability aspects in terms of groups. A reusability *with* reuse is a commonality held uniformly across the given group of the products of the PL. A reusability *for* reuse is a variability true of only some members of the PL. In our opinion, the reusability aspects are associated with the commonality and variability analysis method SCV [22].

The rules are similar to factoring algebra. The common factor in a set of two formulas $(x+y)^2$ and $(x^2+y)(x+y)$ is $(x+y)$. The remainder, $(x+y)$ and (x^2+y), are variables. If we introduce a small change, such as replacing a minus with a plus in the first equation, then $x+y$ is no longer the common factor. Also, if the equations are rewritten as $x^3+x^2y+xy+y^2$ and $x(x-y)+y(y+3x)$, they are algebraically equivalent, but the commonality and variability are much harder to identify. By extracting the common features we obtain abstractions based on the composition principle and the remaining parts (the one from parenthesis) are the variable aspects.

In a PLA, both aspects of reusability are present. The diversity view of PLA presents the 'with reuse' aspect in the concrete components used by each of the product members. In the diversity view diagram, the number of the 'uses' relations associated to one concrete component measures the degree of reusability in the PL of that component. For example, the degree of reusability for the SEC_DataAcquisition component is 2 and for the Coordinated_Control component, 1.

The with-reuse aspect of reusability is described in the PLA by the abstract features of the PLA. The abstract features encapsulated in three main abstract components MeasurementController, DataManagement and DataAcquisition, are completely reused in all the product membersThe PLA diversity view contains both aspects regarding reusability encapsulated in components. From the point of view of other reusable assets, which could be represented by an architecture view, the organization in packages of features could be a possible solution. Figure 36 describes the mapping of product features into packages and the relations between packages for the spectrometer controller PL.

The AbstractSpectrometerFeatures package has the highest degree of reusability but also the highest degree of dependability. The abstract features depend on the commonality between EGY and SEC features. A change in the problem domain of one of the three products is mostly reflected in the degree of reusability of the abstract domain features.

.

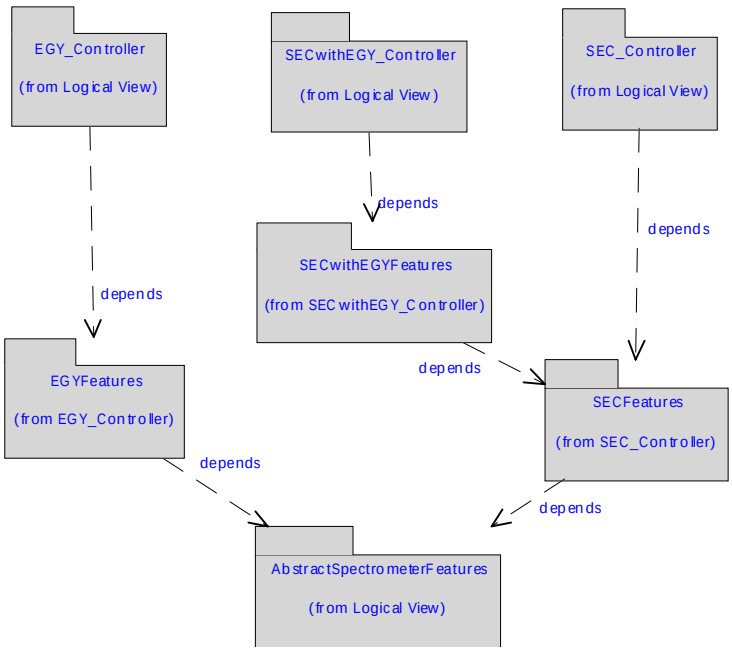

Figure 36. Mapping the product features into packages.

7.2 Modifiability strategy

7.2.1 Change scenarios for spectrometer controller software PL

***1**:* Receive the target coordinates from a STAR TRACKER (ST).

Effect on the architecture: The target coordinates received from a ST will overwrite the coordinates given previously with the COORD-ground command that contains the pre-planned (intended) coordinates. This scenario requires a new protocol for the Measurement Control component to interface with the ST. The protocol is included in the abstract features package, because it is a common feature of the two concrete components; StandAloneControl and CoordinatedControl.

Result: Modify one component in the detailed functional decomposition view.

2: Change SIXA parameters (i.e. change the number of detectors).

Effect on the architecture: This is a hardware modification scenario, which is reflected in the CPAR command of the ground interface. The measurement control subsystem uses the service on a parameter management subsystem (PMS). This scenario requires modification in PMS – in this way the measurement control subsystem is decoupled from any changes in the hardware-specific domain.

Result: Modify one component in the conceptual view.

3: Change the number of EEPROM banks.

Effect on the architecture: This is a hardware-specific change scenario. The program code is stored in the EEPROM memory, which is organized into 4 banks. The Read memory (RMEM) command contains the information representation of the physical EEPROM addresses. The measurement control subsystem is decoupled from the memory organization by a startup subsystem, which has the role of executing the RMEM command.

Result: Modify one component in the conceptual view.

4: Add a hard disk for a SEC product.

Effect on the architecture: The SEC_controller and SECwithEGY_controller contain a HD for data storage. This change scenario requires a lot of changes at the architectural level, most of them related to the DataManagement component. A detailed description of this scenario analysis is contained in the next section.

Result: Multiple changes in detailed functional decomposition, localized in the SECFeatures.

5: Change the generator polynomial (from CCITT polynomial) for 16 bit CRC sum of errors handlers.

Effect on the architecture: A memory management subsystem (MMS) consists of service functions for managing the storage RAM and EEPROM. It also

includes a state for refreshing RAM and the memory error exception handlers (double and single bit).

Result: Modification in one component in the conceptual view.

6: Add a new command – SetDiskFull – in the service commands group of ground interface.

Effect on the architecture: The command is specific to the software configuration with the hard disk. It is supposed that the DiskManagement component is already added. The measurement control component has the role of interpreting this new command and starting the required action. This scenario requires the addition of a new signal to the controller commands protocol and data management protocol. A new transition states transition in the DataFileManagement component.

Result: Changes in the detailed decomposition view.

7: Specific hardware configuration (hard disk capacity or RAM capacity) changes.

Effect on the architecture: If the capacity of the hard disk or the amount of fault-free RAM in use for SEC spectrum storage is changed, then the calculation of the SEC event limit used in SEC1 and SEC2 modes is changed.

Result: This scenario requires no change at the architecture level.

8: The type of hard disk is changed.

Effect on the architecture: The hard disk used in the implementation is a CP2040 disk. This scenario is applicable for the software products with SECFeatures included. Reading data from the disk during dumping is time-critical because the average SRG-bus speed of 744kbit/sec must be maintained. The DISK driver must be optimized so that it makes maximum use of the internal 16 kbyte internal cache of CP2040.

Result: Modify one component in the conceptual view.

9: How is the control of analog electronics changed on behalf of several spectrometers?

Effect on the architecture: In case of several spectrometers, only one of them takes control over the analog electronics. This feature is included in the CoordinatedControl component.

Result: Modification of one component in the detailed functional decomposition.

10: How is the architecture affected when the operation mode is changed?

The operation modes of different products are one of the variabilities of the PLA. This is encapsulated into DataAcquisition and DataFileManagement Components. The variability is expressed in Table 10.

Table 10. Architectural changes in the detailed functional decomposition.

Components **PL members**	**DataAcquisition**	**DataFile Management**	**Protocols**
EGY products consider three simultaneous operation modes: ESM, WCM, and TIM	*Behavior:* EGY_DataAcquisition is specialized in behavior to handle full spectra collection for all operation modes. The acquisition is finished when the last repetition is completed *Structure:* add a new port savedEGYData	*Structure:* add a new port savedEGYData	Add a new protocol to the EGY features package: savedEGYData
SEC products can operate in one of these three modes: SEC1, SEC2, SEC3	*Behavior:* SEC_DataAcquisition collects spectra data continuously based on a sampling time. Must check if the buffers are full. *Structure:* add a new port savedSECData	*Structure:* add a new port savedSECData	Add a new protocol to the SEC features package: savedSECData

Figures 20 and 21 describe the specialization in the behavior of the components to handle the variability in functionality of the product members of the PLA.

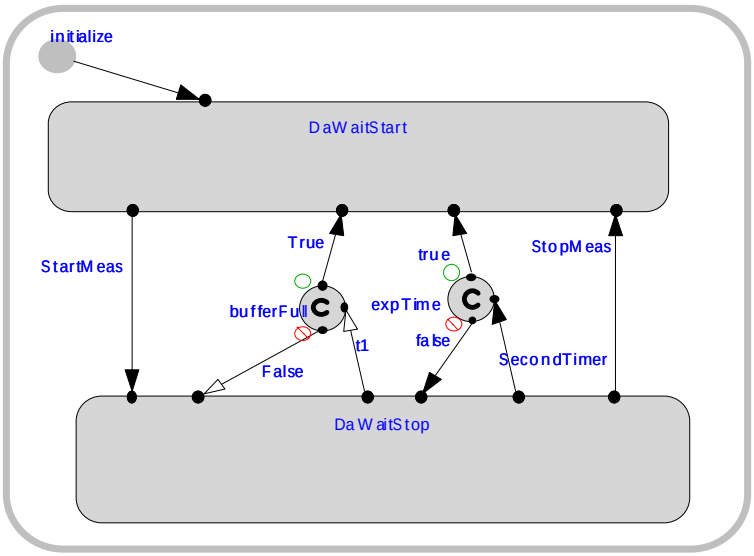

Figure 38. Statechart diagram of the SEC_DataAcquisition concrete component.

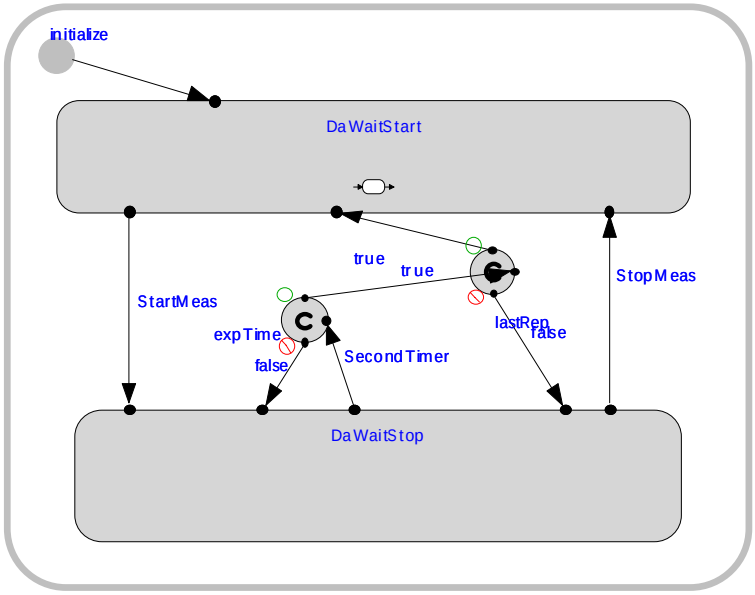

Figure 39. Statechart diagram of the EGY_DataAcquisition concrete component.

Effect on the architecture: The measurement control component is decoupled from the operation mode of different products, which is encapsulated into the DataAcquisition component.

Result: No change to the PLA – abstract concrete or features of measurement control.

11: How is the average SRG-bus speed of 744kbit/sec on reading data from disk, which is time-critical, maintained?

Alternative solutions:

1. Change the HD: Use Fast disk: Optimal disk interleaving factor and storing the data file in sequential sectors on the disk.

2. Send filler blocks to the bus while waiting for the disk – a sufficient number of filler blocks could be reserved in the vector word sent in advance to BIUS.

3. Use a busy bit of SRG-bus.

4. Optimize disk driver – If the disk drive has been changed, the software has to be tuned separately for the new disk.

Result: Not applicable to the available views.

12: Change the CPU.

Effect on the architecture: CPU-specific services provide highly optimized high speed assembly language services (high speed word copy, interrupt enable/disable, etc.). The services are not applicable at the level of description.

Result: Not applicable to the available views.

7.2.2 A summary of the analysis PLA for modifiability

We defined twelve change scenarios for different categories of changes. The effect of the scenarios and the required PLA view are summarized in Table 7.

Table 12. A summary of the PLA analysis for modifiability.

Category	**Scenario**	**PLA view**	**Effect**
General-purpose hardware changes	12. Change the CPU	Not applicable	No change
Domain-specific hardware changes	2. Change SIXA parameters.	Conceptual view	1 component
	3. Change the number of EEPROM banks.	Conceptual view	1component
	4. Add a hard disk for a SEC product.	Detailed functional decomposition	Localized to SEC features
	7. Specific hardware configuration changes.	Not available	No change.
	8. The type of hard disk is changed	Conceptual view	1 component
Changes in technology	5. Change the generator polynomial	Conceptual view	1component

Table 7. A summary of the PLA analysis for modifiability continues.

Changes in functionality	10. Change operation mode	Detailed functional decomposition	Localized at the concrete components of PLA
	6. Add a new command	Detailed functional decomposition	Localized at the concrete components of PLA
	9. Multiple spectrometers analog electronics control	Detailed functional decomposition	Localized in Coordinated Control
Change in non-functional requirements	11. How is the average transmit speed maintained?	Not applicable to available views	Localized in the Disk Driver concrete component
Other changes	1. Receive the target coordinates from a STAR TRACKER .	Detailed functional decomposition	1 component

7.2.3 Detailed analysis of scenario categories related to hard disk

The hard disk is a specific feature of the SEC_Controller spectrometer product. Two scenarios could be considered in the category of hardware changes: the addition of the hard disk and the change of the type of hard disk.

7.2.3.1 The addition of the hard disk scenario analysis

In the embedded systems domain to which the spectrometer controller belongs, there is a tight relation between hardware and software, and changes in the hardware context influence most of the architectural decisions (Figure 40).

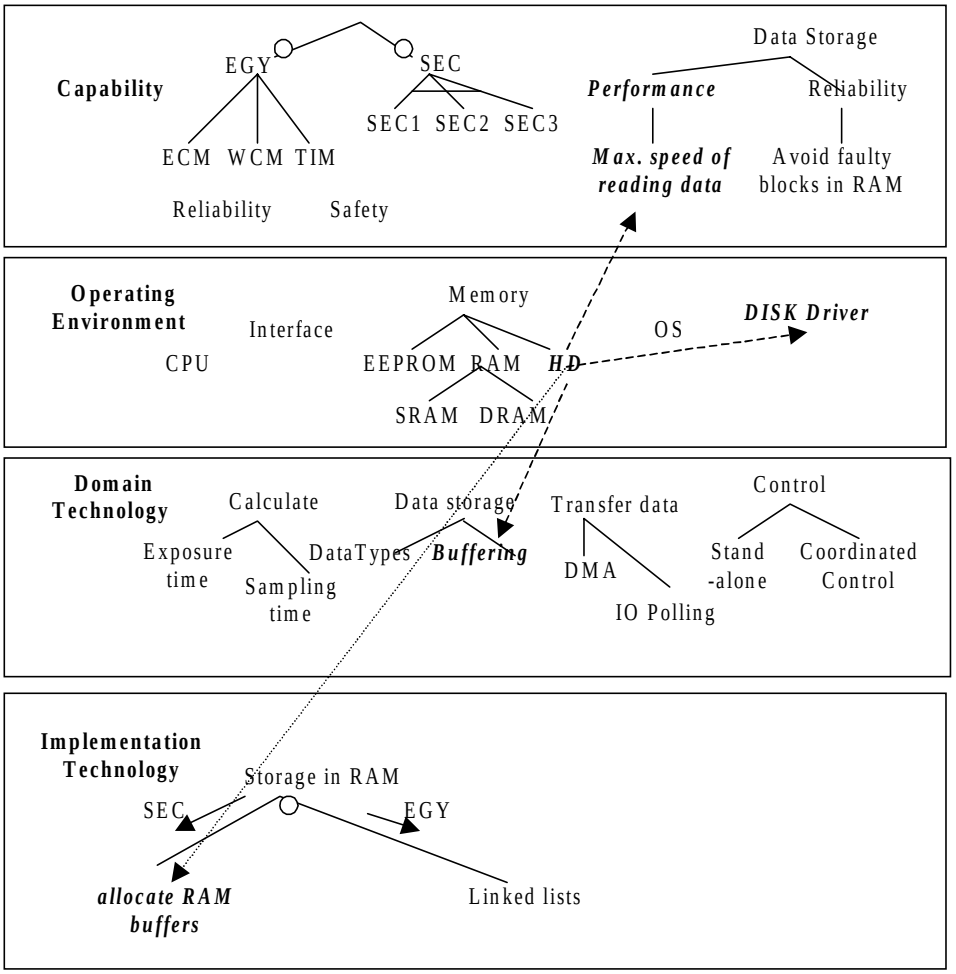

Figure 40. Adding new features in the problem domain together with the HD.

The hard disk provides one of the variables in the spectrometer controller domain. This is the parameter for the SEC_Controller products. Figure 22 describes the supplementary additions of features required by this operation in the domain. The domain features are divided into four categories: capability, operating environment, domain technology, and implementation techniques.

The addition of the hard disk device requires four other new features to be considered in the tree. These are:

- An SCSI Driver in the same category,

- A non-functional requirement for maximization of speed in reading data in the capability category,

- Buffering for data storage in the domain technology category,

- Storage of data in RAM at successive locations in the implementation technology category.

Most of these features could not be represented in the conceptual view of the architecture. The representation diagrams only contain components and connections between these components together with their external properties. The realization view should consider these new features.

From the conceptual viewpoint, all these new features must be mapped in the SEC_Controller package to different diagrams of components. The addition of the hard disk requires:

- Adding a DiskManagement component in data management,

- Adding a new protocol, DiskControl, for the connection between SEC_File_Management and Disk_Management components,

- Addition of a new signal in the ControllerCommands: SetDiskFull – a new command in service commands (abnormal situation) group, and

- A new transition in the MeasurementControl state diagram.

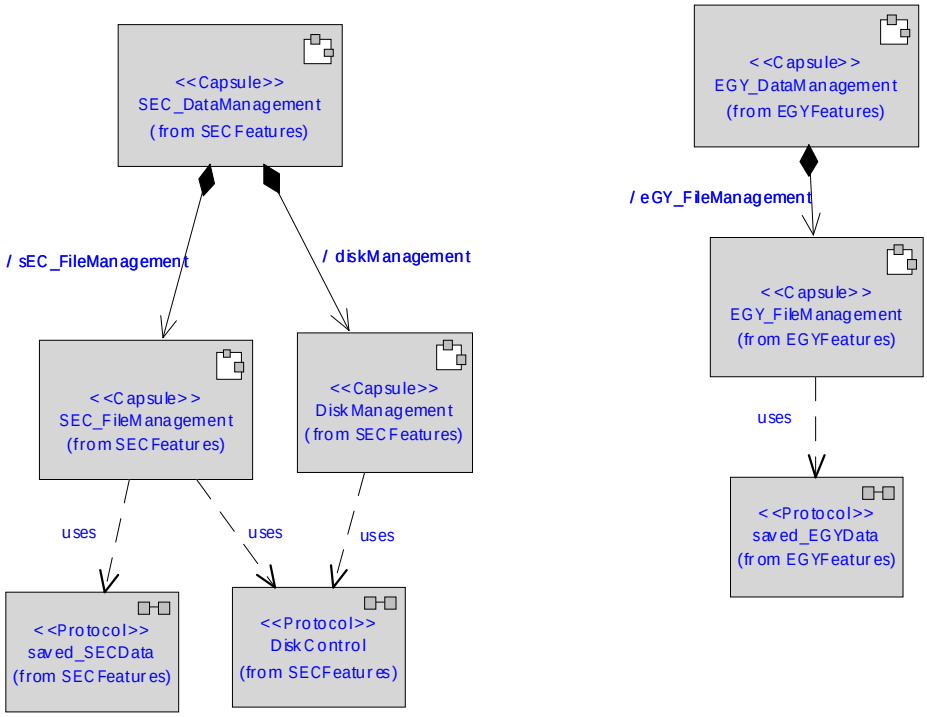

Figure 42. Specialization of the DataManagement in SECFeatures and EGYFeatures packages.

Figures 23 and 24 describe the specialization of DataManagement components for the products, which include SECFeatures or EGY features. For EGY products, the data management component encapsulates only the FileManagement component, which uses a saved_EGYData protocol. The SECData Management component encapsulates two components, SEC_FileManagement and DiskManagement. These components use saved_SECData and ControlDisk protocols.

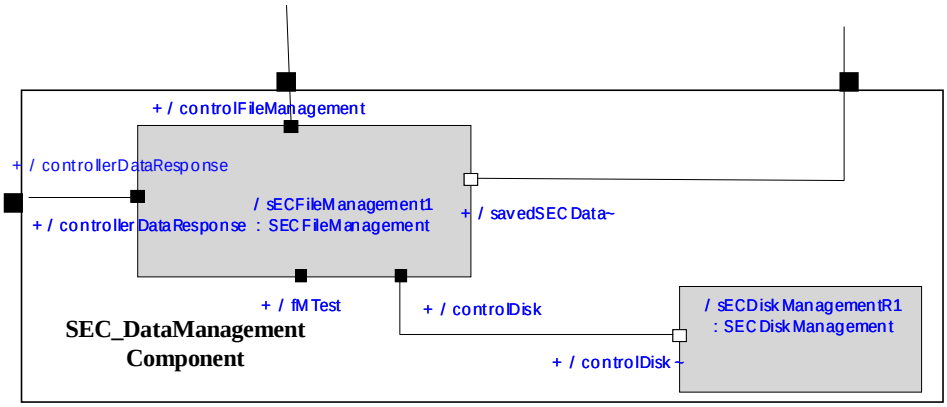

Figure 43. The structure diagram of the SEC_DataManagement component.

7.2.3.2 The change of the type of hard disk.

In the interface with hardware, the description of the architecture must include the mechanism of mapping virtual information which is used in software to physical hardware. It is the role of a hardware abstraction layer to perform this operation.

Figure 44 describes the general mechanism of decoupling software from hardware. The component whose interface must handle the variability aspects of hardware elements is decoupled from this variability by introducing a hardware abstraction layer.

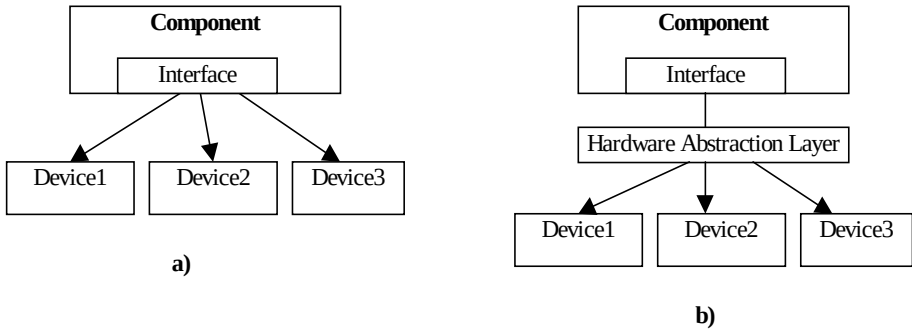

Figure 44. Decoupling of a software component from hardware.

This mechanism was applied for the case of adding the hard disk. A disk driver component, a part of the hardware abstraction layer, has been introduced simultaneously with the hard disk. The change of the hard disk type is localizing in this component.

7.3 Summary of the PLA analysis strategy

The PLA of the spectrometer controller has been represented by three main views:

- A conceptual view, considered a functional decomposition of the PLA into subsystems. The relationships between components are based on pass control and pass data or uses.

- A more detailed functional description, where the main objects are packages, components, ports, and protocols. The relationships are association, specialization, generalization, etc. Considering the dynamic aspect statechart diagrams and message-sequence charts (MSC) are also part of this view.

- A diversity view as we introduced for the reusability consideration.

A good architecture design must provide localization of changes. Most of the changes required by scenarios were applied to one component, which indicates a decoupling of concerns. The most important change was the addition of the hard disk, one variable point among PL members. This scenario required changes to the concrete components common to products which require this feature.

By structuring the PLA in abstract components which encapsulate common features of the PL members and concrete components, which in turn represents the specialization of the variable features, the effects of the change scenarios are minimized and localized.

For the moment, only scenarios could be used in the PLA analysis for modifiability. One problem with the scenario-based analysis is that the result and the expressiveness of the analysis are dependent on the selection of the scenarios and their relevance for identifying critical assumptions and weaknesses in the

architecture. There is no fixed minimum number of scenarios whose evaluation guarantees that the analysis is meaningful. According to this, we tried to use five categories of possible changes in general hardware, specific hardware, functionality, non-functional requirements and software technology. A helpful strategy in scenario elicitation is the analysis of commonality and variability. This is not a part of the architecture analysis method, but it is considered a pre-condition of it. One aim of the analysis should be to show how flexible a PLA is in order to handle the anticipated changes provided by the variability of the products. Another aim is to analyze which is the potential of the PLA to be adapted to changes in common features.

The change scenarios did not affect the PLA diversity view. This view confirms the stability of the PLA in the domain, delimited by the considered product members. The diversity view of the PLA is useful for a reusability analysis, but it is not recommended for modifiability. The results of the analysis depend not only on the views of the architecture, but also on the level of detail of the component descriptions. By using only the conceptual view, the effects of the change scenarios are reduced. On the detailed functional decomposition view, which has been developed with the help of a CASE tool, the effect is more relevant. The interaction of unrelated scenarios is lower and reveals a good separation of concerns when the functional decomposition is detailed.

8. Conclusions and future work

In this research report we have discussed our work related to the product line architecture analysis. During the chapters, we have presented our original contributions in developing this significant and, in the same time, very new research domain. In order to be able to discuss an analysis strategy for a product line architecture, it is a considerable advantage to have a good knowledge of the state of art and practice in the software architecture domain. One of our contributions is to extract the main concepts which are common to any software architectures and to present what is additional and specific for a product line approach in software development. We have also gathered together and discussed, for the first time to our knowledge, several important architecture analysis methods. The survey study represented was a very important step for defining a theoretical and general available PLA analysis strategy. In practice we simplified our analysis approach and the last part of this report has focused on our experiences with the product line architecture analysis of the spectrometer controller PL.

8.1 Conclusions related to software architecture analysis methods

The survey study has shown the real level of the research at this moment, in this domain, by presenting and discussing eight of the most representative architecture analysis methods. The work in this domain is in progress and future improvements are expected. There are also some unsolved problems and open questions. Future research work for improvement and refinement of the architecture analysis methods is therefore needed.

8.1.1 Progress identification and methods improvement techniques

8.1.1.1 Progress in risk assessment

The purpose of the evaluation is to analyze the architecture to identify potential risks by predicting the quality of the system before it has been built. Related to the potential risk identification, the reflections of this general goal have been

distinguished in all the studied methods. In this sense the uses of change scenarios and scenario interaction reveal a potential problem area of the architecture. The degree of modification captured when evaluating a system's response to a scenario represents a measured risk. Complexity of scenarios is also an important factor for risk assessment. The required changes and domain experts' experiences represent another modality of suggestion of how the system could support the risk levels for evolution or reuse. The chances of surfacing decisions at risk are optimized by using exploratory scenarios. The potential risk is also minimized by analyzing attribute interactions. Iterative methods promote an analysis of multiple resolutions as a means of minimizing the risk at an acceptable level of time and effort. Areas of high risk are analyzed more profoundly (simulated, modeled or prototyped) than the rest of the architecture. Each level of analysis helps to determine where to analyze more deeply in the next iteration.

8.1.1.2 A possible combination of methods

Looking at the existent analysis techniques the possibility to combine a coarse-grained and broad technique with a fine level one would provide an improved result, but the costs of time and effort would also be increased. Scenario-based analysis techniques can be combined with a specific analysis technique for quality attributes. For example, a scenario may identify a critical path of execution which can then be examined in detail using a real-time analysis method like RMA, or other analysis techniques for scrutinizing any dynamic properties of an application.

8.1.1.3 Metrics – more precise techniques in evaluating attributes in terms of architectures

Most of the researchers in the domain consider metrics to be more precise techniques in evaluating attributes in terms of architecture [36, 52]. Metrics specification must contain *the selected measure for a quality attribute, a measurement scale and a set of methods for measurement*. Two approaches could be identified: to adapt existing metrics [10] or to define new ones [24]. The adaptation of object-oriented metrics which were validated to be good predictors of software maintenance [49] is required because the metrics suite uses data that can only be collected from the source code, and at the architecture

level no prototype or source exists. Considering the other approach, GQM [7] is a good technique to define new metrics following a certain reasoning process. The main activities of GQM are: to define a goal in terms of purpose, perspective and environment, to establish the questions that indicate the attributes related to the goal, and to answer each question. The purpose is related to the software architecture evaluation, indication and comparison, and the end product quality prediction. The perspective depends on the aims of the assessment and it is closely related to the role of the evaluation staff, which can be that of a developer, user, management, or maintainer. *There are two suitable environments: the software architecture representation considered as an intermediate design product or as an end product itself.*

8.1.2 Existing problems and future work

8.1.2.1 Scenarios and quality attributes naming problems

One problem with the scenario-based analysis is that the result and the expressiveness of the analysis are dependent on the selection of the scenarios and their relevance to identifying critical assumptions and weaknesses of the architecture. There is no fixed minimum number of the scenarios the evaluation of which guarantees that the analysis is meaningful. According to this, the definition of a set of complex scenarios and a two-dimensional framework is a solution, but a future study of the completeness of this set and of the relative importance of each of the cells in the framework is needed. The idea to use an instrument which should include all aspects relevant to the complexity of changes, is original and useful, but the measures should be comparable to make the interpretation of the results possible.

Future studies are needed to investigate *how domain knowledge and the degree of expertise affect the coverage of the selected scenarios.* In the same direction, quality attributes prediction methods could be improved by studying their sensitivities for different variations of the inputs and how significant the used assumptions parameters are for the results. For example, how sensitive is ALPSM to the representative sample of the maintenance scenario profile, or how significant is the size estimation for the results.

An examination of the existent methods reveals a lack of understanding of quality attributes in the software engineering community at this moment. The same interpretation but with different attribute names could be identified for flexibility [48] which has the same meaning with modifiability in [34] or maintainability in [11].

8.1.2.2 Future work for methods improvement and refining

Until now SAAM has been the only method which has appeared in a book [8]. This is a confirmation of its maturity. SAAM has been used for different quality attributes like modifiability, performance, availability, and security. It has also been applied in several domains. This is another validation of its completeness. The others are still young and are undergoing refinement and improvement. Future work is needed to evaluate the effects of their various usages and to make a method repeatable of repositories of scenarios, screening and elicitation questions (ATAM). In this respect, ABASes and qualitative analysis heuristics are building up. Building a handbook of ABASes requires collection, documentation and testing of many examples of problems, quality attributes measures, stimuli and parameters.

The extension of the reengineering method for more non-functional requirements and the application of the method in more industrial case studies are the main future objectives of SBAR. The authors of this method consider it important to obtain a reasonable balance between the different quality requirements in the top-level architectural design. A small taxonomy is defined for performance and modifiability, and eight design guidelines are formulated. Each guideline is associated with a quality requirement in the taxonomy. Future work is desired to extent these guidelines to other quality requirements.

A stronger methodical integration in the development process is also required. ESAAMI needs to provide a complete support for the reuse-based and architecture-driven development approaches. Integrating the technique into reuse-based and architecture-centric development process should provide a refinement of the method.

Future research work is needed to develop systematic ways of bridging other quality requirements of the PL domain to the PLA. Several progressive steps

have been identified in research papers for the analysis of the architecture at the one product level in quality attributes interaction analysis for optimizing the design decisions [16, 6].

8.2 Closing words

This report identifies and demonstrates a method for the analysis of a PLA of a product line initialized in a revolutionary approach. The analysis strategy based on scenarios could be considered mature enough to verify PLA against anticipated changes in the requirements of various product members. Also, the diversity view introduced for reusability considerations indicates, in a structured form, the reusability degree of each component for each product in the PLA. This view is a stable element for the PLA in the domain.

Our experience shows that the PLA analysis method should be applied iteratively while the PLA design becomes more detailed. The purpose of the evaluation is to analyze the architecture to identify potential risks by predicting the quality of the PL before it has been built. Iterative methods promote analysis of multiple resolutions as a means of minimizing the risk at acceptable levels of time and effort. Areas of high risk are analyzed more profoundly (simulated, modeled or prototyped) than the rest of the architecture. Each level of analysis helps determine where to analyze more deeply in the next iteration.

The case study also requires performance, safety and reliability for each product. These quality requirements are common features for all products, but other PL domains could include variability in these aspects. These variable features must be considered from the PLA design perspective in order to minimize the risk that the final software products do not conform to these quality attributes. For PLA evaluation, these aspects are still an open question. It is important to estimate what is the degree of reuse of a PLA and what are the reusable assets when the variability of these run-time qualities is considered.

Acknowledgements

We would like to thank Mr. Tuomas Ihme, who modelled the initial product line architecture of the spectometer controllers family. His research is reported in the following report: Ihme, T., Kumara, P., Suihkonen, K., Holsti, N. Paakko, M. 1998. Developing application frameworks for mission-critical software. Using space applications as an example. VTT Research Notes 1933. Espoo: Technical Research Centre of Finland. 92 p. + app. 20 p.

References

1. G. Abowd, L. Bass, P. Clements, R. Kazman, L. Northop, A. Zaremski. *Recommended Best Industrial Practice for Software Architecture Evaluation*. CMU/SEI-96-TR-025, 1997.

2. P. America, H. Obbink, R. van Ommering, F. van der Linden. *CoPAM: A Component-Oriented Platform Architecting Method for Product Family Engineering*. Software Product Lines – Experience and Research Directions, Proceedings of the First Software Product Lines Conference, Kluwer Academic Publishers, 2000. Pp. 167–181.

3. A. Alonso, M. Garcia-Valls, J. A. de la Puente. *Assessment of Timing Properties of Family Products*. Proceedings of the Second International ESPRIT ARES Workshop, Las Palmas, February, 1998. LNCS 1429, Springer Verlag. Pp. 161–169.

4. M. Barbacci, M. Klein, C. Weinstock. *Principles for Evaluating the Quality Attributes of a Software Architecture*. Technical Report, CMU/SEI-96-TR-036, ESC-TR-96-136, 1997.

5. M. Barbacci, M. Klein, T. Longstaff, C. Weinstock. *Quality attributes*. CMU/SEI-95-TR-021, ESC-TR-95-021, 1995.

6. M. Barbacci, S. Carriere, P. Feiler, R. Kazman, M. Klein, H. Lipson, T. Longstaff, C. Weinstock. *Steps in an Architecture Tradeoff Analysis Method: Quality Attribute Models and Analysis*. Technical Report, CMU/SEI-97-TR-029 ESC-TR-97-029, 1998.

7. V. R. Basili, H. D. Rombach. *Goal/Question/Metric Paradigm: The TAME Project: Towards Improvement-Oriented Software Environments*. IEEE Transactions on Software Engineering, 1988, Vol. 14, No. 6, pp. 758–774.

8. L. Bass, P. Clements, R. Kazman. *Software Architecture in Practice*. Reading, Massachusetts: Addison-Wesley, 1998.

9. PO. Bengtsson, J. Bosch. *Scenario-based Architecture Reengineering*. Proceedings of the 5th International Conference on Software Reuse (ICSR5), 1998. Pp. 308–317.

10. PO. Bengtsson. *Towards Maintainability Metrics on Software Architecture: An Adaptation of Object Oriented Metrics*. First Nordic Workshop on Software Architecture (NOSA'98), Ronneby, August 20–21, 1998.

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

12. S. A. Bohner, R. S. Arnold. *Software Change Impact Analysis*. Los Alamitos, CA: IEEE Computer Society, 1996.

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

14. J. Bosch. *Evolution and Composition of Reusable Assets in Product line Architectures: A Case Study*. Proceedings of the First Working IFIP Conference on Software Architecture, February 1999. http://www.ipd.hk-r.se/bosch/.

15. J. Bosch. *Design and Use of Software Architectures - Adopting and Evolving a Product line Approach*, Addison Wesley, 2000. ISBN 0-201-67494-7.

16. S. Bot, C.-H. Lung, M. Farrell. *A stakeholder – Centric Software Architecture Analysis Approach*. Proceedings of the International Software Architecture Workshop (ISAW 2), 1996. Pp.152–154.

17. L. Bratthall, P. Runeson. *A Taxonomy of Orthogonal Properties of Software Architecture*. Proceedings of the Second Nordic Software Architecture Workshop (NOSA'99), 1999. 11 p.

18. L. C. Briand, S. Morasca, V. Basili. *Measuring and Assessing Maintainability at the End of High Level Design*. University of Maryland, Technical Report CS-TR-3105, UMAICS-TR-93-65. July 1993.

19. S. Bryan, Doerr and David Sharp. *Freeing Product Line Architectures from Execution Dependencies*. Software Product Lines – Experience and Research Directions, Proceedings of the First Software Product Lines Conference, Kluwer Academic Publishers, 2000. Pp. 313-331.

20. F. Buschmann, R. Meunier, P. Sommerland, M. Stal. *Pattern-oriented Software Architectures, a System of Patterns.* Chichester: Wiley & Sons, 1996.

21. P. Clements, L. Northop. *A framework for software product line practice*, version 2.0, Technical Report. Software Engineering Institute, July 1999.

22. J. Coplien, D. Hoffman, D. Weiss. *Commonality and Variability in Software Engineering*. In: IEEE Software (November/December 1998), pp. 37–45.

23. R. Day. *Quality Function Deployment. Linking a Company with Its Customers*. Milwaukee, Wisconsin: ASQC Quality Press, 1993.

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

25. E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison-Wesley, 1995.

26. G. Gannod, R. Lutz. *An Approach to Architectural Analysis of Product Lines*. ICSE2000, 22nd International Conference on Software Engineering, to appear, Dublin, Ireland, 2000.

27. C. Hofmeister, R. Nord, D. Soni. *Applied software architecture*. Addison-Wesley, 1999.

28. Iannino, A. *Software Reliability Theory*. Encyclopedia of Software Engineering, Volume 2, Marciniak, J.J. (ed.). New York: Wiley & Sons, 1994. Pp. 1237–1253.

29. IEEE Standard *Glossary of Software engineering Terminology*. IEEE Std. 610.12-1990.

30. IEEE Standard 1061-1992. *Standard for software quality metrics methodology*. New York: Institute of Electrical and Electronics Engineers, 1992.

31. ISO/IEC91 – International Organization of Standardisation and International Electrotechnical Commission. *Information Technology – Software Product Evaluation – Quality Characteristics and Guidelines for Their Use*. ISO/IEC 9216: 1991(E).

32. E. Karlsson (ed.). *Software Reuse, a Holistic Approach*. Chichester: Wiley & Sons, 1995.

33. R. Kazman, M. Barbacci, M. Klein, S. J. Carriere, S. G. Woods. *Experience with Performing Architecture Tradeoff Analysis*. Proceedings of ICSE99, Los Angeles, CA, May 1999. Pp. 54–63.

34. R. Kazman, G. Abowd, L. Bass, P. Clements. *Scenario-based Analysis of Software Architecture*. IEEE Software, November 1996. Pp. 47–55.

35. R. Kazman, G. Abowd, L. Bass, M. Webb. *Analyzing the Properties of User Interface Software Architectures*. Carnegie Mellon University, School of Computer Science, Technical Report, CMU-CS-93-201, 1993.

36. R. Kazman, L. Bass, G. Abowd, M. Webb. *SAAM: A method for analyzing the properties of Software Architectures*. Proceedings of the 16th International Conference on Software Engineering, 1994. Pp. 81–90.

37. R. Kazman, M. Klein, M. Barbacci, H. Lipson, T. Longstaff, S. J. Carrière. *The Architecture Tradeoff Analysis Method*. Proceedings of ICECCS, Monterey, CA, August 1998. Pp. 68–78.

38. R. Kazman, S. J. Carriere, S. G. Woods. *Toward a Discipline of Scenario-Based Architectural Engineering*. Annals of Software Engineering, Vol. 9. To appear, 2000, http://www.cgl.uwaterloo.ca/~rnkazman/SE-papers.html.

39. M. Klein, T. Ralya, B. Pollak, R. Obenza, M. Gonzales Harbour. *A Practitioner's Handbook for Real-Time Analysis*. Boston: Kluwer Academic Publishers, 1993.

40. M. Klein, R. Kazman, L. Bass, S. J. Carriere, M. Barbacci, H. Lipson. *Attribute-Based Architectural Styles*. Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1), San Antonio, Texas, February 1999. Pp. 225–243.

41. T. Kishi, N. Noda. *Aspect-Oriented Analysis for Product Line Architecture*. In: Software Product Lines – Experience and Research Directions, Proceedings of the First Software Product Lines Conference, Kluwer Academic Publishers, 2000. Pp. 135–147.

42. P. B. Krutchen. *The 4+1 View Model of Architecture*. IEEE Software, November 1995, pp. 42–50.

43. T. Ihme. *A ROOM Framework for the Spectrometer Controller Product Line*. In: Proceedings of Workshop on Object Technology for Product Line Architecture. Pp. 119–128. ESI-199-TR-034.

44. P. Lalanda, J. Bosch, R. Lerchundi, S. Cherki. *Object Technology for Product- Line Architectures*, ECOOP'99 Workshops, LNCS 1743, Berlin Heidelberg: Spinger-Verlag. 1999. Pp. 193–206.

45. N. Lassing, D. Rijsenbrij, H. van Vliet. *Flexibility in ComBAD architecture*. Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1), San Antonio, Texas, February 1999. 15 p.

46. N. Lassing, D. Rijsenbrij, H. van Vliet. *On software architecture analysis of flexibility, Complexity of changes: Size isn't everything*. Proceedings of the second Nordic Software Architecture Workshop (NOSA'99), 1999. 6 p. ISSN 1103-1581.

47. N. Lassing, D. Rijsenbrij, H. van Vliet. *The goal of software architecture analysis: confidence building or risk assessment*. Proceedings of the First Benelux Conference on State-of-the-art of ICT architecture, 1999. 6 p.

48. N. Lassing, D. Rijsenbrij, H. van Vliet. *Towards a Broader View on Software Architecture Analysis of Flexibility*. Proceedings of the Asian-Pacific Software Engineering Conference (APSEC'99), 1999. Pp. 238–245.

49. W. Li, S. Henry. *Object-Oriented Metrics that Predict Maintainability*. Journal of Systems and Software, 1993. Vol. 23, No. 2, pp. 111–122.

50. J. W. S. Liu, R. Ha. *Efficient Methods of Validating Timing Constraints*. Advances in Real-Time Systems, S. H. Son (ed.). Englewood Cliffs, NJ: Prentice Hall, 1995. Pp. 199–223.

51. L. Lundberg, J. Bosch, D. Häggander, P.O. Bengtsson. *Quality Attributes In Software Architecture Design*. Proceedings of the IASTED 3rd International Conference on Software Engineering and Applications, October 1999. Pp. 353–362.

52. C. Lung, S. Bot, K. Kalaichelvan, R. Kazman. *An Approach to Software Architecture Analysis for Evolution and Reusability*. Proceedings of CASCON '97, November 1997. 11 p.

53. J. A. McCall. *Quality Factors.* Encyclopedia of Software Engineering. Vol. 2, J.J. Marciniak (ed.) New York: Wiley & Sons, 1994. Pp. 958–971.

54. G. Molter. *Integrating SAAM in Domain-centric and Reuse-based Development Processes*. Proceedings of the Second Nordic Workshop on Software Architecture (NOSA'99), 1999. 10 p. ISSN 1103-1581.

55. D. Parnas, *On the criteria to be used in decomposing systems into modules*. Communications of the ACM, 1972. Vol. 15, No. 12, pp. 1053–1058.

56. D. Perry, A. Wolf. *Foundation for the Study of Software Architecture*. SIGSOFT Software Engineering Notes, 1992. Vol. 17, No. 4, pp. 40–52.

57. J.S. Poulin. *Measuring Software Reusability*. Proceedings of the Third Conference of Software Reuse, Rio de Janeiro, Brazil, November, 1994. Pp. 126–138.

58. B. M. Reed, D.A. Jacobs. *Quality Function Deployment for Large Space Systems*. National Aeronautics and Space Administration, 1993.

59. P. Runeson, C. Wohlin. *Statistical Usage Testing for Software Reliability Control*. Informatica, 1995. Vol. 19, No. 2, pp. 195–207.

60. M. Shaw, D. Garlan. *Software architecture. Perspectives on an Emerging Discipline*. Upper Saddle River, NJ: Prentice Hall, 1996.

61. C. Smith. *Performance Engineering of Software Systems*. Reading, Massachusetts: Addison-Wesley, 1990.

62. C. Smith, L. Williams. *Software performance engineering: A case study including performance comparison with design alternatives*. IEEE Transactions on Software Engineering, 1993. Vol. 19, No. 7, pp. 720–741.

63. W. P. Stevens, G.J. Myers, L.L. Constantine. *Structured Design*. IBM Systems Journal, 1974. Vol. 13, No. 2, pp. 115–139.

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 427
VTT-PUBS-427

Author(s)
Dobrica, Liliana & Niemelä, Eila

Title

A strategy for analysing product line software architectures

Abstract

The purpose of the architecture evaluation of a software system is to analyze the architecture to identify potential risks and to verify that the quality requirements have been addressed in the design. This research report addresses the issue of how to perform an analysis of software product line architectures. Throughout the chapters we try to present a way of thinking founded on analysis at the architecture level of the quality attributes, with the purpose to initiate and maintain a software product line considering the quality as the main driver in the product line development. The analysis strategy is exemplified with a spectrometer controller product line, a case study where the product line is initiated in a revolutionary style, such that the product line architecture and its components are elaborated to match the requirements of all the expected product line members.

In this report, we present our original contributions in developing this significant and, in the same time, new research domain. In order to be able to discuss an analysis strategy for a product line architecture, it is a considerable advantage to have a good knowledge of the state of art and practice in the software architecture domain. One of our contributions is to extract the main concepts that are common to software architectures and to present what is specific for the product line approach in software development. The study in the first part of the report gathers together, for the first time to our knowledge, all of the important published software architecture analysis methods and attempts to compare them. This survey shows the state of the research at this moment, in this domain, by presenting and discussing eight of the most representative architecture analysis methods. The study represents a step towards defining a general product line architecture analysis strategy. In practice, we simplify our analysis approach and the last part of this report focuses on our experiences with the product line architecture analysis of the spectrometer controller PL.

Keywords
software product line, analysis techniques and methods, scenarios, quality attributes, spectrometer controller

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

ISBN			Project number
951–38–5598–8 (soft back ed.) 951–38–5599–6 (URL: http://www.inf.vtt.fi/pdf/)			E0SU00132
Date	Language	Pages	Price
December 2000	English	124 p.	C
Name of project		Commissioned by	
EMU-PLA			
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	