Development and Use of Dynamic Product-line Architectures

Jesper Andersson
Växjö universitet
Jesper.Andersson@msi.vxu.se

Jan Bosch
University of Groningen
J.Bosch@cs.rug.nl

Abstract

Software product families are used to shorten time-to-market, improve quality, and lower cost, by means of effective reuse. This paper presents the results of case study conducted at four Swedish companies that are involved in either the development of or development with a software product family. We identify and discuss several issues such as inter-organisational development of platforms, platforms that employ dynamic reconfiguration, and platforms as a vehicle to achieve certain quality attributes. We analyse issues and demonstrate how these can be deduced to shortcomings in scoping and variability management for non-functional quality attributes and dynamic architectures.

1 Introduction

One of the most prominent trends present in software engineering practice is the delaying of design decisions. Whereas software architects and engineers earlier decided during the specification phase what exact product to build, currently the ambition is to delay the decision concerning the exact product to build to the latest possible point. The rationale for this is that design decisions, once taken, are very difficult to revoke. Consequently, when the requirements change during the development process or when the developed software is considered for use in a different context, it is, sometimes prohibitively, expensive to replace one design decision with another. The delayed design decisions typically are concerned with features that should be present in some configurations, but not in others. A feature may consist both of functional and quality requirements. Several types of quality requirements exist. Some are relevant from a user perspective, e.g. dependability, whereas others are relevant for the development organisation, e.g. maintainability. Several engineers and researchers have identified the connection between the software architecture and the qualities a system exhibits. Software Architecture (SA) is defined as "the fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding

its design and evolution" [1]. Architecture as a facilitator for software quality is described in several publications [2, 3].

Software reuse, cuts costs and improves overall quality, hence software development organisations search for more advanced techniques improving their day-to-day operations. Examples of such techniques are libraries, frameworks, and platforms, all widely used vehicles for increased reuse of functionality (and quality). With libraries, reuse is calling library methods. Reuse in the framework context, is about specialisation, where behaviour and structure is inherited and specialised in application classes. A platform (or middle-ware) is a combination of a framework, a library, and a run-time system. Platforms provide functionality and run-time support that assist developers in developing applications in certain functional and/or quality domains. The platform provides methods in the library style for access and control. In order to access them, components have to conform to certain properties. This is achieved by inheriting properties from abstract classes. Like in frameworks, a CORBA platform requires that client and server components conform to some abstract CORBA component.

A more recent development within the software engineering community is the increasing use of Software Product Families (SPF). A SPF consists of a product family architecture (unification of individual product architectures), a set of shared software components and set of products that have been derived from the shared assets. The main advantage of a software product family is that it allows for pervasive intra-organisational reuse by exploiting the commonalities between a set of products, while allowing for variability. The notion of variability and variation points is discussed by, among others, van Gurp et.al. [4]. A variation point is a location in a software asset for which multiple variants and/or parametric configurations exist. A variation point is introduced when a design decision has been delayed. The main aim of software product families is to delay design decisions concerning the precise requirements (both functional and quality) until product derivation and, consequently, the compilation and linking phase. However, in several systems this is not sufficient and several of the variation points need to be kept unbound until run-time. Several products provide provisioning for run-time configurability, were end-users control the feature set available in a product. The notion of dynamic software architectures [5], i.e. software architectures with the ability to change configuration at run-time, has recently started to attract increasing amounts of attention. The extended capabilities provide developers with proper means for implementing different functional as well as quality requirements.

One can identify several examples of the trend to delay design decisions and improve reusability, including the increasing configurability of software products and the widespread use of software product families. The problems and issues concerning the industrial use of dynamic software architectures, especially in the context of software product families and platforms, has received relatively little attention in the software engineering research community.

To address this, we present in this paper the results of a case study involving four Swedish software development organisations, i.e. MECEL Automotive Sys-

tems, Ericsson Radio Systems (ERA), Ericsson Mobile Data Design (ERV), and ENEA-OSE (we refer to these organisations by the enterprise name, although the study was conducted at groups within these organisations). The objective for the case study was to survey the state of practice, identify issues and problems, and, based on this, deduce new relevant research questions for the SA community. The case study is based on interviews and document studies. The selection of organisations for this study spans from, low-level, operating system functionality providers, via developers of middle-ware and traditional system developers, to system engineers that integrate pre-developed subsystems. A second dimension is whether the organisation is a provider or a user of a platform. ENEA and ERV develop platforms while ERA and MECEL represent "users" of such platforms.

Interestingly, the primary reason for selecting a platform for product development has been to support the implementation of specific quality attributes, e.g. high-availability, performance and fault-tolerance, rather than to achieve reuse of functionality or decreased time-to-market. The company's argument for employing dynamic architecture technology is that requirements like availability and flexibility require a dynamic, configurable, solution. The back side of this approach, i.e. it's the complex technical structure and consequently expensive implementation, are accepted and dealt with accordingly. Managing non-functional requirements in a software engineering project is difficult. By employing platforms the companies believe that some of the risks connected to this can be mitigated.

The contribution of this paper is twofold. The investigation has revealed new and interesting issues and research questions concerned with inter-organisational product family architectures, problems related to scoping and variability management for quality attributes, and dynamic product family architectures. In addition, we present an in-depth analysis of issues connected to commonality, variability, and scoping of non-functional requirements.

The remainder of this paper is organised as follows. First we introduce related work in Section 2. In Section 3, we define the field of SPFs and the relation between SA and quality attributes. Section 4 and 5 present the case study method and organisations. In Section 6, we present the identified problems and issues in-depth. Section 7 contains a general discussion and presents a research agenda based on the findings in the case study. Finally, in Section 8 we conclude and discuss future work.

2 Related Work

The ideas of designing "catalogues" of standard functionality, selecting instances, and including these in a product are not new, as are the problems and issues connected to it. A historical retrospect reveals that all major contributions in the area of software libraries have recognised variation and the issue of achieving quality.

In McIlroys seminal paper [6], families of routines and the number of variants of individual routines that a standard catalogue of routines should offer are

discussed. Continuing, McIlroy also discusses what will happen if quality and context dimensions, such as robustness and precision, are included. McIlroy conclude that if these dimensions are included, the variability will grow exponentially with respect to the number of possible implementations. McIlroy also identifies the need for trade-offs, and to include combinations of tradeoffs among the candidate variants. As an example of a trade-off, he uses the classic time-space trade-off for algorithms.

Parnas introduces the concept of program families [7] in the software module context. He argues that if a set of similar products is to be designed and implemented one should consider them as a whole, a program family. Further on he discusses the concept of variation and how important it is to keep the family in focus and not the individual products.

Frameworks [8] are sets of interacting and cooperating classes that users reuse via specialisation or direct instantiation in an application. Frameworks provide an abstract design for a set of related problems. The applications must not necessarily be in the same domain, but at some conceptual level, the basic requirements should be similar. Usually frameworks are divided into two classes, vertical frameworks that provide domain specific support, and horizontal frameworks, providing more general application support for use in many different domains.

Some systems described in this study are examples of middle-ware software systems. More formally, a middle-ware software is a set of purpose specific services, usually packaged as a platform, that "glue" and coordinate the behaviour of two or more components. The most recognised middle-ware software packages provide services for transparent distributed computing, examples here are CORBA and .NET. Beside service functionality, several middle-ware platforms include general support for some (set of) quality attributes.

The related work in the area of SPFs will be covered in Section 3 below. Most of the discussed publications are dealing with problems connected to family functionality and activities pre-execution which contrast to our study.

Managing quality attributes (or non-functional requirements) in software engineering is regarded as difficult to address, mainly because the nature of these have been insufficiently explored. Recently, more pragmatic, still formal, approaches to effective management of quality attributes in software development have been proposed [9].

3 Software Product Families

Software enterprises are all involved in the never-ending activities of cutting development costs as well as improving product quality. One of the more promising approaches (at least according to the number of published success stories), which both cuts costs and improves quality, is domain-specific, intra-organisational, reuse-based development. Built on top of the foundation of software architectures, software product families focus on improving reuse and, thus cut costs, shorten lead-

times, and improve quality. A software development organisation normally specialises in one or a limited number of areas. It produces a number of individual software products within its domain of expertise. The related products, typically, have common structure and behaviour. A product family architecture is the unification of all system architectures in a SPF. The product family architecture captures the architectural commonalities and variabilities [10] in a product family and comprises a map that guides developers when new application instances are derived. The selection of specific component implementations is guided by configuration information made explicit in the family architecture. The product family is used when products are created (i.e. instantiated or derived). Instantiation of a new product typically involves selection of components from the product family architecture and addition of product specific components. This procedure guarantees that the amount of superfluous functionality in the product is minimised (n.b. not minimal). The product specific components can later be evaluated. If they are common to other products as well they are included in the product family architecture. Figure 1 depicts how products are generated, a common product family architecture is derived from product specific architectures and component instances of the family and product-specific components are integrated.

To fully understand a product-family, we need tools to describe the common parts and the differences between two or more products. A naive approach is to talk about which components or classes are used in a specific product and which are not. This approach is too low level and not feasible in practice (i.e. for large families or large products). For instance, the behaviour(role) of a component may vary from product to product so two products with the same set of components may demonstrate different behaviour. Approaching this problem on the level of components discards the important fact that behaviour is also connected to component interactions. Instead, the concept of features, is much more promising. A feature, we define as "a logical unit of behaviour that is specified by a set of functional and quality requirements" [2, p. 194]. A feature is realized by sub-architectures with a (limited) set of interacting components. Components and features have an $n \times m$ relation, i.e. multiple features may depend on the same component. This will impact the traceability and complicate tracing from features to components and vice versa. If something is changed in the shared component, it will possibly affect two or more features. There are also other, non-explicit, feature dependencies.

Several of the non-functional requirements are not purely local, i.e. their behaviours are affected by and have effect on other features in the system. Coplien uses the term "Good domains" [10] for domains that can be managed easily. The other class of domains contains domains that interfere with other domains (crosscuts). These domains are much more difficult to manage. This observation holds for features, requirements, and components. Consequently, a "good" feature would correspond to a feature that has no explicit or non-explicit shares with other features. This illustrates the importance of decomposition and how the decomposition made will prescribe if a feature is referred to as "good" or "bad". The ultimate goal is of course to find ways to engineer "good features" for all domains and develop

systems by composition of "good features" only. Returning to the notion of "good features", we see that for functional behaviour there are only explicit connections between features. A typical "functional connection" between two features is a component dependency that is explicitly declared. This kind of connections can be traced and, hence effectively managed [11]. Explicit connections make it possible to trace inter and intra "feature cause-effect"-relations. Hence, change management for connected features is simplified. For features that predominantly define required qualities, the situation is different. Several qualities are impossible to realize locally and are inherently cross-cutting. This means that even if the feature, from a functional point of view, is a "good feature", cross-cutting qualities will violate the definition of "good", thus affecting the composability negatively.

One possible path around such problems would be to "factor out" qualities into independent and orthogonal aspects specifications. Previous work, such as Composition Filters [12], Subject Oriented Programming [13], Aspect Oriented Programming (AOP) [14] and the layered object model [15], take the first steps towards robust techniques for factorisation of cross-cutting features. Still, a lot of work remains before general techniques that manage design and implementation of quality aspects are standard tooling for software engineers.

Selecting features during application development begins as soon as the requirements are understood (in fact sometimes even before that). Some features are selected and "frozen" early in development. Examples of this include low-level features like operating systems. On the application level, features that are used in the early phases of the process are abstract (feature skeletons) in the sense that they capture several designs. The feature-skeletons are far from complete and several design decisions remain. As decisions are made, the features are refined, and become more concrete.

Several authors discuss the connection between a configuration of components in an architecture and system wide properties that the system exhibits [2, 3, 16]. As previously mentioned, the difference between emerging, system wide properties, and functionality is that functionality can be located in a single component within the architecture (or a sub-architecture encompassing a group of components), the system wide properties however, are scattered over the system in several components, i.e. cross-cutting. Even if we create an architecture with a configuration that fulfils many of the quality requirements, such as maintainability, there are still many requirements in this class that need support at run-time. Filman states that several of quality attributes or "ilities" can be achieved by "systematically controlling the inter-component communications" [17]. Szyperski and Vernik make the same observation, and conclude that, system wide properties in component systems "require dedicated support outside of the participating components" [18].

The software architecture community has proposed a number of different "tools" to achieve system quality. Architectural design methods [2, 3], focus on the structure of the software (product family) architecture and provide toolboxes for criterion-based evaluations. The idea is to design architectures that, while providing the expected functionality, at the same time organise the structure in a way that certain

system-wide properties are achieved or can be achieved later in the development process. Another approach is to design in capabilities for performing dynamic reconfigurations in the architecture [19, 20]. In combination with monitoring, these techniques initiate "state-triggered" reconfigurations. For instance, if the system load is too high, this triggers a load-balancing component that reconfigures the system in a way that a system load criterion still holds.

Even though the first "tottery" steps towards systematic approaches for system quality realization have been taken, there are still many (undiscovered) problems that remain to be solved. If we scrutinise works in the area of software product families the majority of problems and issues we find are concerned with commonality and variability management, and scoping product families [4, 21]. The research community has also reported on how these can be better controlled and managed [22, 23].

Scoping

> The development team of a software product family initially tries to grasp the "market" of their product family. By understanding the context in which their products will run and additional customer needs, the group defines the domain specific functional and quality boundaries. The process of deciding what to include and what to exclude is referred to as scoping. When commonalities are found and isolated, and variabilities defined, the scope is set. Scoping is not easy and several parameters must be considered. For instance, software product-families evolve over time so future scope extensions must be accounted for. Diversity in the user group can also cause problems. Typical problems recognised in requirements engineering reoccur here. To cope with these problems different methods can be applied. Examination of existing products in a domain, product/feature matrices, and stakeholder workshops are all usable vehicles towards the definition of a SPFs scope. A more systematic approach to the problem is captured in the PuLSE-Eco method [24].

Commonality and Variability

> One of the more widely used mechanisms to grasp huge complex problems is abstraction. Abstraction is a process where some "thing" is described by isolated aspects important in a specific context. Thus, the basis for an abstraction is a set of common aspects valid in the domain currently being abstracted. From a SPF perspective, commonality analysis aims at pinpointing common features for a family of products. Once the commonalities are determined, one can start to investigate the differences, i.e. what makes a specific product unique. The variability of a family is the union of all these product specific differences. In order to better describe the commonalities and variabilities within the product family architecture, the notion of variation points is useful. This opens up a product family architecture and the result is a more flexible and modifiable architecture. During product

derivation, the product engineers select variants and bind these to the appropriate variation points. Variation points in the architecture evolve through three phases, i.e. implicit, designed and bound. Before a variation point is designed into the system, it is considered to be implicit. When support for this implicit variation is designed into the system the variation point becomes explicit. The explicit variation point is bound, when a specific variant is chosen. Another dimension for variation points is open and closed, i.e. modifiable or not. If the time when the variations are bound can be delayed, and if closed variation points can be reopened, the positive result will be flexible development were unforeseen events, such as requirement changes, are more easily handled. Of course such a scheme can introduce other issues such as coordination related problems when un-tested subsystems are introduced at run-time. This and connected problems can be mitigated if only components with certified quality are used. Examples of this include the Microsoft driver signing and non-tainted (open source) modules in Linux.

Our case-study organisations primarily work with systems where the notion of variation point and binding has moved beyond load-time into execution time. The controlling "intelligence", i.e. the mechanisms that govern these activities in the system, is also improved, thus providing for more complex and generic variations.

4 Case Study Method

The objective of the case study reported in this paper was twofold.

- Survey the state of practice within different software development organisations, with an emphasis on the use of dynamic reconfiguration and updating techniques as a vehicle for achieving flexible quality requirements.
- Identify issues and problems described by the organisations and based on those, deduce industrially relevant research questions.

We concluded that the best method to achieve this was to combine interviews with key persons in different software development organisations and document studies. The case study involves four Swedish software development organisations; MECEL Automotive Systems, Ericsson Radio Systems (ERA), Ericsson Mobile Data Design, and ENEA-OSE. MECEL Automotive Systems and ERA are "users" that use (or would like to use) platforms and product family architectures for improved reuse, ERV and ENEA-OSE are platform providers. At the time of the study there was no cooperation between the organisations in this area. The interviews were open, but a set of prepared questions was used to drive the discussions forward. The questionnaire we used was divided in four categories, i.e. context, current use, future use, and vision. The first category aims at collecting general information, such as type of organisation, products, and the organisations general

view on dynamic updating and reconfiguration. For the categories “current” and “future use”, we discussed issues connected to different phases in a generic development process; requirements, analysis and design, implementation, verification & validation, and maintenance. The interviews were taped and later transcribed. These transcripts were further analysed. The interviewees provided us with additional documentation, which was used as a second source of information for the analysis.

5 Case Study Organisations

Below, we present the individual organisations, the primary non-functional requirements they face, the technologies they use, and future challenges.

5.1 Systems engineers - MECEL

MECEL Automotive Systems is an engineering consultancy organisation, mainly contracting the automotive industry. They have been involved in numerous projects as system engineers, where they have worked with integration of software and hardware systems.

Currently, high-availability and fault-tolerance are key qualities in automotive systems, but recent developments towards “infotainment”-systems being integrated into vehicles, will require additional support, for instance, flexibility and configurability.

In a development project, MECEL employs their AUTOSOFT method. The method uses a traditional requirement specification, in the form of a description of all vehicle functions implemented in software and/or hardware. Beside the functional and non-functional requirements captured in the vehicle function description, AUTOSOFT includes a Life Cycle Cost (LCC) analysis. The LCC support is crucial when the clients have a focus on overall cost reduction since reduced costs can be used to increase the margins or lower the price paid by the customer for the final product. The functional descriptions form the basis for decomposition where functions are decomposed into sub-functions and a physical decomposition where sub-systems are identified. The next step is architectural design, which includes design of software and hardware architectures. This step includes several activities such as hardware configuration design, identification of sub-function commonalities, allocation of sub-functions to sub-systems, and descriptions of communication between sub-functions. The architectural descriptions are later used in the implementation and deployment phases of the process. AUTOSOFT is an industrial strength development process, but MECEL foresees several adaptations that will be needed when the automotive industry moves towards more functionality in software. Currently, the process depends on hardware implementations for specific qualities; fault-tolerance for instance is handled by means of hardware redundancy.

Even though the automotive industry of today is somewhat conservative, the increased competitiveness will affect how automotive software is developed. Commonplace practice employ shared components (mechanical and digital) between different manufacturers model programs but software is not shared although it is not unique. MECEL believes that this situation will change sooner rather than later. MECEL aims to move towards a more reuse-oriented process. Software components should be reused, across model programs and manufacturers, in the same manner as mechanical components. The platform should support "plug n' play" for both functional and non-functional properties. This will require, besides improved support for managing the system qualities, extensive work where the automotive industry and sub-contractors' agree on specific standards for automotive software components.

5.2 Application development - Ericsson Radio

ERA develops software for GSM (Global System for Mobile communications) based networks. The group we interviewed was relatively recently formed, and mainly used Java technology.

In their application domain, high-availability, including fault-tolerance, scalability and performance are key qualities. The different quality characteristics can change at run-time, as a reflection of the scalability requirement. High-availability is supported by dynamic updating. As previously mentioned, another important property is scalability. If additional hardware is introduced to increase the system capability, the system should adapt and new processes could be spawned and executed on the new hardware.

Scalability, high-availability, and performance are all managed in a similar manner using the same technique. The technique uses two or more parallel class-loaders in the virtual machine that executes the application. Each class-loader is responsible for one configuration of the system. When a new configuration is needed the responsibility is gradually transferred to a new class-loader instance, which creates its own class instances. Instances states' are transferred from the active application to the new instance over the serialisation interface provided by the Java platform. Serialisation provides output and input streams for object information. Obviously the serialised information is not always directly interchangeable between different instances, for instance new and removed attributes must be managed by special code. Finally, when all objects are migrated from the active system instance the new system instance can be activated.

Currently, the core functionality supporting qualities are implemented in the application, with limited support from the platform. This reduces the possibility to reuse application components. Albeit ERA aims to move their development towards parameterised, reuse oriented development, with more support for quality realizations in a platform.

5.3 Platform developers - ERV

Ericsson Mobile Data Design is involved in the development of systems for packet based mobile communication. One of their recent products is the GSN (GPRS Support Node), an important component in the General Packet Radio Service (GPRS) communication systems. ERV is also responsible for the development of the Wireless Packet Platform (WPP), a common platform for the next generation mobile communication systems, for instance the aforementioned GPRS and the next generation, the Universal Mobile Telecommunications System (UMTS).

Key qualities in the ERV platforms are flexible software delivery and installation, load balancing, fault-tolerance by redundancy, and dynamic upgrading.

One important component in the WPP is the Distributed Processing Environment (DPE). The DPE architecture, depicted in Figure 2 and described here, supports heterogeneous distributed computing on multiple hardware platforms running different operating systems. In Figure 2, the boxes represents a networked computing device, the gray circles depict the operating systems executing on each node. One of the computing devices is appointed as “Node Control Board” (NCB). On the NCB most of the DPE logic (NCL) is executed. One node is appointed as the standby NCB (in Figure 2 depicted by the white triangle). The state of the active NCL is replicated to the standby NCL. The NCL and DPE agents constitute a reflective and reactive architecture. These components are capable of and responsible for monitoring certain events and based on application and system specific data performing dynamic reconfigurations. Application developers access the DPE via an API that currently consists of approximately 100 functions divided into three categories; System events, Application distribution, and Inter/Intra application signalling.

The challenges that ERV face are concerned with an improved understanding of how to design and evolve platforms that support qualities. A better understanding will support future development and maintenance scenarios.

5.4 Operating system developer - ENEA/OSE

ENEA develops OSE; a message based real-time operating system and additional support systems for design, test, and administration. The operating system is used in different domains, e.g. “telecom” applications and automotive industry.

Several applications in these domains have quality requirements like; fault-tolerance, high-availability, and performance. In order to support development of systems with these kinds of qualities, ENEA/OSE offers certain services at the system level that the application programmer may utilise.

OSE is a SPF where each customer selects a set of services that they require from the system. A conservative estimate of the common parts (commonalities) in all products is somewhere between 50-90% depending on which hardware architecture the customer use and features they need. The most important architectural mechanism is the logical channels for component communication. A logical chan-

nel, depicted in Figure 3, is a connector that offers transparent communication for processes. The mechanism is implemented by means of three different components; the Phantom process, the Link handler, and the Hunt process (n.b. not displayed in Figure 3).

The logical channel mechanism can be used to implement qualities, such as performance (load-balancing), fault-tolerance, and high-availability (dynamic updating). For instance, if new and modified components are available these can be initiated in parallel with the old, active components. Data can be migrated and when the components are in stable states the old components can be deactivated. Components that would like to communicate with the components "hunt" for a component and the hunt daemon will redirect the communication to the new components. The same strategy can be used to implement other quality attributes, such as process-migration used in load balancing.

A better understanding of the nature of quality attributes is in the interest of ENEA. Such an understanding will improve their design and development capabilities and strengthen their position for future system development challenges.

6 Problems and Issues

This section is divided into two subsections. First, we report on our initial findings based on a primary analysis of the material. There we give a more precise characterisation and classification of our case-study organisations SPFs'. Second, we identify, describe, and exemplify the identified problems and issues connected to development and use of product-families and platforms. We also attempt to identify and discuss the underlying causes for a particular problem or issue.

6.1 Overview

In this section, we present an overview of the characteristics of the cases under study and issues that we identified as relevant for a broader audience than the individual cases. As discussed in the related work section, to the best of our knowledge these issues have received little attention in research literature. Among others, we found several interesting problems mainly concerned with variability management and scoping.

6.1.1 Characteristics

Inter-organisational SPFs

> The case-study organisations develop SPFs for other organisations or are users of such SPFs. We label these SPFs inter-organisational. MECEL describes a situation where organisations that provide hardware components also provide accompanying software. This software is later integrated into

the "automotive system" in a vehicle. Hardware is usually available in different variants, hence is the software for the hardware components. By definition this software can be developed as a SPF and we arrive at a situation where the platform developer is outside of the user organisation. ENEA, with their OSE operating system, and ERV with DPE, also develop platforms that are targeted for use by other organisations. Where existing literature typically reports on intra-organisational SPFs in large organisations, we have identified that SPFs provide the same advantages when crossing inter-organisational boundaries. However, as discussed in the next section, there are a number of unique challenges that need to be addressed.

SPFs for Qualities in Embedded Systems

The market for ENEA, MECEL, ERV, and (to some extent) ERA is embedded systems. The market characteristics impose certain specific quality requirements on their SPFs, such as multi-tolerance [25] and real-time characteristics. Distinctive for our case study SPFs are that quality requirements are the most prominent and important requirements. For instance, the DPE from ERV have a focus on support for scalable fault-tolerant distributed computing. OSE from ENEA provides a solid foundation for scalable fault-tolerant real-time systems. The underlying idea is that other organisations will use these SPFs and build functionality (maybe as another SPF) on top. For instance, the DPE will be used by other Ericsson organisations that implements application specific functionality. Thus, where most product families reported upon in literature aim at reusing functionality, the SPFs in our case study organisations are predominantly used to provide quality attribute support.

SPFs and Dynamism

Finally, we note the usage of dynamic reconfiguration to implement quality requirements such as scalability, performance, and availability. This is demonstrated by the system descriptions provided in Section 5. But the packaging of dynamic architecture aspects in SPFs and the implications for development and use of SPFs to which this lead are, we believe, of great interest. In our experience with these and other cases, we have seen that with increasing experience with and maturity of software product family approaches, there is a strong tendency to bind variation points at later stages in the life cycle, ultimately at run-time.

6.1.2 Issues

Below we introduce and briefly present the identified issues. Each issue will be described in more detail in Section 6.2.

Evolving Quality Requirements

This was presented as a key problem by the platform vendors ENEA and ERV. It occurs when the scope of a quality-platform is extended as part of platform evolution. The cross-cutting nature of qualities makes it extremely difficult to introduce support for additional qualities with preserved service abstraction levels. A platform supports a set of qualities and provides platform users with high-level services that simplify integration of services in user applications. An expansion where support for additional qualities are included requires major revisions to the platform design or "lowering" the support level, i.e. leave the integration (coordination) aspect of quality implementation to the platform user. The core problem is how to expand a quality platform with preserved support levels for current platform users.

Service Level for Quality Requirements Support

A second problem, also caused by evolution and identified at both ENEA and ERV is the situation where some customers require improved support, i.e. the platform should take on more responsibility from the application in terms of control and integration. Here, platform developers have to design in more control logic into the platform. However, this leads to problems when other applications that use functionality on a lower service abstraction level (as provided by previous platform versions). The organisations both stressed that this is something of a Catch-22. One aspect is to meet the new expectations from customers by providing high-level support and at the same time allow other users to continue working with a low-level interface. The core problem is how to make applications, working on different levels, coexist on a platform without causing behavioural problems with unexpected consequences.

Dynamic Quality Footprint

All case study organisations put quality attributes at the top of the list when we asked why they employed or utilised dynamic architectures. Using the words of ERV: "Attributes like availability, scalability, and modifiability all come hand in hand with a system that can change its configuration". ENEA expressed a similar motivation while the user organisations ERA and MECEL stressed flexibility. All organisations added to this, when they discussed how quality attributes might change at run-time. For instance, changes in the execution environment like hardware changes will affect the quality attributes of an application. They all believed that there were no royal-road solutions available when it came to management of dynamically changing quality attributes. The solution that all discussed was to utilise dynamic architectures making their platforms and systems adaptable. In that way, the platform or system could be tuned so that it meet the changed

quality attribute levels. The problems connected to this is how to manage quality variation points, how to express quality attributes and their interrelationships, and the lack of standardised, reusable descriptions of how to design evolvable architectures.

Quality Support Mismatch

Another issue expressed by both users and providers where the apparent threat of different kinds of mismatches. One can divide the mismatches into intra and inter platform mismatches. Intra platform mismatches represents mismatches between applications, executing in parallel, that utilise support from the same platform. Inter platform mismatches comprise mismatches where two or more platforms, supporting the same or different quality attributes fail to integrate. The first category is a result of shortcomings in variability management and scoping while mismatches in the second category can be deduced from integration problems such as conflicting or missing intra platform coordination (in fact this is a scoping problem). The solution brought forward, and to some extent implemented in some cases, was more flexible and extendible platforms where users configure (add, remove, and modify) quality support in the platforms and provide tailored integration support when more than one platform is used.

Flexibility & Extendibility of Quality Support

This issue was highlighted by all cases as an extremely important development for future quality platforms. The two keywords reappeared in all solution discussions connected to the issues listed above. This is an important development for providers of platforms as well as for platform users. From the provider horizon it is important since it will simplify the process of tailoring platforms for specific customers, hence shorten lead times. It will also broaden the customer base since customers within certain boundaries can modify the platform behaviour adjusting it and adapting it for their particular application domain. Part of the problem here is of course the difficulty of modelling and designing an "open scope" platform that still guarantees certain behaviours in every situation. Part of the solution to problems here can be found in the area of object-oriented frameworks [8], where flexibility and extendibility are key concepts. Still, resolving the complex dependencies among different quality attribute implementations, due to the cross-cutting nature of quality, remains to be effectively addressed.

6.2 Identified Problems and Issues

In this section, we provide a detailed description of the issues that were briefly presented in the overview section above. We use a template based on five different parts for the presentations. First, we assign the problem to a domain, either

Provider or User. Some of the problems presented apply to both the user and provider domain. The second part, the description, gives a generalised view and more company and product specific details have been abstracted away. The description is followed by an example taken from one of the case study organisations. In the fourth part we discuss the primary causes of the problem described. We use a simple classification scheme for causes, since there are multiple cause-effect relations present. This classification groups causes by technical, process, and/or business. The rationale for this classification is to more easily describe the complexity hiding underneath the problem surface. The fifth and final section discusses existing and proposed solutions (when present) to a particular type of problem.

Title: EVOLVING QUALITY REQUIREMENTS

Domain: Provider

Description:

> As a platform developer, a primary goal is to expand the market and increase the number of potential platform users. In order to support a wider range of applications, platform developers need to improve existing support for qualities and add support for new qualities. The problem is that a specific quality can hardly ever be regarded in isolation. Thus, when support for new qualities is added, the platform must take on the responsibility and integrate this with existing quality support in the platform. This often becomes a major problem that requires trade-off decisions in order to be resolved. Consequently, the resulting platform provides a more restricted support for combinations of qualities. In Figure 4, we illustrate this phenomenon graphically. A platform contains support primitives for a set of quality attributes and some coordination functionality. During evolution, support for new qualities is added and the platform scope expanded. Thus, new products can utilise the platform. In Figure 4, the evolved platform (Platform´) includes support for the quality attribute QA_4. But the expansion generates a "rubber-band effect" where the support levels (represented by the interfaces in Figure 4) for different qualities must be lowered and a "semantic gap" is introduced. This "gap" requires extensive redesign and re-implementation of applications designed for previous platform versions. The "semantic gap" emerges when the platform no longer provides the high-level support that integrates and coordinates the behaviour of several quality attributes. Instead only fundamental or restricted support with less integration is provided; leaving integration and coordination issues for the platform users to solve (illustrated by a smaller coordination rectangle in the right-most picture in Figure 4).

Example:

> This issue as was identified as a risk (and later resolved) in one of the case study organisations. Hence, we cannot provide an example where this actually occurred. But still the organisations believe that this is an important

issue. Therefore, we present the rationale for the design decisions in the ERV DPE system, which illustrates what would have happened if ERV had chosen a different approach. The designers of the DPE at ERV identified this problem early in the design process. They faced a difficult design decision of whether to initially provide a toolbox with a set of low-level primitives supporting qualities or a platform with more integration support for quality aspects included. If they chose a toolbox approach, the feeling was that leaving the integration aspect of quality support entirely to the platform users, gave application developers "too much freedom". On the other hand, they realized that they didn't have sufficient knowledge about platform usage. So, addition of support for other qualities was a likely development in future platform evolution. The outcome of this philosophical reasoning was to do two things. Initially they narrowed the scope of the platform supporting fewer quality attributes and lowered the support and integration level. This initial trade-off made it possible for future evolution of the platform as both the domain and platform use became better known.

Causes:

Business — Different business considerations contribute to this issue. The foremost are adjustments to different customer categories. A platform developer needs to provide a flexible platform to increase the number of potential users. According to ENEA and ERV, customers can roughly be dived into two categories; low-end and high-end. Of course there also exist users that would be categorised as somewhere in between these categories. To summarise their description of these categories, low-end user prefers high level support, since their platform use is circumscribed, while high-end user is much more demanding. When low-end users occasionally use it they need simple access to for instance default behaviour and nothing else. The high-end users on the other hand typically develop a system that is heavily dependent on behaviour provided by the platform. They also require full control of this behaviour so that they can fine-tune the platform for a specific application using low-level primitives.

Technical — The foremost technical problems are all connected to the knotty nature of software qualities. We previously discussed cross-cutting qualities and quality domains not being "good domains" [10]. This means that when a new quality is added and integrated into a platform it typically require redesign of the platform internals. Designers need to approach this problem cautiously to guarantee behavioural consistency for other qualities and with previous platform versions. Adding a single quality to a platform also result in an explosion of new probable quality configurations. In order to provide platform users with means of configuration the APIs require redesign, adding support for configuring the newly added quality attribute(s).

Existing Solutions:

None of the case study organisation that expressed this concern believed that there was any, generically applicable, solution to this problem. One approach is of course to initially lower the support levels and narrow the platform scope, as ERV did. But sooner or later platform developers will face problems similar to what we have described here. The inherent complexity and number of facets of this problem make it difficult, with current state-of-the-art technology, to bring forward methodologies that make this issue tractable.

Title: SERVICE LEVEL FOR QUALITY REQUIREMENTS SUPPORT

Domain: Provider

Description:

This issue is also related to the evolution of quality attribute platforms. The previous discussion was about extending the support provided by a quality platform introducing support for additional qualities. This issue focuses on another evolutionary aspect, improved support for existing qualities. Better understanding of the problem domain and how applications use the platform, together with technical innovations, allow developers to find ways to improve and simplify the use (interface) of the quality support services. Improvements include simplified access to the behaviour and more quality coordination integrated in the platform. Problems surface when applications that use a new quality support interface, run in conjunction with applications that use earlier versions of the platform interface. This problem is a variant of the well-known "interface-evolution" problem, addressed in software engineering for decades. Here, the "client-interface-implementation" dependencies are related to support for quality attributes, which adds a twist of complexity. If the level of abstraction is raised, providing more high-level support, an opaque barrier can emerge which does not allow for fine-grained control of the quality related behaviour. What happens is that the quality scope of the platform becomes more restricted when the flexibility is limited. Applications that, for some reason, need to diverge from the pre-declared behaviour in the evolved interface and integrated coordination, must find "workarounds" for these problems that usually involve violations of the intentional layering and encapsulation.

We illustrate this problem in Figure 5. Here, two applications have been designed and developed for an old version of the platform (these are depicted by the two gray diamonds). The third application, depicted by the white diamond, has been developed for the latest release of the platform. If the two application categories, i.e. systems built on the new and old platform variant respectively, should run simultaneously using the same platform version there are two options at hand. The applications based on the

old version could be redesigned and re-implemented to adapt to the changes made in the platform interface. Alternatively, the platform can open up its interfaces or provide alternative interfaces that reflect the interfaces of the earlier platform releases. These provide openings for the applications to work directly against lower-level primitives supporting different quality attributes. If that procedure succeeds new problems arise. The behaviour of the platform cannot be guaranteed. The platform takes care of certain activities related to quality attributes (e.g. process scheduling), which must be coordinated at the platform level. If some application bypasses the high-level interface and directly accesses the platform, different problems may surface. For instance, coordinating the high-level behavioural mechanisms provided in the new version with user controlled behaviour implemented with low-level primitives is, in the general case, insurmountable. User controlled behaviour will typically interfere with the platform functionality and can cause unexplainable misbehaviour immediately or even worse further down the line. Secondly, bypassing the intentional layering will break the design that was intended to leverage other qualities, such as maintainability, hence making future evolution more difficult.

Example:

To exemplify the difficulty of promoting support for quality attributes, we describe the situation at ENEA and their approach to resolve (avoid) the problem. ENEA/OSE is primarily an operating system family with real-time properties. OSE systems are used in different domains, such as telecommunication and transportation. Software systems in these areas often have a high-availability requirement. OSE has system services, which can be used to dynamically modify systems, i.e. dynamically update and migrate components. They have chosen to confine their support for this type of activities, providing only basic mechanisms, and leave it to applications to handle it. This is due to intricate problems with conflicting requirements. One of the more complex problems is how to perform updates and still retain the real-time property. For instance, in order for the system to know when to schedule an update, it must predict how much time this update will consume. It is obviously difficult for a platform to take on this kind of responsibility, for any type of application. By avoiding the inclusion of advanced control logic for updates ENEA leaves the responsibilities to its end-users. The intricate problem of providing flexible, easy to use and still dependable services is left unsolved due to its complex nature.

Causes:

Business — Among several business considerations that cause these kind of problems are “market diversity” and “lack of market communication”, the most prominent. It is extremely important to keep in mind that “not all

developments are improvements for all customers". A company developing a platform for an external customer must consider all customer groups and their requirements, before changing the platform interfaces. If not, some group will fall outside the platform scope with the result that the customer potentially stops using the platform. If the customer is internal, the causes are more organisational than business and should be more easily managed. Still, we use the word "should", though we know by experience that it could be even more difficult to get acceptance from in-house customers for radical changes and, from a user perspective, response on detailed product specific requirements.

Technical — The technical causes for these problems are connected to the fact that the cross-cutting nature of quality attributes. If the support should be improved, more coordination has to be built-in in the platform. This action will limit the flexibility provided by the platform interface. The lack of flexibility relates to the many possible versions and configurations that different quality attributes in combination may cause. Compared to functional features, for instance a file dialog box in a graphical user interface, the variability for a combination of quality attributes is more or less infinite. The exponential growth regarding the implementation of quality attributes comes from the inherently bad cohesion of quality attributes in software systems. The problems are analogous to those found when developing applications with 4th-generation languages. As long as developers stay inside the predefined scope of the language, the system behaves as expected. But as soon as they step out of the scope, several problems occur.

Existing Solutions:

Some organisations mentioned the possibility to provide several different APIs that provided different support levels, but they believed that this would introduce several problems during maintenance and still not solve the integration problem, making systems using different support levels run in parallel on the same platform. A variant of the multi-levelled API is of course to keep the support at low levels and provide special packages (applications) with more advanced support. This is the approach advocated by ENEA that provide separate tools that provide high-level support for different attributes while the API is kept at a lower level.

Title: DYNAMIC QUALITY FOOTPRINT

Domain: Provider, User

Description:

The idea of a dynamic quality footprint is a conclusive statement founded in different problems expressed by several of the case study organisations. It emerged during the discussions and all organisations contributed to different

aspects of this formalisation. When all of them are using slightly different terminology and focus, this issue is a reduction of their statements. The notion of a "dynamic quality footprint" is, to the best of our knowledge, a variability type not identified earlier at all. Several situations require modifications to the quality footprint, for instance, as depicted in Figure 7, changes in the execution environment or changes of the systems operational mode. To illustrate this specific type of product-family, we use an example. Create an n-dimensional space where each dimension corresponds to one non-functional requirement for an application. Figure 6 depicts a Kiviat diagram of the multi-dimensional "quality space" and we see that when the "level" of each quality is set, we have a unique footprint, which represents the "quality of this system" at a given point in time. During execution the different qualities may vary along different axis, forming new footprints. A platform providing support for certain qualities, must provide for the specification of these dynamically changed qualities. Specification includes declaring when a change should take place and definition of a new quality footprint for the application. When a platform does not support a dynamic quality footprint, application developers implement it at the application level. Still, the platform must provide sufficient reconfiguration primitives for reconfiguration at runtime. For instance, if there is a scheduling mechanism in the platform, the scheduling strategies must be modifiable and changes to it must be directly reflected in the behaviour. The problem in this case is that these implementations are scarcely anything else than suboptimal. Several other problems arise when direct manipulation of the platform is moved to the application. Usage that circumvents the platform interfaces introduces maintenance problems when the platform evolves. It is also difficult for platform providers to perform exhaustive testing of the platform if its exact use is not known at the time of development.

Example:

The systems produced by ERA are designed to handle continuous operation. New pieces of hardware and software can be introduced dynamically. As the system evolves, the quality requirements change. For instance, new hardware affects performance-related load-balancing activities, which elevates thresholds that trigger such activities. The contrasting event, i.e. hardware failure, also requires dynamic changes to qualities. Currently there is little or no provision for this in the platform. Consequently, developers are forced to implement this at the application level.

Causes:

Business — There are some business related causes for this kind of problems. Most often they can be inferred from risk reduction activities. For instance, instead of including a rather complex mechanism such as "dynamic

quality footprint”, it is better to provide a sufficient set of mechanisms in the API and sufficient openness in the system. This allows platform users to include limited support for dynamic changes to the quality footprint. Another important factor is, of course, financial considerations. Full support for dynamic quality is expensive both in terms of resource usage and money and not all users are willing to pay the associated costs.

Technical — The technical difficulties are numerous. For instance, there is no generally applicable specification method for quality requirements that can be used at run-time. This can be deduced from the fact that most quality attributes are not discrete, i.e. difficult to parameterise. Other problems lie in the area of dynamic reconfigurations. For a large group of users, platform providers must build a “generic” reconfiguration engine, supporting monitoring techniques that measure the quality attribute, a agent/process/daemon that verifies that quality attribute throughout operation, a mechanism to predict the quality attribute effects of a configuration change and a technique to perform configuration changes at run time [5].

Process — There are also some causes that can be deduced to the development process. Dynamic reconfigurations are not a “standard” design consideration and by far not every design process includes this option. Hence it is extremely difficult for platform developers to provide a mechanism that suits all users. Inclusion of a mechanism will impose certain design directives and restrictions.

Existing Solutions:

We have previously discussed realization techniques used to implement specific qualities in a system. Many state-of-the-art systems allow for some (limited) change to information controlling the application quality, at least for a restricted domain. For instance, the DPE system from ERV utilises external information to calculate optimal distribution of processes. When new hardware is introduced this is taken into account when new distributions are calculated. What we see is that specific solutions can be developed, but in a product family context, reusability is important, hence more configurable solutions are desirable.

Title: QUALITY SUPPORT MISMATCHES

Domain: Provider, User

Description:

The basic definition of this problem is that two or more qualities may conflict in some cases, but are independent of or reinforcing each other in others. If it’s difficult to predict when and understand why this happens we have a problem at hand. We have found two causes for this kind of problems. One occurs when two applications, executing on the same platform

instance, require different quality behaviour, the second occurs when an application uses two platforms that have problems with coordination of their quality behaviour. In some situations an application requires a support level, which lies in between two levels supported by the platform. This requires that application developers make a trade-off decision, either to bring it one level up or one level down. This may affect the realization of other system qualities, hence resulting in a redesign of the system. The contrasting approach, with a user organisation that utilises more than one product family (platform) for a product, potentially from different vendors or other organisational units, supporting different quality attributes, will cause problems and raises many questions. None of the case study organisations faced this situation but both vendors and users expressed that this would add complexity to "an already complex world". Examples include integration problems and more fundamental problems such as conflicts between two quality realizations present in two different platforms.

Example:

As an example consider the ERV DPE-platform that supports fault-tolerance and performance. A possible situation is that one product in the family requires limited support for fault-tolerance, while another requires advanced fault-tolerance techniques with exhaustive checking at run-time. For the first application, the platform works fine as it only requires limited support from fault-tolerance mechanisms. The second application though has higher demands on the platform requiring broad support for fault-tolerance and performance. If the platform cannot handle such a situation it fails in supporting the second. An example of situations where integration mismatches can appear is when an application that utilises functionality from two SPFs, one supporting fault-tolerance and performance (scheduling and load-balancing) and one supporting security. Both platforms impose special processes that control specific aspects of a running system and both require a specific scheduling model, say threads and co-routines respectively. Here, an absolute requirement on the first platform would be flexibility in terms of support for different process models. If this is not met, there is a mismatch between these two platforms and they cannot be used in conjunction with each other. The second example is a direct quality mismatch between the two platform implementations. At first it appears to be no integration mismatches between two platforms. The problems appear first when platform users try to trade-off between two qualities supported by different platforms.

Causes:

Business — Among the causes for this class of problems we find some that are related to different business considerations. If one considers the process and technical causes below we see that resolving these costs a lot,

hence make the platform more expensive. Integration with platforms from other vendors was the second facet of this issue. Aligning platforms so that they more easily will integrate with other vendors' platforms also cost more and will affect the overall product cost. It will also require major standardisation efforts if one should reach a degree of openness so that this problem is removed or at least diminished to a manageable level.

Process — From a process perspective, this problem is founded in variability management and scope evolution, where platform providers have not identified all application specific, quality attribute characteristics, when scoping the platform. This is yet again a problem connected to the difficulty of providing precise parameterisations and hence specifications of quality attributes. The second part is also a scoping issue. The platform is considered in isolation (with only client applications in mind), one does not consider cooperating platforms or even worse, competing platforms, when setting the platform scope.

Technical — Obviously the technical causes are numerous. If we consider the first category of mismatches, where one platform has difficulties with supporting two applications with different quality requirements is also connected to the quality behaviour coordination. As the problem is described above, it is the difficulty of supporting two quality models (levels) that is the core issue. This is mainly due to the fact that such capabilities require more coordination support, hence the implementation will be more complex to engineer.

For the inter-platform coordination problem, the normal situation is that one platform is unaware of the other platform. This unawareness makes it difficult (impossible) to achieve full flexibility and maximum configurability of both platforms since that would require two, fully orthogonal, platform implementations with absolutely no side effects what so ever. If a platform is aware of another, coordination is required. Coordination aims at achieving orthogonal support for the different quality attribute behaviours supported in different platforms. By intuition this would be more easily achievable if it is different quality attributes (non-related) that should be coordinated, compared to a situation where two different implementations supporting the same or related quality should be coordinated.

Existing Solutions:

Solutions to the mismatch problems discussed above can certainly be found in several deployed systems. With hands-on integration, adapting platforms or applications one can achieve an acceptable level of integration, where applications co-exist with other applications on the same platform or where two or more platforms work together. But still, in this setting (product family), reuse is a major criterion and in order to simplify adaptation/integration

of applications and platforms more flexibility would be extremely valuable. Still there is no general solution to be found around the corner and we probably have to accept the fact that we cannot provide something that supports everything always. Still we believe that major advances can be made with relatively little effort. Careful and thought out designs that just don't focus on the particular technology (quality) to be supported, but also focus on (soft) criterions like reusability and adaptability will change the situation considerably.

Title: FLEXIBILITY & EXTENDIBILITY OF QUALITY SUPPORT

Domain: Provider, User

Description:

Flexibility and modifiability are two important qualities for any SPFAs [2]. In our study we have found that inter-organisational development of platforms introduces an additional requirement, user-side modifiability. Design for modifiability and evaluation of modifiability have been discussed earlier from a platform developer point-of view [26]. We would like to extend these discussions from the perspective of a platform user. Flexibility and extendibility are key concepts for user-side modifiability. In the previous presentations of issues, a common theme is that platforms only provide partial support for a specific quality attribute. The platform user is interested in extending and/or adapting the support provided by the platform. A possible extension could be replacement of strategies, such as scheduling algorithms. Beside the possibility to extend the platform behaviour, the platform must provide possibilities for users to configure the platform, for instance using parameterisation or hotspots.

Example:

A tentative example for this is the request for plug-and-play qualities expressed by the MECEL case. Their situation with a responsibility for integrating systems from multiple hardware manufacturers delivering products to different car manufacturers is currently a difficult task to manage. Making software from multiple producers work together from a functional perspective and on-top of that deliver the expected quality is currently managed by the process developed and used by MECEL. The work that this process prescribes is time-consuming and requires trade-offs. From a developer perspective a higher degree of flexibility in the later parts of the process would introduce more possibilities (design space extension) in the earlier phases and thus reduce the number of trade-offs made early in the development.

Causes:

Business — We have identified two major business considerations in our material. From a business perspective development cost is of course an important decision parameter and introducing hot spots where end users can configure and/or modify a platform is expensive. The second more controversial cause (at least from the developer side) is that many developers sell consultancy services for their platforms and the introduction of end-user configuration could kill (or hurt) a goose that lays the golden eggs.

Process — From a process perspective one can identify some causes for this issue. First consider scoping, how to scope a modifiable platform is not obvious. Since end users can modify both common parts and variable parts it will be even more difficult to set the scope for a platform, compared to the current situation. Other causes expressed by the cases are the lack of understanding and standardisation. From a developer perspective it is extremely difficult (impossible) to predict which parts that could be targeted for modification and/or configuration by end users. The lack of standardisation and hence understanding also has implications. Since modification of platform behaviour is a complex task one has to certify in some way that the initial developer and the end user developer have a common terminology and understanding of concepts used in a platform if it should work in a general setting.

Technical — We identify two technical considerations that limit the presence of end user configurability. First, the cross cutting nature of qualities make it difficult to provide suitable interfaces for end users where the behaviour can be modified/controlled. The second consideration is concerned with overall platform behaviour. Some cases in the case study described a situation where behaviour of a platform most be certified. In certain domains it is prohibited to use, hence market, software that has not passed an extensive certification process. This is obviously a severe strike on the idea of end-user configurability.

Existing Solutions:

The systems in this study, all provide some user side configurability. In most cases we are talking about a simple parameterisation that offer a possibility for modest modification of platform behaviour. The issue here is about taking end-user modifiability to another level providing provisioning for more advanced and complex adaptations and modifications. Obviously the introduction of these capabilities is a complex task that requires thought-out designs and implementations of platforms. Providing these capabilities in a platform implies that the scope of a platform instance is to some extent controlled by a user. This is an idea similar to the one of object-oriented frameworks. In this context it would be a configurable framework-like platform with open variation points were platform users could configure or plug-in application specific support for and coordination of qualities.

7 Discussion

Above we have presented and described several problems and issues present in our case study organisations that we believe require more investigation. The objective of this section is to generalise problems and issues and bring forward a set of research topics for SPFs used as the facilitator for quality attributes, in the form of a set of research questions. Below, we recapture the specific characteristics of our case study organisations; SPFs as a vehicle to achieve qualities in embedded systems, development of inter-organisational SPFs, and SPFs that employ dynamism and relate these to well-known SPF concepts commonality and variability analysis, and scoping.

Software Qualities

> Identification of commonalities and variabilities for two or more products has by tradition a focus on core functionality. This is also true for scoping. Introducing the concept of "feature" puts the important qualities more into focus, but we believe that this is not sufficient in order to identify all kinds of commonalities and variabilities. New interesting, earlier undiscovered problems would probably surface if we study non-functional properties. What are the commonalities and variabilities in an SPF from quality point-of-view? Inducing from a comparison of functional variation points to quality variation points, we should find a similar cross-cutting phenomenon that appeared when we studied quality realizations in isolation. Even though a local design decision, say setting performance criterions, may be dressed as a single variation point, the cross-cutting tells us that this decision will impact several other features.
>
> Another issue is how to specify variability? What is quality variability, and what happens if we extend binding of variation points beyond load-time into execution time, i.e. leave certain variation points open so that the can be bound (closed) or rebound at run-time?
>
> Other problems connected to commonality and variability analysis, from the quality point-of-view, include traditional specification issues such as preciseness. Functional specifications are discrete in the sense that either a function (or some defined variant of this function) is included or not. There are also dependencies among functionality where a specific function can require that another function is included or is excluded. Compared to functional specifications, qualities do not adhere to the simple classification, "included" or "excluded". Instead specifications express the importance of a quality and threshold values that define the boundaries where the quality requirements are meet or violated. A quality specification may also vary depending on other qualities and external parameters. One usable technique to capture external dependencies and its connections to the system quality footprint is to use operational profiles [27]. Operational profiles provide for

modelling of different system modes and consequently the systems variable quality footprint.

Given the body of problems presented above, we come to a conclusion that identifying commonalities and variabilities at the "software quality level" is much more difficult than "pure" functionality, hence scoping of platforms for realizations of qualities is extremely difficult. Currently we have seen no method or technique that sufficiently supports proper analysis of quality attributes with respect to commonality and variability. We believe that this would be valuable for SPF-architects involved in all types of development projects, not only the "quality-platforms" covered in this case-study.

Inter-organisational SPFs

In this study, we have seen examples of SPFs developed for an "off-the-shelf" market. Earlier works on SPFs have shed little or no light on problems appearing when the "customer" is another organisation. In commonality and variability analysis the key question is how to identify common and deviant parts. How can this be carried out in a controlled setting, when the potential users and sometimes domains are not known in advance? Beside the communication related problems appearing when direct, in-house, communication is not possible, there are other, possibly more complicated, problems present. When a product is developed for the "market", not all customers are known on beforehand. Hence, not all products that will use the platform and their domains are known. Even though the situation described above cause problems, some of the platforms in our case study are developed for and successfully used by out-side organisations. One important factor that makes this possible is the tight interactions between developers and potential users. The user organisations are almost always as skilled as the developing organisations. Another important factor is domain maturity. Many quality domains are mature and well known. This simplifies commonality and variability analysis, and product scoping, hence supporting the development of more generic solutions. Still we have seen many examples of typical use and evolution patterns where both users and providers would like to see much more flexible and configurable platforms. Here we get two competing requirements, openness (flexibility) of a platform versus the platform scope. Developers prefer a fixed scope with little or no flexibility, since this will require less complex implementations. Users on the other hand, sometimes prefer tailored platforms for a specific application and sometimes off-the-shelf configurable platforms. Here business considerations come in to play. Is it worthwhile to pay for a tailored platform or is an off-the-shelf product sufficient?

SPFAs that employ Dynamism

The initial objective for our case study was to investigate development and use of a special variant of SPFs, namely SPFs that employ dynamic architecture. In terms of features this means that both the configuration (availability) of features and the quality constraints for these features vary at run-time. Another important characteristic for this type of SPF is that the underlying architecture also is dynamic, i.e. it is reconfigured at run-time. Reconfigurations are used as an implementation technique for feature availability and qualities. Dynamism has been studied earlier but not directly in the context of SPFs. We clearly see a lack of capabilities in handling especially platform variability when the system opens up for dynamic changes. An important remark is that dynamic architectures facilitate both what we call "anticipated dynamic variability" but also unanticipated dynamic variabilities. The dynamic variability is anticipated during development and the system is designed with these variability mechanisms in mind. In the plug-in case, typically a "plug-in capability" (feature) is selected during development. Unanticipated variability can be managed by a dynamic updating system providing provisioning for dynamic software evolution [5]. For some types of systems it is impossible to anticipate all dynamic variabilities during development.

In this area we see immense amount of work in the future. First and foremost a well-defined and clear theoretical framework for dynamic SPFs is needed. We need to define exactly what we mean when we use "dynamic variability", dynamic feature sets, feature evolution etc.

8 Conclusions

In this paper we have presented a case study of four Swedish software companies. The focus of the study is on development and use of dynamic SPFs. The contribution of this paper is that new and interesting issues and research questions revealed by this investigation and, secondly, an in-depth analysis of issues connected to commonality, variability, and scoping. Open issues and research areas identified and discussed include

1. inter-organisational development of software product family
2. scoping and variability management for quality attributes
3. product family architectures that employ dynamism

We have also demonstrated how the issues above relate to variability management. One interesting result is why the participating organisations chose a platform approach when developing their software. The primary reason was not, as expected, shortened lead-time and better reuse. Instead all expressed that platforms were a convenient technique to support specific quality attributes, e.g. maintainability,

high-availability, performance and fault-tolerance, in an application family. In our analysis, we have found that several of the issues can be related to the absence of a method that provides techniques for scoping SPF quality, including support for proper analysis of quality attributes with respect to variability and commonality. Several issues are also related to the lack of adequate specification techniques that provide support for precise and concise specification of quality attributes. Future developments in these two areas could make the processes of developing and using quality platforms easier as platforms will be easier to build configurable and easier to tailor from the user point-of-view. Finally the third issue, concerned with dynamic variability. At present the concept of dynamic variability is by far not sufficiently explored. More investigation is needed in order to understand its applications and consequences. The problems connected to quality attributes and variability discussed above obviously complicates scoping. Not only do we need to address functional scope but also the non-functional scope of an SPF.

The generality of a study like this can always be questioned, since the number of organisations investigated is small. For instance it its difficult investigate if all problems connected to different quality attributes are general or specific to the combinations our case-study organisations face. Therefore, future work in this area involves further investigations where we will attempt to strengthen the argumentation for the problems identified. We also plan to study how variability management can be improved for non-functional requirements and, more in-depth, study of the dynamism aspect and how this is managed in the architectural design process.

9 Acknowledgements

We would like to thank the people at MECEL AB, ERV in Gothenburg and ERA and ENEA/OSE in Stockholm for their participation in this case study. This material is based upon work sponsored by the Swedish National Board for Industrial and Technical Development (NUTEK), Swedish Agency for Innovation Systems (VINNOVA), Växjö university and the ECSEL program at Linköping university.

References

[1] IEEE-Std-1471-2000. IEEE recommended practice for architectural description of software-intensive systems. 1471-2000.

[2] J. Bosch. *Design & Use of Software Architectures - Adopting and Evolving a Product Line Approach*. Addison-Wesley, 2000.

[3] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 1997.

[4] J. van Gurp, M. Svahnberg, and J. Bosch. On the notion of variability in software product lines. In *Proceeedings of WICSA 2001*, August 2001.

[5] P. Oreizy, N. Medvidovic, and R.N. Taylor. Architecture-Based Runtime Software Evolution. In *Proceedings of the International Conference on Software Engineering 1998 (ICSE'98)*. ACM, ACM-Press, April 1998.

[6] M. D. McIlroy. Mass produced software components. In *Software Engineering, Report on a conference sponsored by the NATO Science Committee*.

[7] D.L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, 1976.

[8] R.E. Johnson and B. Foote. Designing Reusable Classes. *Journal on Object-Oriented Programming*, 1(2), June 1988.

[9] L. Chung, B.A. Nixon, E. Yu, and J. Mylopoulos. *Non-functional Requirements in Software Engineering*. The Kluwer international series in Software Engineering. Kluwer Academic Publishers, 2000.

[10] J.O. Coplien. *Multi-Paradigm Design*. PhD thesis, Vrije Universiteit Brussel, 2000.

[11] M. Lindwall. *An Empirical Study of Requirements-Driven Impact Analysis in Object-Oriented Software Evolution*. PhD thesis, Linköping University, 1997.

[12] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting object-interactions using composition-filters. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, *Object-based distributed processing*, volume 791 of *LNCS*, pages 152–184. Springer Verlag, 1993.

[13] W. Harrison and H. Ossher. Subject-oriented programming: a critique of pure objects. In *The eighth annual conference on Object-oriented programming systems, languages, and applications (OOPSLA'93)*, pages 411 – 428. ACM Press, Sep. 1993.

[14] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *ECOOP'97*, Lecture Notes on Computer Science, 1997.

[15] J. Bosch. *Layered Object Model - Investigating Paradigm Extensibility*. PhD thesis, epartment of Computer Science, Lund University, Nov. 1995.

[16] M. Shaw and D. Garlan. *Software Architecture, Perspectives on an emerging Discipline*. Prentice-Hall, Inc., 1996.

[17] R. E. Filman. Achieving "ilities". Presented at the OMG-DARPA-MCC Workshop on Compositional Software Architectures, January 1998. Available at: `http://www.objs.com/workshops/ws9801/papers/`.

[18] C. Szyperski and R. Vernik. A case for tired component frameworks. Presented at the OMG-DARPA-MCC Workshop on Compositional Software Architectures, January 1998. Available at: `http://www.objs.com/workshops/ws9801/papers/`.

[19] D. C. Luckham and J. Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, 12(9):717–734, Sept. 1995.

[20] J. Andersson. Towards reactive software architectures. Licentiate Thesis. 769, Linköpings universitet, May 1999. In Linköping Studies in Science and Technology.

[21] J. Bosch. Product-line architectures in industry: A case study. In *Proceedings of the 21st International Conference on Software Engineering*.

[22] M.L. Griss. Implementing product-line features with component reuse. In *6 th International Conference on Software Reuse*.

[23] M.L. Griss. Implementing product-line features by composing component aspects. In *First International Software Product Line Conference*.

[24] J-M. Debaud and K. Schmid. A systematic approach to derive the scope of software product lines. In *Proceedings of the 21st International Conference on Software Engineering*, pages 34–43, 1999.

[25] A. Arora and S.S. Kulkarni. Component Based Design of Multitolerant Systems. *Transactions on Software Engineering*, 1998.

[26] PO. Bengtsson. Design and evaluation of software architecture. Licentiate thesis, Department of Software Engineering and Computer Science, Blekinge Institute of Technology.

[27] J.D. Musa. *Software Reliability Engineering*. McGraw-Hill, 1998.

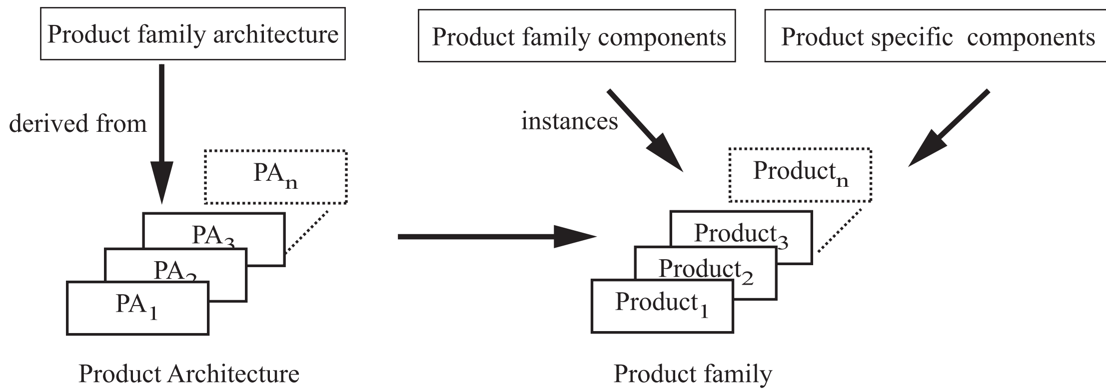

Figure 1: Elements of a product family

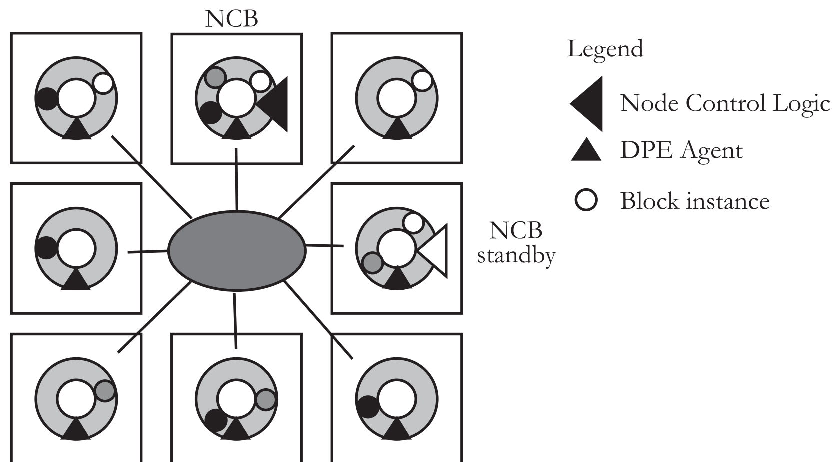

Figure 2: DPE architecture instance

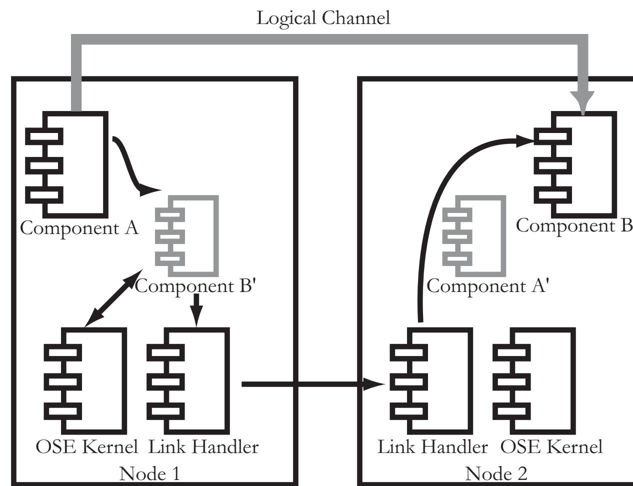

Figure 3: Logical channels in OSE Δ

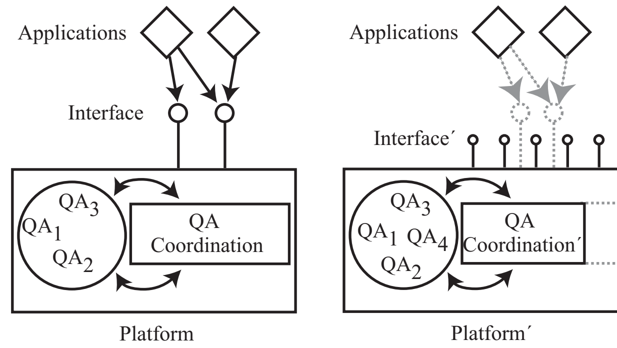

Figure 4: Improving support for quality attributes

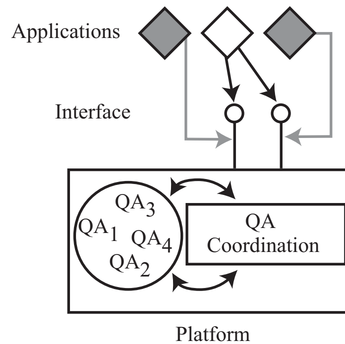

Figure 5: Introducing higher-level support for quality attributes

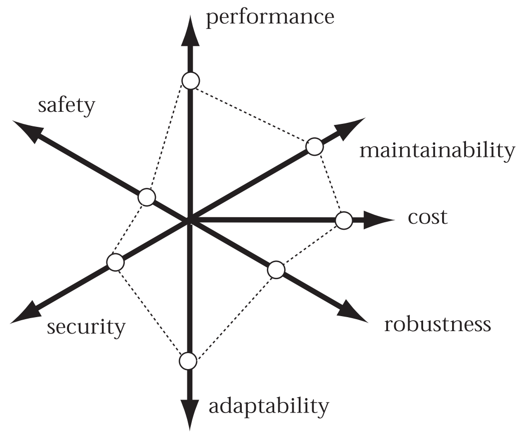

Figure 6: Quality footprint in a Kiviat diagram

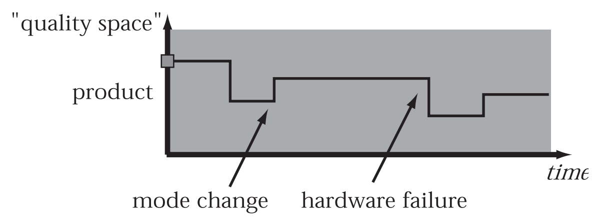

Figure 7: Dynamic Changes to the “quality footprint”