COMPOSITIONALITY IN SOFTWARE PLATFORMS

Christian Prehofer, Jilles van Gurp, Jan Bosch
Nokia Research, Helsinki, Finland

1 Introduction

Software product lines or platforms have received attention in research, but especially in industry. Many companies have moved away from developing software from scratch for each product and instead focused on the commonalities between the different products and capturing those in a product line architecture and an associated set of reusable assets. This development is, especially in the embedded systems industry, a logical development since software is an increasingly large part of products and often defines the competitive advantage. When moving from a marginal to a major part of products, the required effort for software development also becomes a major issue and industry searches for ways to increase reuse of existing software to minimize product-specific development and to increase the quality of software. A number of authors have reported on industrial experiences with product line architectures. Early work on goes back to the late 90ies [Bass et al. 97, Macala et al. 96, Dikel et al. 97] and the field has since then developed in new directions to extend the applicability of the concept.

This paper addresses several challenges posed by the increasing range of successful product lines and develops new concepts for a compositional approach to software product lines [Ommering & Bosch 02, Ommering 02, and Bosch 06]. Further, the implications of this approach to the different aspects of software development are discussed, including also process and organizational questions.

Although the concept of software platforms is easy to understand in theory, in practice there are significant challenges. As discussed in [Bosch 00], the platform model is supposed to capture the most generic and consequently the least differentiating functionality. However, the product specific functionality frequently does not respect the boundary between the platform and the software on top of it. Innovations in embedded systems can originate from mechanics, hardware or software. Both mechanical and hardware innovations typically have an impact on the software stack. However, due to the fact that the interface to hardware is placed in device drivers at the very bottom of the stack and the affected applications and their user interface are located at the very top of the stack, changes to mechanics and hardware typically have a cross-cutting effect that causes changes in many places both below and above the platform boundary.

A second source of cross-cutting changes is software specific. New products often enable new use cases that put new demands on the software that can not be captured in a single component or application, but rather have architectural impact. Examples include adding security, a more advanced user interface framework or a web-services framework. Such demands result in cross-cutting changes that affect many places in the software, again both above and below the platform boundary.

Software product families have, in many cases, been very successful for the companies that have applied them. Due to their success, however, during recent years one can identify a development where companies are stretching their product families significantly beyond their initial scope. This occurs either because the company desires to ship a broader range of products due to, among others, convergence, or because the proven success of the product family causes earlier unrelated products to be placed under the same family. This easily causes a situation where the software product family becomes a victim of its own success. With the increasing scope and diversity of the products that are to be supported, the original integration-oriented platform approach increasingly results in several serious problems in the technical, process, organizational and, consequently, the business dimension.

The “conventional”, integration-oriented platform approach in software product lines typically splits R&D into a platform organization which delivers a platform as a large, integrated and tested software system with an API that can be used by the product teams to derive their products from. In addition, there often is top-down, hierarchical governance by the central platform organization including the central management of requirements and roadmap, complete control of features, variability and product derivations and integration and testing for product platform and derivations

A specific refinement of the integration-oriented platform approach is the hierarchical platform approach [Bosch 06], where the platform is used a basis and extra functionality is added on top without modifying the base. This is however only suitable if the base is well defined and sufficient for fast product creation. Second, it does not limit the sharing of code between different, derived products beyond the borderline of the base platform.

Despite the success of the integration-oriented platform approach, in this paper we argue that the approach suffers from limitations when the scope of the product line starts to increase. An intuitive reaction adopted by some companies is to adopt a hierarchical platform approach. However, as discussed in [Bosch 06], this approach fails to be sufficiently scalable for large systems with many product derivations, especially when there is a wide range of products with unpredicted variations and in the case of product derivations requiring to modify and re-engineer significant parts of the system.

A main reason to question the traditional platform approach is that we have seen the organization into platform and product units to create a number of drawbacks regarding integration. First, as integration and testing are done both at the platform and at the product level, this can lead to an extremely high cost of integration if the scope of a software product line widens. This strong dependence on repeated integration also leads lack of predictability and inefficiency. Second, the broad scope of the platform functionality impedes flexibility and time to market for new features. Third, this easily leads to a lack of flexibility and unacceptably long development cycles because of the brittleness of platform caused by its inability to go beyond initial scope. Finally, software developed in a product unit cannot properly be reused by other units before it has been included into a release of the platform unit. As the platform unit has typically longer release cycles, such reuse of software is difficult to achieve in such an approach. In other words, we need to enable horizontal sharing between products in addition to vertical sharing between the platform and products.

Another significant fact is the growing significance of open-source and off-the-shelf components in software product lines. This presents several challenges to the traditional approaches to software product lines as it requires a more open approach which does not assume full control of features, roadmaps and tools.

As software development is now a key competence and differentiating factor in many areas such as embedded devices, there is a strong need for a new approach. We argue that we need a fundamental paradigm shift in the way that we engineer software product lines if we are to achieve a significant improvement in productivity and time to market.

This paper details a more flexible and more open approach, called the *compositional product family approach*, which addresses the above issues. The main idea of the compositional approach to software product lines is that the software platform of the product line is not a fully integrated software solution, but a set of components, architecture guidelines, principles as well as testing, documentation and use case support. Based on this flexible and open platform environment, the full products are composed, integrated and tested. While in this approach software component technology will play a much bigger role, we also propose in this paper the use of “architecture slices”, which are fragments of the full architecture. These are integrated and tested as part of the compositional software platform. Such architecture slices are typically cover one or more subsystem of the architecture. In this way, we can ensure integration and testing which goes beyond component-level testing. This is also important regarding non-functional requirements, beyond the scope of individual components. This process is illustrated in Figure 1 below. A critical point is that the architecture slices are not a full integration and some external component dependencies have to be made explicit.

Adopting a compositional product family approach causes an intentional shift of integration and testing responsibilities to the product creation, where products are composed from the aforementioned elements. In our experience, this approach is more adequate for product families with wide scope, fast evolution and open innovation and competition at the component level.

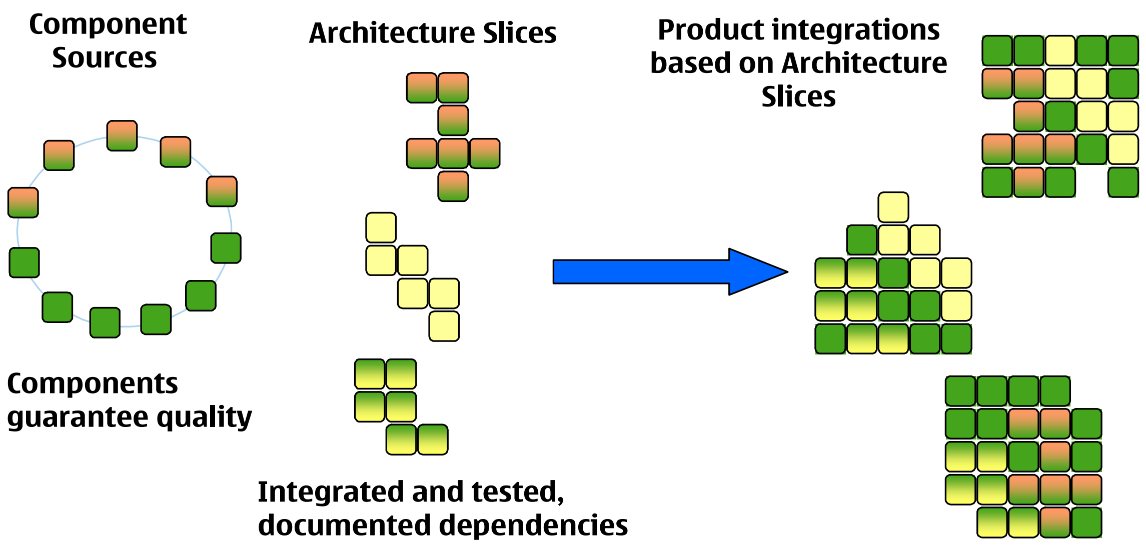
Component Sources
Architecture Slices
Product integrations based on Architecture Slices
Components guarantee quality
Integrated and tested, documented dependencies

Figure 1: Compositional Approach

The motivation and basic characteristics of the compositional product family approach were introduced in Bosch 06]. The approach is detailed and enhanced in this paper, in particular by the architecture slice concept. The contribution of this paper is the following. First, we present a detailed and precise assessment of the problems of the integration-oriented platform approach. Second, we impart the consequences of adopting a compositional product family approach on all aspects of SW development, including requirements, process, organization, architecture and tools.

The remainder of this paper is organized as follows. In the next section, we introduce the integration-oriented approach, discuss its disadvantages and introduce the compositional approach as an alternative. Subsequently, in section 3, the concept of architecture slices is presented as a key element of the compositional approach. Section 0 presents a set of research challenges for the evolving and maturing the compositional approach.

2 From Integration-oriented to the Compositional Approach

This chapter discusses and presents an alternative to the traditional, integration centric approach to product families. However, before we can discuss this, the concerns of the integration-oriented platform approach need to be defined more clearly. Traditionally, product families are organized using a strict separation between the domain engineering organization and the product organizations. The domain engineering organization employs a periodic release cycle where the domain artifacts are released in a fully integrated and tested fashion, often referred to as a platform. The product organizations use the platform as a basis for creating and evolving their product by extending the platform with product-specific features.

The platform organization is divided in a number of teams, in the best case mirroring the architecture of the platform. Each team develops and evolves the component (or set of related components) that it is responsible for and delivers the result for integration in the platform. Although many organizations have moved to applying a continuous integration process where components are constantly integrated during development, in practice significant verification and validation work is performed in the period before the release of the platform and many critical errors are only found in that stage.

The platform organization delivers the platform as a large, integrated and tested software system with an API that can be used by the product teams to derive their products from. As platforms bring together a large collection of features and qualities, the release frequency of the platform is often relatively low compared to the frequency of product programs. Consequently, the platform organization often is under significant pressure to deliver as many new features and qualities during the release. Hence, there is a tendency to short-cut processes, especially quality assurance processes. Especially during the period leading up to a major platform release, all validation and verification is often transferred to the integration team. As the components lose quality and integration team is confronted with both integration problems and component-level problems, in the worst case an interesting cycle appears where errors are identified by testing staff that has no understanding of the system architecture and can consequently only identify symptoms, component teams receive error reports that turn out to originate from other parts in the system and the integration team has to manage highly conflicting messages from the testing and development staff, leading to new error reports, new versions of components that do not solve problems, etc.

Although several software engineering challenges associated with software platforms have been outlined, the approach often proves highly successful in terms of maximizing R&D efficiency and cost-effectively offering a rich product portfolio. Thus, in its initial scope, the integration-centric approach has often proven itself as a success. However, the success can easily turn into a failure when the organization decides to build on the success of the initial software platform and significantly broadens the scope of the product family. The broadening of the scope can be the result of the company deciding to bring more existing product categories under the platform umbrella or because it decides to diversify its product portfolio as the cost of creating new products has decreased considerably. At this stage, we have identified in a number of companies that broadening the scope of the software product family without adjusting the mode of operation quite fundamentally leads to a number of key concerns and problems that are logical and unavoidable. However, because of the earlier success that the organization has experienced, the problems are insufficiently identified as fundamental, but rather as execution challenges, and fundamental changes to the mode of operation are not made until the company experiences significant financial consequences.

In the following, we detail these problems, first regarding the scope of the platform, then regarding openness.

2.1 Problems from Over-extended Scope

So, what are the problems causing a mode of operation that was initially so successful to turn into such a problematic approach? In the list below, we discuss the problems regarding scope that one can observe and perceive directly.

- **Lack of component generality**: Although most components were useful for most products in the initial scope of the product family, in the expanded scope, the number of components that is only used in a subset of products is increasing.
- **Incorporation of immature functionality**: As discussed in the previous bullet, in the initial scope most functionality useful for one product is likely to become relevant for other products over time. Hence, there often is a tendency to incorporate product specific functionality into the platform very early on, sometimes already before it has been used in the first product. When the scope of the family increases, the disadvantages of incorporating immature functionality become apparent.
- **Slow evolution of functionality**: As the scope of the product family increases and the organization maintains an integration-oriented approach to developing the shared software artifacts, the response time of the platform in response to requests to add functionality of existing features increases.
- **Implicit dependencies**: In an integration-oriented approach, a relative high degree of connectivity between components is accepted as there are few disadvantages at that stage and it increases short-term developer productivity. When the scope of the product family increases, the components need to be composed in more creative configurations and suddenly the, often implicit, dependencies between components become a significant problem for the creation new products.
- **Unresponsiveness of platform development**: Especially for product categories early in the maturation cycle, the slow release cycle of software platforms is particularly frustrating. Often, a new feature is required rapidly in a new product. However, the feature requires changes in some platform components. As the platform has a slow release cycle, the platform is typically unable to respond to the request of the product team. The product team is willing to implement this functionality itself, but the platform team is often not allowing this because of the potential consequences for the quality of the product team.

When analyzing these problems with the intention to understand their underlying causes, among others, the following causes can be identified:

- **Decreasing complete commonality**: Before broadening the scope of the product family, the platform formed the common core of product functionality. However, with the increasing scope, the products are increasingly diverse in their requirements and amount of functionality that is required for all products is decreasing, in either absolute or relative terms. Consequently, the (relative) number of components that is shared by all products is decreasing, reducing the relevance of the common platform.
- **Increasing partial commonality**: Functionality that is shared by some or many products, though not by all, is increasingly significantly with the increasing scope. Consequently, the (relative) number of components that is shared by some or

most products is increasing. The typical approach to this model is the adoption of hierarchical product families. In this case, business groups or teams responsible for certain product categories build a platform on top of the company wide platform. Although this alleviates part of the problem, it does not provide an effective mechanism to share components between business groups or teams developing products in different product categories.

- **Over-engineered architecture**: With the increasing scope of the product family, the set of business and technical qualities that needs to be supported by the common platform is broadening as well. Although no product needs support for all qualities, the architecture of the platform is required to do so and, consequently, needs to be over-engineered to satisfy the needs of all products and product categories. This however impedes extensibility and increases maintenance effort.
- **Cross–cutting features**: Especially in embedded systems, new features frequently fail to respect the boundaries of the platform. Whereas the typical approach is that differentiating features are implemented in the product specific code, often these features require changes in the common components as well. Depending on the domain in which the organization develops products, the notion of a platform capturing the common functionality between all products may easily turn into an illusion as the scope of the product family increases.
- **Maturity of product categories**: Different product categories developed by one organization frequently are in different phases of the lifecycle. The challenge is that, depending on the maturity of a product category, the requirements on the common platform are quite different. For instance, for mature product categories cost and reliability are typically the most important whereas for product categories early in the maturity phase feature richness and time-to-market are the most important drivers. A common platform has to satisfy the requirements of all product categories, which easily leads to tensions between the platform organization and the product categories.

2.2 Problems of the Closed Approach

A second major challenge to the traditional integration-oriented approach is growing importance of open source software and components off the shelf (COTS). The core issue is that the integration-oriented approach assumes a strong governance of the requirements, features and roadmaps of both the platform as well as the products. In particular, this leads to several problems:

- **Inflexible base platform:** If the base platform integrates external software, such as open source software, it does not have control over the roadmap, development process and release cycles. This leads to a number of constraints and in our experience makes it more challenging to find an architecture which suits all derived products.
- **Evolution:** In case an individual derived product extends the platform by open source software, it is again difficult to later include this software into the main product line platform.

- **Tools and organization mismatch:** Open source software employs different organizational approaches, tools and quality assurance and testing practices. These are difficult to integrate into software product lines as discussed in 0.

The consequence is that we need a more open approach which does not assume full control of features, roadmaps and tools. The compositional approach as discussed below assumes a more open environment and caters much easier with this setting of decentralized management and heterogeneous processes and tools.

2.3 The Compositional Product Family Approach

The main idea of the compositional approach to software product lines is that the product line is not an integrated, complete software solution: Instead, is provided as an open, but integrated toolbox and software environment. The role of a fully integrated reference platform is now taken over by a set of *architecture slices*, which are integrated and tested component compositions for one or more subsystems. This includes specific components with high cohesion, which is typically a vertical integration of highly dependent components and can extend to small component frameworks. The set of architecture slices shall cover the full scope of the software product line and both exemplify and enable the later product integration. While the concept of architecture slides appears similar to related work on architecture modeling [Pree 00, Kim 99, van Gurp 02], our notion of architecture slice is driven by integration and testing aspect and we consider more the overall software product line approach.

For instance, on a mobile phone a set of multimedia applications which use a built-in camera can be an architecture slice. This may include several, interchangeable camera drivers, picture taking and picture viewing applications as well as several operating system components for fast storage and retrieval of media data. These components can vary depending on the camera resolution or on the number of cameras. Further, the operating system support for the cameras may depend on the resolution and the performance needed for data storage. In addition, the dependencies to other components and architecture slices must be specified. In this example, the viewer application can interface with messaging application to send pictures directly. Also, a video conferencing application may depend the camera support for recording video calls. While these dependencies must be documented and can be tested with sample code, the actual product creation has the responsibility for the integration. More details on architecture slices are found in the following section.

In summary, a compositional software product family includes the following:

- A set of coherently managed components that facilitate easy integration.
- Overall architecture blueprints and principles that guide the later product development.
- Architecture slices which cover the full scope of the software product line and exemplify the product integration.
- Test cases and test environments for both components and architecture slices.
- Detailed documentation on dependability between components, architecture slices as well their dependencies and recommended compositions.

Based on this flexible and open environment, the full products are composed, integrated and tested.

The compositional product family approach can in addition include a fully integrated platform for certain base products without integrating all components. This can also serve as reference product in order to test integration and non-functional requirements. The main difference is that the majority of product derivation is a composition of the above pieces.

In the integration-oriented approach, the fully integrated software package must resolve all dependencies and interactions between components by actual code. In this approach this is done by integrating typical sets of components by architecture slices and by documenting the external dependencies. This documentation of dependencies is highly important and must be treated as an essential part of the compositional approach. Compared to the traditional approach, we make these dependencies explicit and leave the actual integration to the product creation.

The main advantages and technical characteristics of this approach are:

- **More flexible product architecture.**The software architecture of products is much more freely defined and not constrained by a platform architecture. This is because there is no enforced platform architecture although a reference architecture may exist to inspire product architectures.
- **Local responsibility.** The reusable components that may be used in product creation accept a much higher level of local responsibility. This responsibility means that (1) each component only uses predefined provided, required and configuration interfaces, (2) verifies at composition or deployment time that all its interfaces are bound correctly and that the component can provide the use cases or features (or parts thereof) that it promises and (3) contains intelligence to dynamically adjust itself to client or server components that either offer less than expected functionality or require more than expected functionality.
- **Reduced integration cost.** Integration of reusable and product-specific software components takes place by each product and no platform integration is performed. Due to the increased intelligence of components, the integration effort is brought down to a fraction of the current effort requirements.

Clearly, this approach leads to several challenges that concern all aspects of software development. We will discuss these in section 0. However, we first discuss the impact of the approach on different aspects of software development.

2.4 Key Differences of the Compositional Approach

To more precisely describe the compositional approach in relation to the integration-oriented approach, we discuss the two approaches from five perspectives that we believe are of predominant importance in the context of broadening the scope of software product families, i.e. business strategy, architecture, components, product creation and evolution:

- **Business strategy.** The original reason when adopting product families often is the reduction of R&D expenditure through the sharing of software artifacts between multiple products. Although this is certainly not ignored in the compositional approach, the main focus is typically to maximize the scope of the product family. Although R&D cost and time-to-market are obviously relevant factors for any technology driven organization, the most important rationale for the composition-oriented approach is that by giving product developers more flexibility and freedom, the creation of a much broader set of products is facilitated.

- **Architecture.** Initially, the architecture for the product family is specified as a complete structural architecture in terms of components and connectors. The architecture is the same for all products and variation is primarily captured through variation points in the components. When moving towards a compositional approach, the structural view of the architecture is diminished and increasing focus is directed towards the underlying architectural principles guaranteeing compositionality. As discussed earlier, the key difference between this approach and other approaches is that the architecture is not described in terms of components and connectors, but rather in terms of the architectural slices, design rules and design constraints.
- **Components.** During the first phase of a software product family, the components are implemented for a specific architecture that is specified in all or most of its aspects. The components contain variation points to satisfy the differences between different products in the family, but these do not spread significantly beyond the interfaces of the components. Finally, the components are implemented such that they depend on the implementation of other components rather than on explicit and specified interfaces. When evolving towards a compositional approach, the focus on components remains. However, these components are not developed ad-hoc, but are constrained in their implementation by the architecture, i.e. the architectural fragments, rules and constraints discussed earlier. As each component is integrated in one or more architecture slices, compositionality of these slices is ensured.
- **Product creation.** The integration-oriented model typically assumes a pre-integrated platform that contains the generic functionality required by all or most products in the family. A product is created by using the pre-integrated platform as a basis and adding the product-specific code on top of the platform. Although not necessarily so, often the company is also organized is along this boundary. The approach works very well for narrowly scoped product families, but less well when the scope of the product family is broadening. In the compositional approach, the explicit goal is to facilitate the derivation of a broad range of products that may need to compose components in an equally wide range of configurations. Product creation is, consequently, the selection of the most suitable components, the configuration of these components according to the product requirements, the development of glue code in places where the interaction between components needs to be adjusted for the product specific requirements and the development of product-specific code on top of the reusable components.
- **Evolution.** Often, in an integration-oriented approach, there is a strong preference towards incorporating new features and requirements into the pre-integrated

platform. The reasoning behind this is that new features, in due time, need to be provided in all products anyway and consequently, the most cost effective approach is to perform this directly. The alternative, intended approach where product-specific functionality evolves and commoditizes over time and is incorporated into the platform is often diminishing. In the compositional approach, product teams as well as component teams can be responsible for the evolution of the code base. Product teams typically extend existing components with functionality that they need for their product but is judged to be useful for future products as well. Product teams may also create new components for the same purpose. Component teams, if used, are more concerned with adding features that are required by multiple products. A typical example is the implementation of a new version of a communication protocol.

3 Components and Architectural Slices

In the above we have the composition oriented approach. In this section we elaborate compositional software development based on the key ingredients, components and architecture slices. The decentral nature of compositional development makes it possible to combine various styles of development since essentially, the composed artifacts are developed independently.

3.1 Component Technology

The key difference between integration oriented and composition oriented development is that the compositional approach is decentralized. The components that are being composed are developed independent from each other and from the products they are to be used in. A definition of a component that suits this chapter well, is the one provided by Clemens Szyperski in his book on components: *"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties"* [Szyperski 1997].
In other words, components are developed indepently and then integrated by a third party. We interpret the "unit of composition" broadly here. In our experience, depending on the level of abstraction, very large software systems, are broken down into hundreds of components that may still be very large.

As noted in the introduction, the key activity in the integration oriented approach is the integration phase during which critical errors in the software and its components are identified and then fixed. A compositional approach to software development allows product and component developers to operate independently. Consequently, it is a more scalable approach that both allows more developers to work together (by decentralizing their work) and to build larger software products (by combining their output). However, integration testing is inherently difficult to accomplish when developing in a compositional way because of the independent modes of operation of both product and component teams and the absence of a central planning and decision making.
Of course, the product developer will need to test the particular configuration of components that is used in the product as well as any product specifics. However, the product developers shall not change the components in the configuration (aside from

manipulating them through the configuration API). Also it is much harder for product developers to demand from the developers of the components that they fix bugs, implement or change features, etc. This may be hard for various reasons:

- The component is provided by external developers (e.g. sub contractors, a commercial off the shelf component (COTS) vendor or an open source project). These parties may have some commitment towards supporting and maintaining their components (e.g. governed through a support contract) but are unlikely to care much about product specific issues manifesting themselves during product integration.
- The component is developed according to its own roadmap with planned major and minor releases. Any issues that do not fit in this roadmap are unlikely to be addressed on short notice.
- The required changes conflict with those required by other users of the component. Components may be used by multiple products. In the case of external components, these may even be competing products. When confronted with such conflicts, the solution will have to be provided in the product rather than in the component.

Consequently, the components that are used in the product need to be stable and well tested well before the product integration phase. In other words, adopting a compositional approach implies putting more emphasis on component testing and makes this the responsibility of the component development teams.

In case essential functionality for a specific product is not provided by the component environment and it cannot be added by extensions or glue code, it may be needed to add a derivative component. This will happen more frequently in the compositional approach as roadmaps and features are less coordinated. We see this option however as an advantage as it increases the internal competition. Using the compositional approach for development it may be expected that these components may eventually become of use to other products developed by the same organization.

3.2 Architecture Slices and Components

The approach we outline in this section addresses component integration testing using the notion of architecture slices and partial integration. An architecture slice does not describe a full product architecture. Rather it describes relevant aspects of the environment a component is expected to be used in.
The IEEE 1471 standard defines software architecture 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"* [IEEE1471 2000]. In the case of compositional development, it is not possible for component developers to consider all products their component is to be used in. Furthermore, all these products may have little more commonality other than the fact that they use the component. In other words, rather than focussing on the full architecture, component developers need to focus on the part of the architecture that relates directly to their component. We refer to this as an architectural slice.

An architecture slice is a set of densely coupled components that are recommended to be used in this combination in products. In most cases, this represents a subsystem of the architecture. For example, in a mobile phone this can be a set of multimedia applications for a built-in camera and include drivers or a set of components for media playing. An architecture slice also defines its external dependencies on other components or subsystems. An architecture slice is not complete without these external dependencies and must describe its relation to these external subsystem. In other words, architecture slices include assumptions about how other subsystems relate to it.

3.3 Architecture Slices and Integration Testing

As argued earlier, an essential part of the compositional approach is the integration testing of architecture slices. For this, a major issue is that the dependencies to external components need to be addressed. In order to test a component or a architecture slice, a developer will need to provide an environment that fulfills these dependencies. Using the resulting architecture slice configurations, simple applications may be implemented that test various aspects of the component functionality. In the case where the intended use is extension by another component, creating such extensions is the preferred way to test.

The dependencies to external components can be categorized in two groups:

- Uses dependencies. The component likely depends on other components. This may either be specific versions of specific components or, as is increasingly common in the Java world, implementations of a specific standard API.
- Usage dependencies. These dependencies indicate other components that depend on this component or must at least work correctly with the component.

The dependency relations may also coupled so that e.g. using component A implies also using component B. Such relations should of course be documented.

The process of integration testing is normally done as part of product development. However, as indicated earlier, this is generally too late to address any component issues. Consequently, integration testing needs to be done earlier at the component and architecture slice level. While it is impossible to realize complete products as a part of the component testing process, it is feasible to provide the environment used for testing representative of known or anticipated uses of the component in actual products:

- For the uses dependencies a selection of components may be used that is likely to be used by product developers. If compatibility is important, various configurations of the architecture slice with different components may be created to e.g. test various versions of the same component or different implementations of the same API. In the situation that a component extends another one, the relation to the extended component can be characterized as a uses relation as well. In that case, selection is easy because it is always the same component that is being extended.
- The usage dependencies may be more difficult. In some cases it may be possible to test using a complete product configuration. However, when developing a new version of the component (and particularly when API changes are involved), it is not likely that existing components are compatible. In that case, usage dependencies may need to be simulated using mock implementations.

Additionally, in the case of commercial products, the full product software may not be available to the component developer.

3.4 Component Dependencies

While in general it is not possible to test all combinations of all dependencies, it is possible to determine combinations of components that are known to work as expected. This information can be provided in documentation.

This practice is quite common in the software industry. For example, the release notes of Apache's Jakarta Commons Logging component (version 1.1.0) state that: "All core classes were compiled with a 1.2.x JDK. JCL may work on some augmented 1.1 series JREs but it is recommended that those wish to run on 1.1 JREs download the source and create a custom implementation". This is a nice example of a dependency that might work but is not recommended because the component developers did not include it in their testing procedure. The recommendation clearly indicates that users are discouraged from using a 1.1 JRE but that it may be possible to do so if needed. This is also an example where product development may choose to create their own versions of components at their own responsibility.

This practice of documenting working and recommended combinations of components is quite common. Many component vendors will certify that their software works in combination with certain other components and will be able to provide more extensive support to users if they use the recommended components in combination with component releases they produce. This certification and support model is the basis for most open source product companies like MySQL and JBoss.

In practice this causes product developers to (strongly) prefer release quality components over development releases (if available) and to satisfy any dependencies those components have using the recommended components. Doing so allows them to rely on the test work that has been done already by the component developers and focus more on testing the product specifics.
A second consequence is that this makes fulfilling the dependencies a decision process that is preferably completed early in the product development process. Generally, in a new product, most dependencies will be fulfilled during the product architecture design phase. Upgrades to new versions may of course occur but it is not likely that product managers will want to risk having to be part of the component integration testing process.
These two practices are markedly different from the practice of co-evolving components and products in a software product line. During the integration phase, components are continuously integrated with development versions of the other components. Likewise, product developers will end up using modified versions of the components thus negating some of the earlier integration testing effort. The practices outlined above, strongly discourage this from happening and allow product developers to build on a well tested foundation.

3.5 Examples

In this section we have characterized how component developers can do integration level without building a full product based on their component. This is important for creating compositions of components because it allows product developers to rely on the integration testing already done, rather than having to do this themselves.

A convincing example of this is the Debian linux distribution. The Debian linux distribution is a collection of thousands of open source software packages running on linux. The stated goal of the Debian foundation is to provide a stable, fully tested and integrated distribution. Essentially, most of their work consists of integrating the thousands of packages into their distribution. While there is some Debian specific development, most of it consists of Debian specific infrastructure and gluecode. Additionally the integration testing feedback is propagated to the dependent open source packages, often along with patches for the problem.

In [Amor-Iglesias et al. 2005] some impressive statistics are presented regarding the size of this distribution: release 3.1 (a.k,a, Sarge) was measured to consist of 230 million lines of code (MLOC). The 3.0 release only three years earlier was measured at 105 MLOC and the 2.1 release (according to [Wheeler 2002]) was 55 MLOC. In other words the distribution has quadrupled in size in roughly five years. According to Wheeler, who has applied the COCOMO model to these metrics, this corresponds to multiple billions of dollars worth of investment; requiring thousands of software engineers to work together in a timeframe that exceeds the actual time spent delivering these versions of Debian, which is governed by a small foundation funded by donations from industry and individuals. In other words, the fully integrated approach that COCOMO models would not be good enough to produce a software system comparable in size to Debian.

An interesting development in recent years is the emergence of open source projects where software is co-developed by developers working for or financed by competing organization. This trend is motivated by the above metrics: it is the only cost effective way to develop large software packages with many software engineers in a short time frame. If the software is not differentiating the core products, continuing under an open source license may actually make them more differentiating.

The natural tendency of companies to protect investments and intellectual property is in direct conflict with this and poses an organizational and strategic challenge. A good example of a company that has overcome this reluctance, is IBM. Five years ago they released the popular java development environment eclipse under an open source license. Later they even transferred development and ownership of this product to an independent foundation. Doing so had benefits for them that outweighed the lost sales of their visual age product. First of all many of their competitors have since contributed to the projects and the resulting development environment is now the industry standard development platform. Because IBM is heavily associated with the eclipse product that means that other companies have lost differentiating power while IBM gained some. Additionally, while IBM continues to invest much resources in eclipse, much of the investment is now shared with their competitors. So either they cut some cost here or they managed to improve and innovate at a lower cost (compared to doing everything in house). Also some things that IBM was not interested in were financed by others and are now also of use to IBM customers. Finally, their strong involvement in this product makes IBM an interesting partner for related products and services such as the middleware, hardware and consultancy services IBM provides. Arguments along the same line may be found in the Cathedral and the Bazaar [Stallman 1999].

4 Research Challenges of the Compositional Approach

The compositional approach represents a potential improvement for organizations currently using a software product line approach for developing their software. However, there are many challenges that will need to be addressed. In this section, we aim to provide an overview of these challenges.

4.1 Decentralized Requirements Management

A consequence of using a compositional approach is that requirements of the integrated products are managed separately from those of the individual components that are used in the system.

In the integration-oriented software product line development, requirements are managed centrally. When developing a new product based on the product line software, the product architect identifies which requirements are product specific and which are fulfilled by the product line. The product line evolution in turn is centrally governed and driven by common requirements across products and other requirements that are believed to be useful for future products. Development of individual components in the product line is driven by these centrally managed requirements.

Characteristic of the compositional approach is that there is no central management of requirements and features. Product developers select components and architecture slices based on how well they support the product requirements but may also consider other factors such as, for example:

- Ability to influence component requirements. Even if the requirements do not match 100%, the ability to influence the roadmap of the component may be decisive.
- Component roadmap. The advertised component roadmap may include items that are currently not relevant for the product but might become relevant in the future
- Reputation. The component may have an established reputation with respect to important quality attributes.
- Openness. While often advertised as black boxes, many components require a level of understanding of the internal component design that effectively makes them white boxes. Arguably, this is an important reason for the apparent lack of success of COTSs. Additionally, it is an important factor in the success of open source components in the current software industry.

A successful component will be used by many products that may have little in common aside from the fact that they somehow depend on the component. Component requirements may be driven by a number of factors:

- Feedback on potential improvements from existing component consumers. In case of a commercial relation between producer and consumer there may also be some contractual terms (e.g. in the context of a support contract).
- Market analysis of product requirements of products that currently do not use the component. Evolving the component to support those requirements presents an opportunity to grow market share.
- External factors such as standards. Component requirements may be (partially) based on standardized or de-facto specifications. When such specifications evolve, supporting the evolved specification can become a requirement.
- Internal factors, such as improving quality factors that are important to the component developers such as e.g. maintainability. Other factors may include the personal interest of the developers that are involved to explore design alternatives or realize small improvements.

This approach results in a more bottom up approach where instead of being tailored to a specific set of products (i.e. top down approach) there tends to be an organic bottom up process where more or less independently components are selected and put together to fulfill products requirements. In this way, potential conflicts between independently developed components are eventually resolved by the component selection and integration process.

4.2 Quality Management & Architecture

A characteristic of the compositional approach is that there is no central architecture. Instead, product and architecture slices each have their own architectures. This poses a number of interesting research challenges with respect to e.g. applying quality assessments methodologies, which mostly assume having a centrally managed architecture and full control over the assets governed by this architecture.

The intention of applying quality assessment methods is to verify conformance to centrally managed quality requirements (which do exist in a compositional approach) and to improve product quality by addressing any identified quality issues. However, performing conventional architecture assessment at the product level makes relatively little sense due to the lack of control over the composed components.

- Consequently, quality assessment and improvements needs to happen at the component or possibly architecture slice level. However, given the lack of central quality requirement managed and the lack of control over depending and dependent components, this means that guaranteeing system level quality with respect to quality requirements such as real-time constraints, throughput, security, etc. is difficult. Component developers need to anticipate quality requirements of their potential customers and convert this anticipated demand into component improvements.
- A second issue is that architecture assessment methods mostly require an already integrated system. In a compositional approach, one would like to consider impact on quality before the components are integrated. Any identified problems might lead to component improvements but might also lead to the selection of alternative components.

A second goal of having explicit software architecture is to enforce architectural style and design rules. The reason for this is that this ensures that the architectural components fit together. A problem with commercial of the shelf components (COTS) has been finding components with matching interfaces. The absence of a centrally governed architecture does not mean that there are no guiding architectural principles. Necessarily, components that are going to be used together must share common architecture. At least a significant level of compatibility is required. Small differences can be bridged using glue code. However, it is not very desirable to create significant amounts of glue code when creating products. Consequently, the compositionality poses a number of interesting new challenges:

- How to document architectural properties of components and architecture slices?
- How to optimize product architecture such that is optimal for the components it will be composed off?
- How to design components such that they do not impose too many constraints.

4.3 Software Component Technology

Component oriented programming and later web services have been advocated as a major step forward in building large software systems [Stal 00]. This has been largely due to providing standardized component infrastructures with well defined APIs and services as well as interoperability support. While this has been significant progress, it does not address the issues of variability and architecture as in software product lines.

In software product lines as well as many tools for configuration management, dependencies between components are managed. Managed dependencies are clearly a basic ingredient for our compositional approach. Yet, this covers only the management of syntactic code dependencies, e.g. regarding versions and API compatibility. The information in such systems can also include tested configurations and can hence be used to describe architecture slices.

Interesting in this context is also the work presented in [Ommering 02]. Van Ommering is a proponent of so called populations of product lines and introduces component technology that supports this. His approach is very similar to ours but focuses more on the technical aspects of composing components rather than the other aspects that we discuss in this chapter.

For our target of compositional software products lines, we need components which can also work in unforeseen use cases and environments. This for instance means more robustness and awareness of dependencies and interactions with other components. More specifically, the two main challenges we discuss here are semantic dependencies which must be managed more explicitly and, secondly, that components are designed in a more robust way.

Semantic interaction between components means that the component behavior has to be adapted when composing with others. This goes well beyond the syntactic compatibility of APIs and has been examined extensively in the context of feature interaction research [Calder 00]. These interactions can be positive or negative:

- Positive interaction require additional functionality to be added. E.g. if we have an email client on a mobile phone and a picture viewer, it should be possible to email pictures from the viewer.
- Negative interaction require to disallow some cases or removal of ambiguities. Typically, two components or features contradict in their behavior or compete for resources which are limited. For instance, the silence mode of a phone should not disable the alarm clock.

In fact, some dependencies occur only when several components are combined and cannot be observed for two features at a time [Prehofer 01]. Negative feature interaction related problems can be very hard to find and are also not likely to be identified in the decentralized integration testing approach outlined earlier.

4.4 Process and Organizational issues

As discussed in [Bosch 02], there are several ways to organize software product line organizations. Adopting a compositional approach makes it both possible and necessary to organize differently. As argued in earlier sections, a key characteristic of a compositional approach is that there is less central management of requirements, architecture and implementation. Essentially development and evolution of components happens in a decentralized fashion. Decoupling of product and component development is an explicit goal of a compositional approach because it allows decision making that affects products to be separate from decision making that affects components.

Doing this in one organization introduces a contradiction in the sense that an organization typically has goals, targets and a mission. All activity conducted by the organization (including component and product development) follows (or should follow from) this overall mission. This implies that product and component development are not independent at all. Consequently, introducing a compositional approach in a product line organization introduces a number of challenges:

- How to organize such that product development teams have the freedom to select external components or initiate development of new components rather than using the internally developed component. The business decision of a product team to not use the in house developed component has negative consequences for the component team. The best technical solution may not be the best for the organization as a whole. Balancing such difficult and conflicting interests is a key challenge that needs to be addressed.
- Allowing component teams to take responsibility for their own roadmap and architecture may lead to a situation that resources are spent on feature development that is not going to be used by any of the product teams. Balancing innovation of component teams and product development is required to keep development cost under control.
- An issue in any organization is the distribution of resources (money, people, time, etc) over the organizational units. Essentially product teams and component teams are all competing for the same resources. However, product teams are typically the only organizational units directly contributing to revenue, which leads to a bias in their favor. Adopting a compositional approach therefore needs to create

an internal value chain or market mechanism to distribute the internal development resources.

- While initially, components may be used only by product teams, components may themselves become products that are potentially interesting for other software developing organizations. Where this does not conflict with product differentiation, it would be desirable to market such components as separate components or share the burden of developing such components with other companies, even if these companies are competitors. Productizing or open sourcing internally components is a natural side effect of a fully implemented compositional approach but may also introduce new requirements that are well outside the scope of product development.

For all these organizational challenges a careful balance needs to be made between the conflicting interests of component and product teams and the overall corporate mission. However, it should be noted that this is also true for a de-compositional approach. The reason for converting from a software product line based development approach to a compositional one is that central management of all these decisions becomes harder as development grows in scale and as the software product line scope widens.

5 SUMMARY

Software product families have found broad adoption in the embedded systems industry, as well as in other domains. Due to their success, product families at several companies experience a significant broadening of the scope of the family. We have discussed several key issues which can arise from this and also from other trends like external or open source software which is outside the control of one organization.

For product families which aim primarily to be open and cover a wide range of products, we have proposed the concept of compositional product families. This approach relies on a decentralized organization that gives the product creation more flexibility and responsibility. Also it gives similar freedom to internal component developers.
Instead of a fully integrated platform, we rely on the new concept of architecture slices to ensure integration and testing beyond component level testing. Furthermore, we have shown that this compositional approach must include all aspects of software development in order to be successful. Additionally have identified several key research challenges of this new approach with respect to requirements, quality management, software component technologies as well as processes and split of responsibility within an organization.
A topic that interests us at Nokia, where development depends mostly on integration oriented approaches, is that of cross-cutting behavior and non-functional requirements such as performance and power consumption. This has proven to be extremely hard already in the integration oriented approach we depend on currently.

REFERENCES

[Bosch 00] J. Bosch, Design and Use of Software Architectures: Adopting and Evolving a Product Line Approach, Pearson Education (Addison-Wesley & ACM Press), ISBN 0-201-67494-7, May 2000.

[Bosch 02] J. Bosch, Maturity and Evolution in Software Product Lines: Approaches, Artefacts and Organization, Proceedings of the Second Conference Software Product Line Conference (SPLC2), pp. 257-271, August 2002.

[Bosch 06] Jan Bosch, Expanding the Scope of Software Product Families: Problems and Alternative Approaches, Proceedings of the 2nd International Conference on the Quality of Software Architectures (QoSA 2006) , June 2006.

[Ommering & Bosch 02] R. van Ommering, J. Bosch, Widening the Scope of Software Product Lines - From Variation to Composition, Proceedings of the Second Software Product Line Conference (SPLC2), pp. 328-347, August 2002.

[Ommering 02] R. van Ommering, Building product populations with software components, Proceedings of the 24th International Conference on Software Engineering, pp. 255 – 265, 2002.

[Pree 00] Wolfgang Pree, Kai Koskimies Framelets—small and loosely coupled frameworks, ACM Computing Surveys, Volume 32 , 2000

[Kim 99] T Kim, YT Song, L Chung, DT Huynh Dynamic Software Architecture Slicing, - COMPSAC, 1999

[van Gurp 02] Jilles van Gurp, Rein Smedinga, Jan Bosch, Architectural Design Support for Composition and Superimposition, proceedings of HICCS 2002.

[Stal 00] Michael Stal, Web services: beyond component-based computing, Communications of the ACM, Volume 45 , Issue 10 (October 2002)

[Calder 00] M Calder, M Kolberg, EH Magill, S Reiff-Marganiec, Feature interaction: a critical review and considered forecast, Computer Networks, 2003, Elsevier

[van Gurp 06] Jilles van Gurp, OSS Product Family Engineering, First International Workshop on Open Source Software and Product Lines at SPLC 2006. Available from http://www.sei.cmu.edu/splc2006/

[Prehofer 01] Christian Prehofer, „Feature-Oriented Programming: A New Way of Object Composition“, Concurrency and Computation, Vol. 13, 2001.

[Szyperski 1997]. C. Szyperski, Component Software - Beyond Object Oriented Programming. Addison-Wesley 1997.

[IEEE1471 2000] IEEE Std P1471-2000, Recommended Practice for Architectural Description of Software-Intensive Systems, IEEE, 2000.

[Wheeler 2002] David Wheeler, More Than a Gigabuck: Estimating GNU/Linux's Size, http://www.dwheeler.com/sloc/redhat71-v1/redhat71sloc.html, 2002.

[Amor-Iglesias et al. 2005] Juan-José Amor-Iglesias, Jesús M. González-Barahona, Gregorio Robles-Martínez, and Israel Herráiz-Tabernero, Measuring Libre Software Using Debian 3.1 (Sarge) as A Case Study: Preliminary Results, Upgrade: the european journal for the informatics årofessionals, vol 6(3), June 2005.

[Dikel et al. 97]. D. Dikel, D. Kane, S. Ornburn, W. Loftus, J. Wilson, ‘Applying Software Product-Line Architecture,’ IEEE Computer, pp. 49-55, August 1997.

[Macala et al. 96]. R.R. Macala, L.D. Stuckey, D.C. Gross, ‘Managing Domain-Specific Product-Line Development,’ IEEE Software, pp. 57-67, 1996.

[Bass et al. 97]. L. Bass, P. Clements, S. Cohen, L. Northrop, J. Withey, ‘Product Line Practice Workshop Report, Technical Report CMU/SEI-97-TR-003, Software Engineering Institute, June 1997.

[Stallman 1999] Eric S. Raymond, The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary. O'Reilly & Associates 1999