Software Product Families: Towards Compositionality

Jan Bosch

Nokia, Technology Platforms/Software Platforms,
P.O. Box 407, FI-00045 NOKIA GROUP, Finland
Jan.Bosch@nokia.com
http://www.janbosch.com

Abstract. Software product families have become the most successful approach to intra-organizational reuse. Especially in the embedded systems industry, but also elsewhere, companies are building rich and diverse product portfolios based on software platforms that capture the commonality between products while allowing for their differences. Software product families, however, easily become victims of their own success in that, once successful, there is a tendency to increase the scope of the product family by incorporating a broader and more diverse product portfolio. This requires organizations to change their approach to product families from relying on a pre-integrated platform for product derivation to a compositional approach where platform components are composed in a product-specific configuration.

Keywords: Software product families, compositionality.

1 Introduction

Over the last decades, embedded systems have emerged as one of the key areas of innovation in software engineering. The increasing complexity, connectedness, feature density and enriched user interaction, when combined, have driven an enormous demand for software. In fact, the size of software in embedded systems seems to follow Moore's law, i.e. with the increased capabilities of the hardware, the software has followed suit in terms of size and complexity. This has lead to a constant struggle to build the software of embedded systems in a cost-effective, rapid and high-quality fashion in the face of a constantly expanding set of requirements. Two of the key approaches evolved to handle this complexity have been software architecture and software product families. Together, these technologies have allowed companies to master, at least in part, the complexity of large scale software systems.

One can identify three main trends that are driving the embedded systems industry, i.e. convergence, end-to-end functionality and software engineering capability. The convergence of the consumer electronics, telecom and IT industries has been discussed for over a decade. Although many may wonder whether and when it will happen, the fact is that the convergence is taking place constantly. Different from what the name may suggest, though, convergence in fact leads to a portfolio of increasingly diverging devices. For instance, in the mobile telecom industry, mobile phones have diverged into still picture camera models, video camera models, music

M.B. Dwyer and A. Lopes (Eds.): FASE 2007, LNCS 4422, pp. 1 – 10, 2007.

player models, mobile TV models, mobile email models, etc. This trend results in a significant pressure on software product families as the amount of variation to be supported by the platform in terms of price points, form factors and feature sets is significantly beyond the requirements just a few years ago. The second trend is that many innovations that have proven their success in the market place require the creation of an end-to-end solution and possibly even the creation or adaptation of a business eco-system. Examples from the mobile domain include, for instance, ring tones, but the ecosystem initiated by Apple around digital music is exemplary in this context. The consequence for most companies is that where earlier, they were able to drive innovations independently to the market, the current mode requires significant partnering and orchestration for innovations to be successful. The third main trend is that a company's ability to engineer software is rapidly becoming a key competitive differentiator. The two main developments underlying this trend are efficiency and responsiveness. With the constant increase in software demands, the cost of software R&D is becoming unacceptable from a business perspective. Thus, some factor difference in productivity is easily turning into being able or not being able to deliver certain feature sets. Responsiveness is growing in importance because innovation cycles are moving increasingly fast and customers are expecting constant improvements in the available functionality. Web 2.0 [7] presents a strong example of this trend. A further consequence for embedded systems is that, in the foreseeable future, the hardware and software innovation cycles will, at least in part, be decoupled, significantly increasing demands for post-deployment distribution of software.

Due to the convergence trend, the number of different embedded products that a manufacturer aims to bring to market is increasing. Consequently, reuse of software (as well as of mechanical and hardware solutions) is a standing ambition for the industry. The typical approach employed in the embedded systems industry is to build a platform that implements the functionality common to all devices. The platform is subsequently used as a basis when creating new product and functionality specific to the product is built on top of the platform. Several embedded system companies have successfully employed product families or platforms and are now reaching the stage where the scope of the product family is expanding considerably. This requires a transition from a traditional, integration-oriented approach to a compositional approach.

The contribution of this paper is that it analyses the problems of traditional approaches to software product families that several companies are now confronted with. In addition, it presents compositional platforms as the key solution approach to addressing these problems and discusses the technical and organizational consequences.

The remainder of this article is organized as follows. The next section defines the challenges faced by traditional product families when expanding their scope. Subsequently, section 3 presents the notion of compositional product families. The component model underlying composability is discussed in more detail in section 4. Finally, the paper is concluded in section 5.

2 Problem Statement

This paper discusses and presents the challenges of the traditional, integration-oriented approach to software product families [1] when the scope of the family is extended. However, before we can discuss this, we need to first define integration-oriented platform approach more precisely. In most cases, the platform approach is organized using a strict separation between the platform organization and the product organizations. The platform organization has typically a periodic release cycle where the complete platform is released in a fully integrated and tested fashion. The product organizations use the platform as a basis for creating and evolving theirs 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.

In figure 1, the approach is presented graphically. The platform consists of a set of components that are integrated, tested and released for product derivation. A product derivation project receives the pre-integrated platform, may change something to the platform architecture but mostly develops product-specific functionality on top of the platform.

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-oriented platform 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.

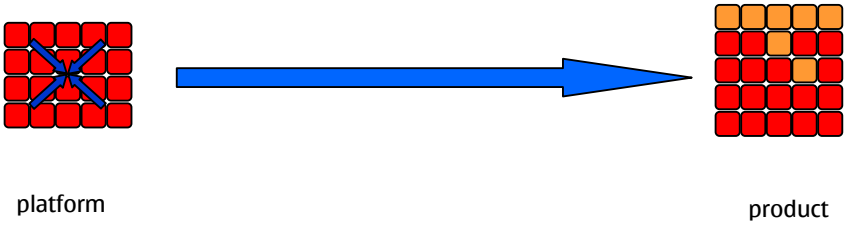

Fig. 1. Integration-oriented approach

The problems and their underlying causes that one may observe when the scope of a product family is broadened considerably over time include, among others, those described below:

1. **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.
2. **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.
3. **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.

4. **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 (category) 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.
5. **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.
6. **Unresponsiveness of platform:** 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.

3 Towards Compositionality

Although software product families have proven their worth, as discussed above, there are several challenges to be faced when the product family approach is applied to an increasingly broad and diverse product portfolio. The most promising direction, as outlined in this paper, is towards a more compositional approach to product creation. One of the reasons for this is that in the integration-oriented approach all additions and changes to the platform components typically are released as part of an integrated platform release. This requires, first, all additions and changes for all components to be synchronized for a specific, typically large and complex, release and, second, easily causes cross-component errors as small glitches in alignment between evolving components cause integration errors.

The compositional approach aims to address these issues through the basic principle of independent deployment [6]. This principle is almost as old as the field of software engineering itself, but is violated in many software engineering efforts. Independent deployment states that a component, during evolution, always has to maintain “replaceability” with older versions. This principle is relatively easy to implement for the *provided interfaces* of a component, as it basically requires the

component to just continue to offer backward compatibility. The principle however also applies to the *required interfaces* of a component. This is more complicated as this requires components to intelligently degrade their functionality when the required interfaces are bound to components that do not provide functionality required for new features. Thus, although the principle is easy to understand in abstract terms, the implementation often is more complicated, leading to situations where an R&D organization may easily abandon the principle.

If the principle of independent deployment is, however, adhered to, then a very powerful compositional model in the context of software product families is created: rather than requiring the evolution of each component or subsystem to be perfectly aligned, in this approach each component or subsystem can evolve separately. Because each component guarantees backward compatibility and supports intelligent degrading of provided functionality based on the composition in which the component is used, it facilitates a "continuous releasing" model, allowing new functionality to be available immediately to product derivation projects. In addition, quality issues can, to a much larger extent, be dealt with locally in individual components, rather than as part of the integration.

Although the approach described in this section has significant advantages for traditional product families, the broadening product scope of many families creates an increasing need for creating *creative configurations* [3]. Some typical reasons for creative configurations include:

- **Structural divergence:** As discussed earlier, the convergence trend is actually causing a divergence in product requirements. Components and subsystems need to be composed in alternative configurations because of product requirements that are deviating significantly from the standard product.
- **Functional divergence:** A second cause for requiring a creative configuration is where platform components need to be replaced with product specific components to allow for diverging product functionality.
- **Temporal divergence:** In some cases, the divergence between product requirements may be temporal, i.e. certain products require functionality significantly earlier than the main, high volume product segment for which the platform is targeted. Although every product family has leading, typically high-end, products feeding the rest of the product portfolio with new functionality, in this case the temporal divergence is much more significant than in those cases. This may, among others, be due to the need to create niche products or because of the need to respond more rapidly to changing market forces to an extent unable to be accounted for by typically slow platform development.
- **Quality divergence:** Finally, a fourth source of divergence is where specific quality attributes, e.g. security or reliability, require the insertion of behaviour *between* platform components in order to achieve certain quality requirements. Although the structure of the original platform architecture may be largely maintained, the connections between the components are replaced with behavioural modules that insert and coordinate functionality.

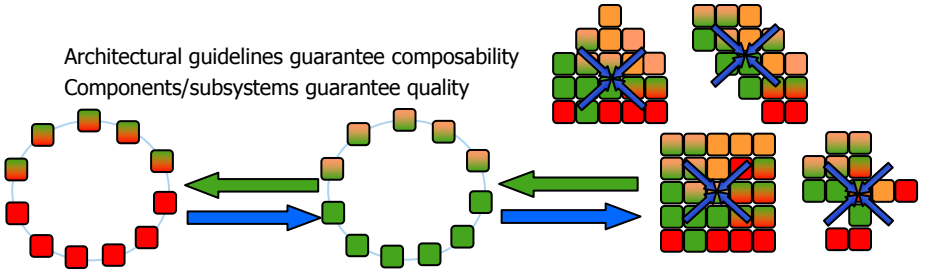

Fig. 2. Compositional approach to software product families

In figure 2, the compositional approach is presented graphically. The main items to highlight include the creative product configurations shown on the right side and the fact that there are two evolutionary flows, i.e. from the platform components towards the products and visa versa.

In the paper so far, we have provided a general overview of the compositional approach to software platforms. However, this approach has bearing on many topics related to software product families. Below, we discuss a few of these.

Software variability management: In the research area of software product families, software variability management (SVM) is an important field of study. One may easily argue that the topics addressed in this paper can be addressed by employing appropriate variability mechanisms. In our experience, SVM is complementary to employing a compositional approach as the components still need to offer variation points and associated variants. In [5] we argue that SVM focuses primarily on varying behaviour in the context of stable architecture, whereas compositionality is primarily concerned with viewing the elements stable and the configurations in which the elements are combined to be the part that varies. In practice, however, both mechanisms are necessary when the scope of a product family extends beyond certain limits.

Software architecture: In most definitions of software architecture, the predominant focus is on the structure of the architecture, i.e. the boxes and lines. In some definitions, there is mention of the architectural principles guiding development and evolution [5], but few expand on this notion. In the context of compositional product families, the structural aspect of software architecture is become increasingly uninteresting from a design perspective, as the structure of the architecture will be different for each derived product and may even change during operation. Consequently, with the overall increase of dynamism in software systems, software architecture is more and more about the architectural principles. In [2], we argue that architectural principles can be categorized into architecture rules, architecture constraints and the associated rationale.

Software configuration management (SCM)**:** At each stage of evolving an existing component, there is a decision to version or to branch. Versioning requires that the resulting component either contains a superset of the original and additional functionality or introduce a variation point that allows the functionality provided by the component to be configured at some point during the product derivation lifecycle. Branching creates an additional parallel version of the component that requires a

selection during the product derivation. Although branching has its place in engineering complex software product families, it has disadvantages with respect to managing continued updates and bug fixes. It easily happens that, once branched, a component branch starts to diverge to the point that the product originally requiring the branching lacks too many features in the component and abandons it.

4 Component Model for Compositional Platforms

The Holy Grail in the software reuse research community has, for the last four decades, been that components not developed for integration with each other can be composed and result in the best possible composed functionality. In practice, this has proven to be surprisingly difficult, among others because components often have expectations on their context of use. In the context of the integration-oriented approach, we see that components typically have more expectations on components both providing and requiring functionality and that these expectations, paradoxically, that are less precisely and explicitly defined. In contrast, composition-oriented components use only explicitly defined dependencies and contain intelligence to handle partially met binding of interfaces.

For the software assets making up a product family, at least the components and subsystems need to satisfy a number of requirements in order facilitate composability. Different aspects of these requirements as well as additional requirements have been identified by other researchers as well.

- **Interface completeness:** The composition of components and subsystems should only require the information specified in the provided, required and configuration interfaces. Depending on the type of product family, compile-time, link-time, installation-time and/or run-time composition of provided interfaces and required interfaces should be facilitated and the composition should lead to systems providing the best possible functionality given the composition.
- **Intelligent degradation:** Components should be constructed such that partial binding of the required interfaces results in automatic, intelligent degradation of the functionality offered through the provided interfaces of the component. In reality, this can not be achieved for all required interfaces, so for most components the required interfaces can be classified as core (must be bound) and non-core (can be bound). This is mirrored in the provided interfaces that degrade their functionality accordingly. In practice, most non-core interfaces represent steps in the evolution of the component or subsystem.
- **Variability management:** Non-core interfaces and configurable internal behaviour are part of the overall variability offered by a component or subsystem and needs to be accessible to the users of the component through a specific configuration or variability interface.

One of the general trends in software engineering is later binding or, in general, delaying decisions to the latest point in the software lifecycle that is still acceptable from an economic perspective. Also for embedded systems, an increasing amount of

configuration and functionality extension can take place after the initial deployment. However, for post-deployment composability to be feasible, again the software assets that are part of the product family need to satisfy some additional requirements.

- **Two descriptions:** A component requires an operational description of its behaviour (code) as well as an inspectable model of its intended behaviour.
- **Monitoring required interfaces:** For each required interface, a component has an inspectable model of the behaviour required from a component bound to the interface. This allows a component to monitor its providing components.
- **Self-monitoring:** In addition to monitoring its providing components, a component observes its own behaviour and identifies mismatches between specified and actual behaviour.
- **Reactive adjustment:** A component can initiate corrective actions for a subset of mismatches between required and actual behaviour of itself or of its providing components and is able to report other mismatches to the encompassing component/subsystem.

Concluding, although some of the techniques described in this section require more advanced solutions provided by the development environment, by and large the compositional approach can be implemented using normal software development tools and environments. The main transformation for most organizations is mostly concerned with organizational and cultural changes.

5 Conclusions

This paper discusses and presents the challenges of the traditional, integration-oriented approach to software product families when the scope of the family is extended. These problems include the decreasing complete commonality, increasing partial commonality, the need to over-engineer the platform architecture, cross–cutting features, different maturity of product categories and, consequently, increasing unresponsiveness of the platform.

As a solution to addressing these concerns we present the compositional platform approach. This approach becomes necessary when the traditional integration-oriented approach needs to be stretched beyond its original boundaries. We have identified at least four types of divergence, i.e. structural divergence, functional divergence, temporal divergence and quality divergence. The compositional platform approach is based on the principle of *independent deployment* [6]. This principle defines rules that components need to satisfy in order to provide backward compatibility and flexibly in addressing partial binding of required interfaces. In particular, three aspects are necessary but not sufficient requirements: interface completeness, intelligent degradation and variability management.

Although many product families implement or support a small slice of the principles and mechanisms, few examples exist that support a fully compositional platform approach. In that sense this paper should be considered as visionary rather than actual. However, the problems and challenges of the integration-oriented approach are real and as a community, we need to develop solutions that can be adopted by the software engineering industry.

References

1. 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.
2. Jan Bosch, Software Architecture: The Next Step, Proceedings of the First European Workshop on Software Architecture (EWSA 2004), Springer LNCS, May 2004.
3. Sybren Deelstra, Marco Sinnema and Jan Bosch, Product Derivation in Software Product Families: A Case Study, Journal of Systems and Software, Volume 74, Issue 2, pp. 173-194, 15 January 2005.
4. www.softwarearchitectureportal.org
5. 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.
6. R. van Ommering, Building product populations with software components, Proceedings of the 24th International Conference on Software Engineering, pp. 255 – 265, 2002.
7. http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html