Evolving Embedded Product Lines: Opportunities for Aspects

Aleksandra Tesanovic
Philips Research Laboratories Eindhoven, The Netherlands
aleksandra.tesanovic@philips.com

ABSTRACT

The traditional constraints on software development and architectures in the consumer electronics domain, including the low cost of manufacturing of a product, support for families of products, etc., have been a key driver for the development of component-based product lines (e.g., in consumer electronics at Philips). In this paper we show that adding new features to a product line over time results in crosscutting changes to a system and its constituting components. Given the nature of problems experienced when evolving consumer products with new features, we outline opportunities for using aspect-oriented technologies to address some of these problems.

Categories and Subject Descriptors

D.2.11 [**Software Architectures**]: Domain specific architectures; D.2.13 [**Reusable Software**]: Domain engineering, Reuse models; K.6.3 [**Software Management**]: Software Maintenance, Software Development

Keywords

Product-line architecture, component-based software, crosscutting concerns, aspects

1. INTRODUCTION

The constraints, such as low cost of manufacturing of a product, support for families of products with variability points, etc., have been a key driver in using component-based product lines for embedded software in consumer electronic (CE) products, e.g., at Philips CE. To meet the market demands, a CE producer is evolving a product line over time to incorporate new features, e.g., USB stick support, web browser, and WiFi for a TV product line. This in turn increases the amount of software in a product line and its complexity [15]. Moreover, we observed that changes introduced by evolving CE products expose characteristics of aspects as they are crosscutting the structure of an original system, from requirements to the code implementation.

In this paper, therefore, we discuss opportunities for using aspect-orientation to address some of the issues associated with evolution of consumer products with new features. The purpose of this position paper is not to present new solutions, rather to identify problems where aspects, including requirement-level, code-level, and architectural aspects, can be applied to facilitate easier evolution of existing component-based product line architectures for embedded systems. We use the Philips TV product line architecture, as an example embedded commercial product line, to illustrate the evolution problems and outline the possible ways of addressing those using aspects.

The paper is organized as follows. In section 2 we discuss the traditional constraints that have been placed on software in consumer electronics products and how they have been addressed by component-based product line solutions. We then show, in section 3, that component composition might not be optimal to satisfy the evolution of products with new features as components are not developed to handle crosscutting nature of emerging features. Further, we identify the opportunities for aspect-orientation in embedded product lines. The paper is summarized in section 4.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Workshop ACP4IS '07, March 12-13, 2007 Vancouver, British Columbia, Canada
Copyright 2007 ACM 1-59593-657-8/07/03 ...$5.00.

2. CHARACTERISTICS AND CONSTRAINTS OF CE SOFTWARE

In this section we first detail the constraints posed on software development of CE products and then discuss how these are met by introducing component-based product-lines. The positive effects of a component-based solution are also presented.

2.1 Constraints

Software used in CE products is increasing by Moore's law [16]. In TVs, for example, the amount of software has increased from tens of Kb in late eighties to tens of Mb in late nineties. Traditionally, the development of software in the CE domain is strongly constrained by product costs. Keeping the integral cost of manufacturing the product low is essential to ensuring that the company profits. This implies finding and maintaining a fine balance between the (cost of) hardware and software in a product, which in turn typically implies decreasing the cost of the hardware and thereby results in limited resources that can be used by software.

CE manufacturers, such as Philips CE, produce a large variety of categories of products, e.g., TVs, DVD recorders, and audio jukeboxes [16]. Moreover, a large range of prod-

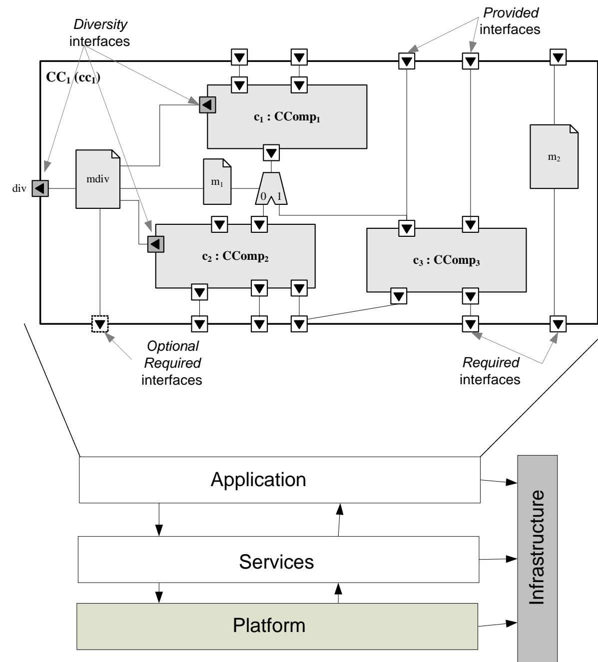

Figure 1: The component-based product line architecture of a TV

ucts are developed within each of the product categories. For example, the TV software differs depending on the region where the TV is going to be used, screen type, user interface, connectivity, etc. Due to the high volume of software being produced, the mechanisms for managing development complexity, in particular the large product range with many variation points, are also considered important.

2.2 Component-based architecture

It has been recognized already in late '90s that supporting development of a large range of products within a relatively short amount of time and acceptable costs requires highly specialized architectures and development processes. This led to explicitly developing, for TV products, components for multiple usages, and reusing those to produce a variety of TVs with different characteristics. The resulting CE TV architecture is component-based and consist of four main reusable subsystem components [16](see figure 1):

- the platform subsystem provides a stable API that abstracts TV-specific hardware;
- the infrastructure subsystem provides an abstraction of the operating system functionality, and also handles time-critical operations;
- the services subsystem is a middleware layer between the platform and the applications; and
- the applications subsystem includes the user interface for various TV applications, e.g., program installation, audio control, and video control, as well as the rules for avoiding undesirable interaction between these applications.

Each subsystem is a compound component obtained by composing smaller components, as depicted in figure 1. Every component represents a unit of functionality and is developed using the Koala component model. Koala is both a component model and an architectural description language as it defines interfaces as well as the connections between the components. A Koala component has a number of *provides* and *requires* interfaces [14]. *Provides* interfaces define the functionality of a component that can be used by other components. Hence, a component can only use functionality of other components that are defined in their *provides* interfaces. *Requires* interfaces define the functionality that a component needs from other components in order to provide the desired functionality, i.e., function correctly. Note that all *requires* interfaces of a component must be bound explicitly when instantiating the component [16]. *Diversity* interfaces are a special type of *requires* interfaces. They contain a set of parameters or functions that implement, together with constructs like switches and diversity modules, variation points in the component. *Provides* and *requires* interfaces are instances of reusable interface definitions that are defined independently from components.

2.3 Implications

The impact of using reusable Koala components and composing a system by combining components into compound components was very positive to the development of TV software. The development time of TVs within a family was drastically reduced.

To obtain a specific product family member is sufficient to instantiate the diversity parameters and functions with appropriate values, specific for a product in question. Note that the diversity management in reality is far more complex and versatile, but is beyond the scope of this paper; for details on diversity management we refer interested readers to [14]. The developers conformed to the Koala component model and in return were able to utilize good support that

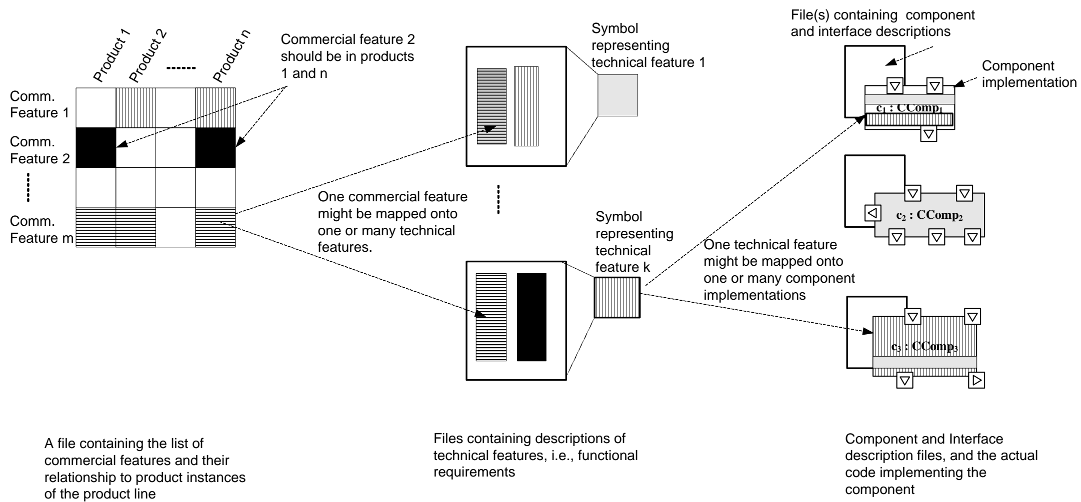

Figure 2: The steps in adding features to a product

Koala accompanying tools offer when it comes to automation of the development process. As a result, components and the product-line architecture were able to successfully cope with the constraints discussed in section 2.1.

3. EVOLVING A PRODUCT LINE

In this section we show that evolving a product line by adding new features results in crosscuttings of the system structure. To deal with some of the crosscutting issues, we outline possible opportunities for using aspect-orientation.

3.1 Crosscutting issues

The product line creation, as depicted in figure 2, actually starts by defining so-called commercial features, such as Ambilight. A commercial feature is then mapped to one or more technical features, which represent the functional requirements. A technical functional feature can, in turn, be implemented by any number of components. The mapping from commercial features to components is complex and often not formalized enough.

Figure 3 presents the evolution of a product line over time, starting from the point of product line creation, and including two evolution points. The depicted features in the figure are technical features. In the remainder of the paper, unless otherwise specified, we use the term feature to denote a technical feature. When studying the evolution of a product line in an industrial setting the following can be observed. At the point of a product line creation, an architecture is defined with an inial set of components that are implementing a given set of features. The obtained product line has a certain initial complexity and code size. For example, in the late '90s, when the TV product line is defined it had features that included manual and automatic program installation, teletext, electronic programming guide, etc.

Typically, after some time is elapsed from its creation, the product line needs to be upgraded with new features, which normally implies adding a new set of features on top of the existing ones. Currently, a feature is added to the CE software by modifying existing reusable components and adding new components (see figure 3). Hence, an initial set of components is changed as the number of existing components have been modified and new components have been added. This increases the code size and the complexity of the software. As the process of evolving products continues, more features are added, the component set is more extensively modified, and the complexity of software and code size is also increased. Since late '90s many more features have been added to the TV software. Examples are content browsing for, e.g., viewing photos on TV, USB sticks, digital TV capabilities, and WiFi connectivity. The architecture remains stable over time (see the architecture description from section 2.2).

Added complexity to the evolution is that adding of features to an existing product line always starts with adding new commercial features. Recall that one commercial feature might crosscut many (technical) feature specifications, and one technical feature can be implemented by a number of components. Therefore, it is often difficult to trace the relationship between these steps in the evolution chain, and recognize and asses the extent of needed changes, e.g., what components will be affected by adding a new feature.

As depicted in figure 4, the changes to the overall software stack, when adding a feature, are often *extensive, complex, and crosscutting*. For example, if audio needs to be evolved to include extra functionality, then audio-related component(s) that reside in the applications subsystem might need to be revisited, as do audio-related components in the service subsystem and the platform subsystem. Note that there might not be direct and a priory knowledge of the impact of changes as adhering to a component model hides the internal component information with respect to features that a component implements. Namely, while components provide separation into well-defined fine-grained units of functionality, this functionality normally entangles a number of features, which are not explicitly captured in the architectural description language.

The fact that each of the subsystems is developed by a development team at a different site attributes to the complexity of changes. Although features are added to a system in a carefully planned process, they still, due to the crosscutting nature of changes that adding or modifying a feature entails, result in many problems when integrating a system

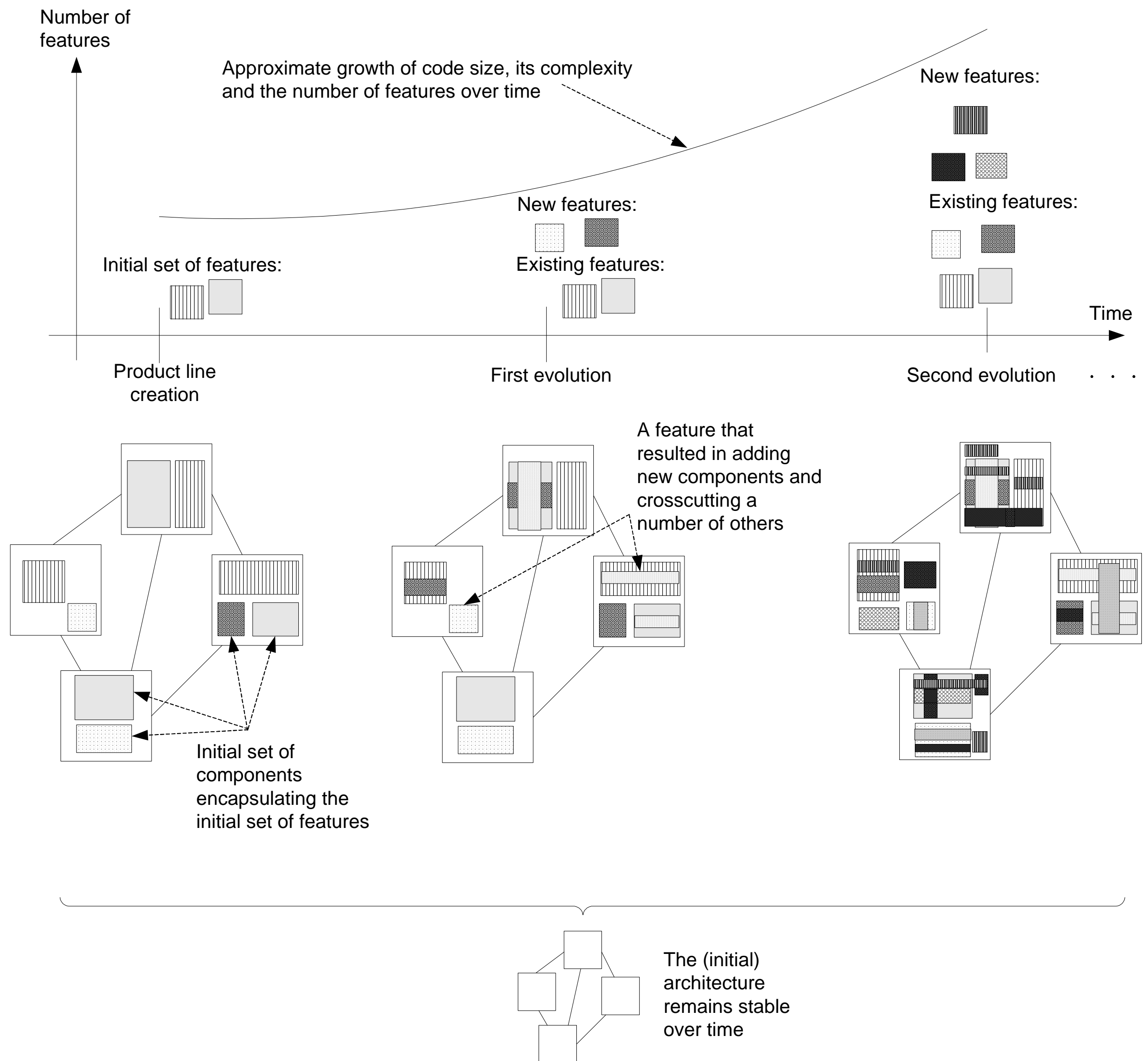

Figure 3: The time line of product line life span

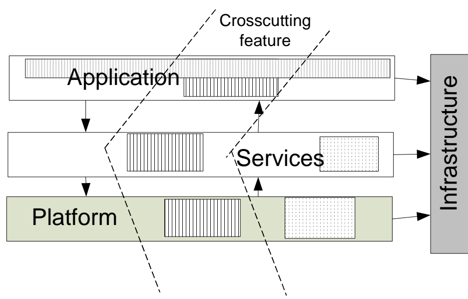

Figure 4: Extending CE software with a (crosscutting) feature

[12]. Hence, the resulting evolution is a complex and costly process.

3.2 Opportunities for aspect-orientation

From the discussion above it is clear that, while components solve the constraints mentioned in section 2.1, when the product line evolves, new mechanisms are desirable that would aid in handling crosscuttings introduced by adding new features. Hence, due to the nature of problems encountered when evolving TV software, aspect-orientation could bring added value, on several levels, for feature-rich product-lines in resource constrained embedded environments.

Requirements.

It would be useful to have explicit support for specifying features and tracing crosscuttings they introduce from commercial feature specifications to the real implementation in the code. Aspect-oriented requirements engineering (AORE) is already an area of extensive research, which provides a new way to modularize and reason about crosscutting concerns during requirements engineering [7]. The approach and accompanying tool, such as [6], are concerned with identifying requirements pertaining to concerns that are (fully or partially) scattered in the statements of other requirements. While this approach can aid in identifying the relationship between various concerns at the requirements level, the concerns here are technical ones, and it is assumed that the mapping from commercial to technical requirements is already done. Additionally, since crosscutting concerns on the requirements level are not necessarily implemented by aspects, it would also be useful to provide support for mapping or tracing these concerns onto elements of an architectural level, i.e., relating them to artifacts (components and possibly aspects) that actually implement them.

Architectural level.

Aspects could help in handling to two issues that can be observed when analyzing the evolution of a product line: structural invariance of the architecture, and crosscuttings in an architectural description language (ADL).

As mentioned before, currently the evolution is done by modifying elements of the architecture, e.g., subsystems in the TV software, and therefore throughout the evolution of a product line its architecture remains fixed (see figure figure 5(a)). This is because of the fixed component-based structure of the architecture where it is easier to modify the elements of the architecture than to modify its architecture, e.g., add additional compound components. To facilitate easier evolution it would be desirable to be able to evolve the structure of the system, as illustrated in figure 5(b). Ensuring the evolution of the architecture would imply developing approaches that enable "weaving" the new architectural elements in an existing structure while preserving the (proven) behavior of the system.

Sometimes in a description of an architecture, such as the Koala ADL, crosscuttings can also be observed. These crosscuttings often are induced by having many variability points and repeating the parameters and their initialized values in multiple (compound) components. Extracting these crosscuttings and encapsulating them into an ADL aspect-like construct that can be woven on the ADL-level would decrease redundancy in description of the architecture and increase possibility to consistently change the values of the parameters across components. Modeling approaches for aspects in an embedded environment could aid in that they would enable transforming aspect-like feature descriptions into ADL descriptions in a consistent manner. The approaches proposed for a domain of distributed (CORBA-like) middleware systems, e.g., [18, 17], could become increasingly important if the shift toward more distributed architectures materializes for the CE products [11].

Code level.

On the code level, the opportunities include applying aspect-oriented programming to encapsulate the code of new features into aspects and automatically weave them into several components. This could be a difficult task given that it is often the case that components need to maintain their black box behavior. A possibility would be to explicitly declare aspect-variation points in the components where the weaving can be done, e.g., as proposed in [10, 9] for a different domain. Of course, it would also be useful to be able to detect and extract already existing crosscuttings from the reusable components into aspects. Redundant code, which exists in many components and is almost the same, could be detected and encapsulated into aspects; thus, reducing the maintenance effort and possibility of introducing (copy-paste) errors.

An initial step toward the goal of adding new features as aspects on the code level has already been made with a development of an aspect-oriented extension of the Koala tools, a tool called INXS [13], which enables developers to, during system development, place advices afore and after the interface functions for debugging purposes. An industrially supported initiative, the IDEALS project [1], also aims to introduce aspects into development, and their results support findings that reducing redundant code in the system by using aspects has promise in a complex industrial setting.

However, a successful commercial application of aspects on the code level in a resource constrained environment is conditioned by the maturity of aspect weavers for programming languages used in an embedded domain, e.g., C language. So far, several "flavors" of aspect weavers for C language have been developed. These include AspectC++ [8], Aspicere [3], C4 [4] , WeaveC [5], and AspectC [2]. Often, each individual problem yields a new (distinct) weaver, and this existing variety of weavers indicates that technology for weaving C language needs to be matured for it to be used for developing industrial products.

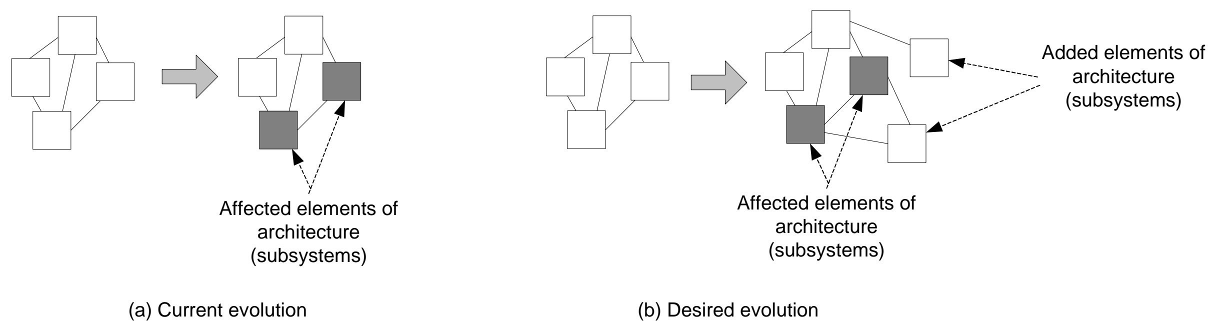

Figure 5: Evolution from the point of view of elements consisting the architecture

Putting it all together.

Given that, going from requirements to the implementation for a single feature, crosscutting occurs in several dimensions, it would be beneficial to have a support for the overall process, and all stages in that process, for embedded software. There is a great opportunity to use aspect-orientation also in tracing and visualizing crosscuttings from the requirements to the code as well as among the components in the system.

4. SUMMARY

In this paper we proposed several dimensions where aspects could ease the evolution problems in today's component-based product lines.

To date, to the best of our knowledge, there is no approach or a tool that would scale up to a commercial use of aspects for embedded consumer products, and that addresses all levels where aspect support could be used to facilitate adding aspects in existing commercial products.

5. REFERENCES

[1] Ideals project. Web page: http://www.embeddedsystems.nl/ site/frames.html?/site/projects/ideals/home.asp, January 2006.

[2] The AspectC project page. http://www.aspectc.net/, February 2007.

[3] The Aspicere project page. http://users.ugent.be/ badams/aspicere/, February 2007.

[4] The C4 project page. http://c4.cs.princeton.edu/, February 2007.

[5] The WeaveC project page. http://weavec.sourceforge.net/, February 2007.

[6] R. Chitchyan, A. Sampaio, A. Rashid, and P. Rayson. A tool suite for aspect-oriented requirements engineering. In *Proceedings of the 2006 International Workshop on Early Aspects at ICSE*, pages 19–26, New York, NY, USA, 2006. ACM Press.

[7] A. Rashid. Early aspects: A model for aspect-oriented requirements engineering. In *Proceedings of the IEEE Joint International Requirements Engineering Conference*, 2002.

[8] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: an aspect-oriented extension to C++. In *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'02)*, Sydney, Australia, February 2002. Australian Computer Society.

[9] A. Tešanović. *Developing Reusable and Reconfigurable Real-Time Software using Aspects and Components.* PhD thesis, Department of Computer and Information Science, Linköping University, Sweden, March 2006.

[10] A. Tešanović, M. Amirijoo, and J. Hansson. Providing configurable QoS management in real-time systems with QoS aspect packages. *Transactions on Aspect-Oriented Software Development II*, 4242:256–288, 2006.

[11] A. Tešanović and T. Trew. Open source in consumer products: Architectural consequences. Technical report, Philips Research Laboratories, November 2006.

[12] T. Trew. Enabling the smooth integration of core assets: Defining and packaging architectural rules for a family of embedded products. In *Proceedings of the 9th International Conference on Software Product Lines (SPLC'05)*, volume 3714 of *Lecture Notes in Computer Science*, pages 137–149. Springer, 2005.

[13] P. van der Laar. Inxs tool. Internal report, Philips Research, 2006.

[14] R. van Ommering. Mechanisms for handling diversity in a product population. In *Proceedings of the Fourth International Software Architecture Workshop*, June 2000.

[15] R. van Ommering. Building product populations with software components. In *Proceedings of the 24th International Conference on Software Engineering*, pages 255–265. ACM Press, 2002.

[16] R. van Ommering. *Building Product Populations with Software Components.* PhD thesis, Rijksuniversiteit Groningen, The Netherlands, 2004.

[17] C. Zhang, D. Gao, and H.-A. Jacobsen. Generic middleware substrate through modelware. In *Proceedings of the ACM/IFIP/USENIX 6th International Middleware Conference (Middleware'05)*, pages 314–333, Grenoble, France, 2005.

[18] C. Zhang, D. Gao, and H.-A. Jacobsen. Towards just-in-time middleware architectures. In *Proceedings of the 4th ACM International Conference on Aspect-Oriented Software Development (AOSD'05)*, pages 63–74. ACM Press, 2005.