Using Software Product Lines for Runtime Interoperability

Norbert Siegmund
University of Magdeburg
Germany
nsiegmun@ovgu.de

Mario Pukall
University of Magdeburg
Germany
pukall@ovgu.de

Michael Soffner
University of Magdeburg
Germany
soffner@ovgu.de

Veit Köppen
University of Magdeburg
Germany
koeppen@ovgu.de

Gunter Saake
University of Magdeburg
Germany
saake@ovgu.de

ABSTRACT

Today, often small, heterogeneous systems have to cooperate in order to fulfill a certain task. Interoperability between these systems is needed for their collaboration. However, achieving this interoperability raises several problems. For example, embedded systems might induce a higher probability for a system failure due to constrained power supply. Nevertheless, interoperability must be guaranteed even in scenarios where embedded systems are used. To overcome this problem, we use services to abstract the functionality from the system which realizes it. We outline how services can be generated using software product line techniques to bridge the heterogeneity of cooperating systems. Additionally, we address runtime changes of already deployed services to overcome system failures. In this paper, we show the runtime adaption process of these changes which includes the following two points. First, we outline why feature-oriented programming is appropriate in such scenarios. Second, we describe the runtime adaption process of services with feature-oriented programming.

Categories and Subject Descriptors

D.2.12 [**Software Engineering**]: Interoperability; D.3.3 [**Programming Languages**]: Language Constructs and Features —*Modules, packages*

General Terms

Design

Keywords

Software Product Lines, Runtime Adaption, Interoperability

1. INTRODUCTION

Nowadays, the number of complex systems which are built from a set of heterogeneous hardware increase rapidly. Such a complex system often use small devices to collect data and server systems to store the data. Examples are mechatronic systems in cars, large sensor networks for flood warning systems, and logistic hubs for cargo flow analysis. The system architecture consists of many different small, embedded systems that transfer and store required data. This raises problems for interoperability between these systems. Because of the different hardware of embedded devices, a developer has to cope with interaction problems that are reasoned of the diversity of different operating systems, communication protocols, and provided functionality.

In order to realize the interoperability of multiple systems, we have to design an interface which enables the communication between these systems. Our idea is to implement the interface as a service integrated in a service-oriented architecture whereas the architecture enforces the interoperability of the whole system. With this approach, we can abstract the functionality of a system from its realization. For example, consider a system that retrieves data via a service from another system that might be in one case an embedded system and in another case a software emulation of this device. The interface service contains different software modules where each of them implements a different functionality needed for the interoperability. Note, in both cases the service would run on a server and not on the device itself. However, developing a service for each system is time consuming and requires much effort. Today, already existing functionality is commonly reimplemented although there are often only few differences between embedded systems. A promising technique to overcome this problem is *software product lines (SPLs)*. The goal of an SPL is to generate a family of related products for a specific domain by reusing a common code base between these products. A user can configure an SPL to derive a tailor-made product which fits to her requirements. We argue that the interoperability is an important domain where SPLs can be applied. The variability in an SPL is required to address a large number of heterogeneous systems and to tailor the interfaces and communication protocols for the actual application scenario.

High availability is important for service-oriented architectures. Server systems can register services of other systems, e.g., sensors to store the sensed data. However, if the sensor fails, the server could be affected as well, because it might provide itself a service where data is aggregated from such a sensor. To keep the overall system stable, a promising solution might be a runtime adaption of the sensor service to

RAM-SE '09 July 7, 2009 Genova, Italy

replace the system behind, e.g., with a simulation program based on historical data.

We argue that software product line techniques and runtime adaption methods are promising to enforce the interoperability of many collaborating system based on a service-oriented architecture. In this paper, we outline our decision to use *feature-oriented programming (FOP)* [27, 6] as the suitable programming technique to implement product lines. Furthermore, we show the combination of FOP and how wrappers can be used to allow dynamic FOP.

1.1 The Logistic Hub Application Scenario

In the research project ViERforES[1], we observed the mentioned problems in several application scenarios. In this paper, we concentrate on the analysis of a logistic hub of an airport . We analyzed the required hardware and communication requirements. These are for example used to register incoming and outgoing goods. In Figure 1 we show a subset of the systems that are used in this domain. The hub uses sensors to retrieve the status of arriving and stocking goods and to maintain the cargo flow. Different sensors and different embedded systems have to communicate and cooperate with each other in order to visualize the hubs current state at a control center and different user interface points, e.g., at mobile devices. In such a scenario, we are confronted with a high dynamic environment. This results in frequent changes of participating systems that have to cooperate with each other. Since a logistic hub is a non-stoppable system, the architecture has to adopt new information sources or information consumers at runtime. Thus, we have to modify existing services while the system is running. Non-stoppable means, that we have to integrate new embedded devices (e.g., intelligent cargo boxes) while the system is running or automatically process a fail-over if a device is not available anymore. Changing each client separately would cause high maintenance costs and maybe down times. Therefore, we aim on changing the service itself instead all its clients.

2. BACKGROUND

A *software product line (SPL)* is a group of products sharing a common, managed set of features that satisfy specific needs of one domain and that are developed from a common set of core assets in a prescribed way [13, 22]. Different variants in an SPL are distinguished in terms of features, where *features* are distinguishable, end-user visible characteristics of a system or domain [14]. The overall goal of SPLs is to systematically reuse software artifacts for different software products. Commonly, a feature model is used to describe variability of an SPL [14]. Feature models are represented in a tree like form including boxes and connections to describe the features and relationships between them. Relationships might be mandatory or optional. Additionally, alternatives and OR relationships can be defined to introduce further variability into an SPL.

After the configuration of required features a program generator, e.g., *CIDE* [19] generates the resulting source code of selected features and compiles the composed files in order to retrieve the desired product.

2.1 Runtime Program Adaptation

[1]http://vierfores.de

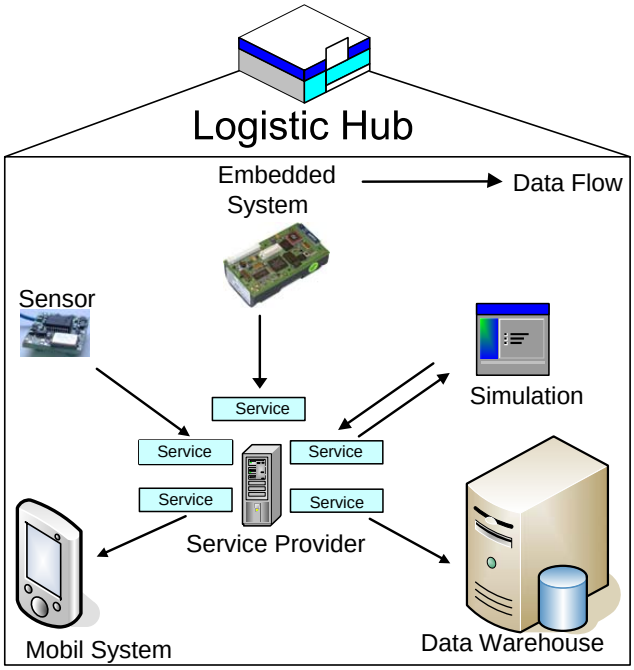

Figure 1: Services for system interoperability in a logistic hub.

Service-oriented architectures are commonly implemented with the Java programming language [35]. This is because Java's platform independence. We decided to use Java the implementation of the service-oriented architecture. However, Java has no runtime adaption functionality. For this reason, we developed an own approach to allow runtime adaption for Java programs. Changing the functionality of a running program is a multi-stage process. First, the classes which implement the new functionality must be identified. Second, the changes must be applied to these classes. Third, all object references which refer to one of the changed classes have to be updated in order to invoke the new or changed functionality. Unfortunately, this process is not supported by Java natively. In order to overcome this problem we developed our own runtime adaptation approach [28]. It satisfies the requirements resulting from the described process and enables Java programs for large-scale runtime changes.

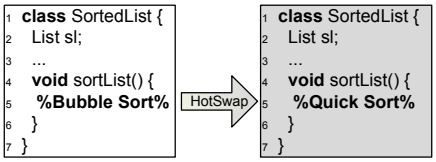

Figure 2: Class schema keeping changes.

Schema keeping class updates. One probable reason for changing a running program is to replace an algorithm such as depicted in Figure 2, where the actual sorting algorithm (*BubbleSort*) of class *SortedList* is replaced by the faster *QuickSort* algorithm. In order to process the algorithm replacement only requires to change the body of method *sortList()* which does not affect the class schema. We achieve such method body reimplementations using Java HotSwap

which is a feature of Sun's standard Java virtual machine called *HotSpot*[2].

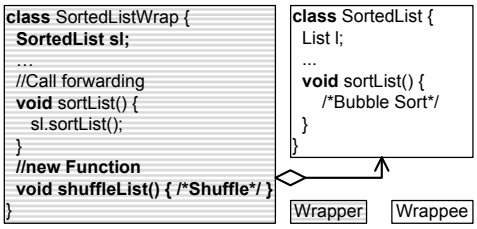

Figure 3: Class schema changing Updates.

Schema changing class updates. In many cases runtime program changes require more complex class updates then provided by Java HotSwap. Such a scenario is depicted in Figure 3 where method *shuffleList()* must be added to the program in order to undo the list sorting. We use object wrapping to apply new elements such as methods to the program. As shown in Figure 3 wrapper *SortedListWrap* adds the new method to the program whereas class *SortedList* continuously provides the sorting functionality.

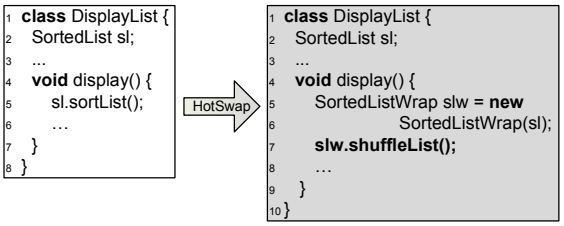

Figure 4: Object reference updates.

Object reference updates. In order to invoke the new or modified functionality provided by the wrapper all object references of the changed class have to be updated. This is also achieved via method reimplementations based on Java HotSwap. Considering our example we suppose that an instance of class *SortedList* is used by class *DisplayList*, see Figure 4. To be able to invoke the new added method *shuffleList()* method *display()* has to be reimplemented. Within the reimplementation an instance of wrapper *SortedListWrap* wraps *sl* of type *SortedList* (lines 5-6). Afterwards the wrapper instance is used to invoke the new functionality (line 7).

3. IMPLEMENTATION TECHNIQUES OF SPLS

As we describe in Section 2, SPLs are intended to reduce the development effort for a specific domain by reusing a common set of core assets, e.g., by reusing components. For this reason, we use an SPL to implement the services for many different systems. However, there are a number of possible techniques to implement an SPL. In this section, we outline our decision to chose feature-oriented programming to realize an SPL for service-oriented architectures.

3.1 Appropriate Techniques

[2]http://java.sun.com/javase/technologies/hotspot/

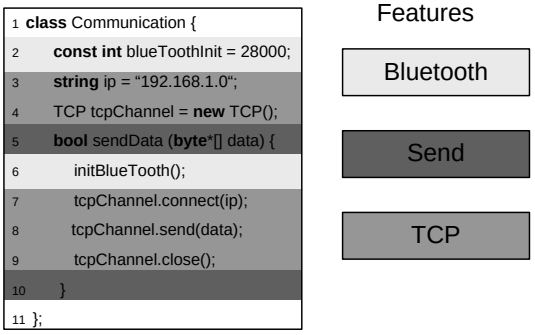

Figure 5: Excerpt of the communication class.

Possible programming approaches that can be used to implement SPLs ranges from new programming paradigms, e.g., *aspect-oriented programming (AOP)* [20], feature-oriented programming (FOP) [27, 6], preprocessor statements like *#IFDEFs* in C programming language, and component based approaches [15, 16, 32]. Note, that we have to consider that approach to be used has to support runtime adaption because it is required in the logistic hub scenario. Because preprocessor statements only allow static product generation, we do not consider it in our approach. The anatomy of runtime changes is an important factor for the decision of the appropriate technique. By analyzing the needs of the logistic hub, we figured out that fine-grained features are required for the configuration. This is due to the needed variability in this area. Furthermore, it is often the case that heterogeneous crosscuts arise during the feature implementation. Heterogeneous crosscuts means that different join points are applied with different code fragments. Homogeneous crosscuts which affect multiple join points with the same code fragment are very uncommon in our service-oriented architecture. We can use existing criteria to chose the appropriate programming technique [1, 3].

Components. Using components for runtime adaption has the advantage that the runtime adaption process is very easy to implement. Loading a component at runtime usually rises no serious problems because it encapsulates a well defined set of functionality. However, this encapsulated functionality is often coarse-grained (multiple features in one component). This is due to the fact that the implementation of crosscutting features is very challenging and in some cases not possible [25]. Additionally, a fine-grained implementation of components is not possible without a performance decrease [12]. In our case, the SPL requires a fine-grained decomposition of functionality, because it has to provide a large flexibility to support heterogeneous systems. Therefore, components are not suitable for our use case.

Using Aspects or Features Modules. Apel et al. analyzed the advantages and drawbacks of AOP and FOP [1, 3]. One result is that aspects should be used for homogeneous crosscuts and feature modules realized with FOP are suitable for heterogeneous crosscuts. Runtime changes using AOP,known as dynamic weaving, are demonstrated for example in [26] and dynamic feature reconfiguration was successfully applied in [29]. Hence, the choice of the appropriate programming technique depends on the features implementation. In Figure

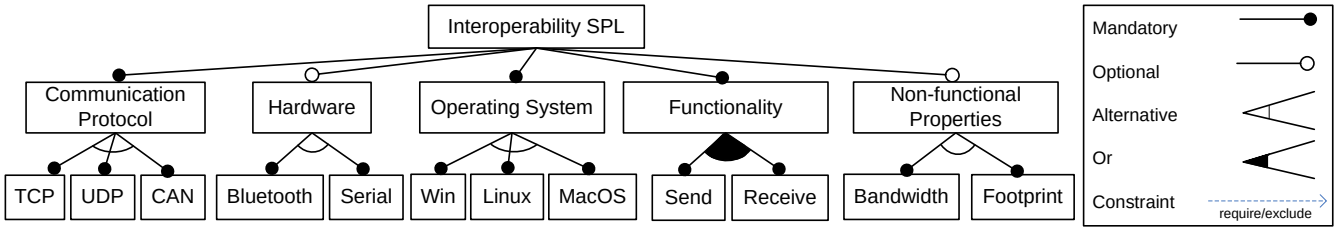

Figure 7: Feature diagram of the interoperability SPL.

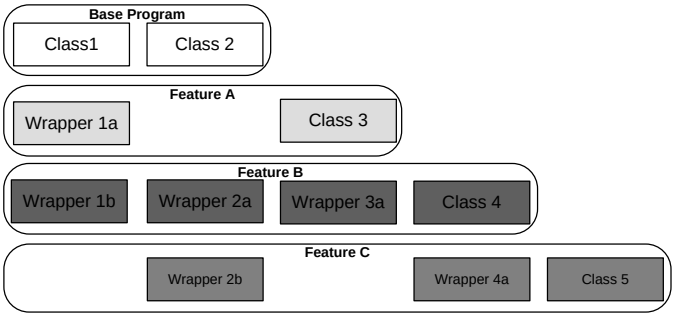

Figure 6: Feature-oriented runtime adaption architecture.

5, we depict an excerpt of the communication class of the service to be used which is composed from multiple features. The feature *Bluetooth* requires initialization code which is given in Line 2 and 6. If the feature *Send* is selected then the send method must be included in the communication class. Depending on the chosen communication protocol the method body looks different. In our example, the *TCP* protocol requires code to open communication (Line 7), send data (Line 8), and close communication (Line 9). Related to different features, this entanglement of code is very common in services. Because of these heterogeneous crosscuts of functionality in different classes, we decided to use feature-oriented programming.

3.2 Feature-oriented Runtime Adaption

FOP treats features of software as fundamental modules of abstraction and composition. It allows programmers to compose a family of similar programs based on domain specific features. These features are assembled together into *feature modules* which are commonly implemented as increments in program functionality [6]. To derive a program, a set of features is successively applied to a base program. FOP can be used as an extension of different programming paradigms. In this paper, we focus on FOP as an extension of object-oriented programming (OOP) using classes as implementation units. In this case, a feature is usually implemented by multiple collaborating classes. However, often only a fraction of a class belongs to a feature and the remaining part to other features. Consequently, the classes have to be decomposed with respect to the features of a software in order to generate classes that contain only desired functionality.

We combine the runtime adaptation approach shown in Section 2.1 with feature-oriented programming to allow a user to add, remove and exchange feature modules and at runtime. The basic elements of our feature-based runtime adaptation approach are classes and wrappers (see Figure 6). Whereby, classes build the base and wrappers act as refinements. Adding a feature at runtime requires to load all base classes and wrappers it introduces. To apply the wrappers the steps described in section 2.1 have to be processed. Feature removing requires to remove all wrappers/classes of the feature from the program. Feature exchange consists of two steps: removing of the obsolete feature and application the new one.

4. INTEROPERABILITY SOFTWARE PRODUCT LINE

Using FOP we are currently implementing the *Interoperability SPL*. Services that play the role of an interface between the server (where the service is located) and the data source (usually an embedded device). In Figure 7 we show an excerpt from our interoperability product line. The *Communication Protocol* feature defines the currently supported protocols. If a user selects a protocol, e.g., feature *TCP*, then we compose the code of the according protocol to generate a tailor-made service. During the lifetime of the many devices, it might be possible that new protocols are needed. Reasons are new devices that must be integrated in the logistic hub. This evolution of a software product line can also be supported by our wrapper approach, e.g., changes that cannot be foreseen when a service was generated can be applied to running services. In addition to the required communication protocols, the service has to consider operating system dependent code (feature *Operating System* in Fig. 7) as well as initialization methods for different hardware, e.g., bluetooth hardware.

The *Functionality* feature (see Fig. 7) implements the data access and data input to achieve an abstraction from the particular underlying system. While in our example the *Send* and *Receive* features carry out only a simple API, in real application scenarios these features are more precisely refined. However, this refinement is dependent upon the application. For example, in our logistic hub scenario, we have to define subfeatures for *Send* in order to express commands like discharge, charge, or transport of cargo. An example for feature *Receive* is the current status of critical cargo, e.g., cooled food. The feature *Non-functional Property* is especially important if the service is used for embedded systems. Because these devices have high resource restrictions, e.g., for processing power or bandwidth, we need to model code that improves the mentioned non-functional properties. In other words, we have to tailor the service to address the constraint hardware which might effect the communication speed, e.g., if data is too frequently queried from a wireless embedded device, this device is fast out of power supply. The measurement and configuration of non-functional properties

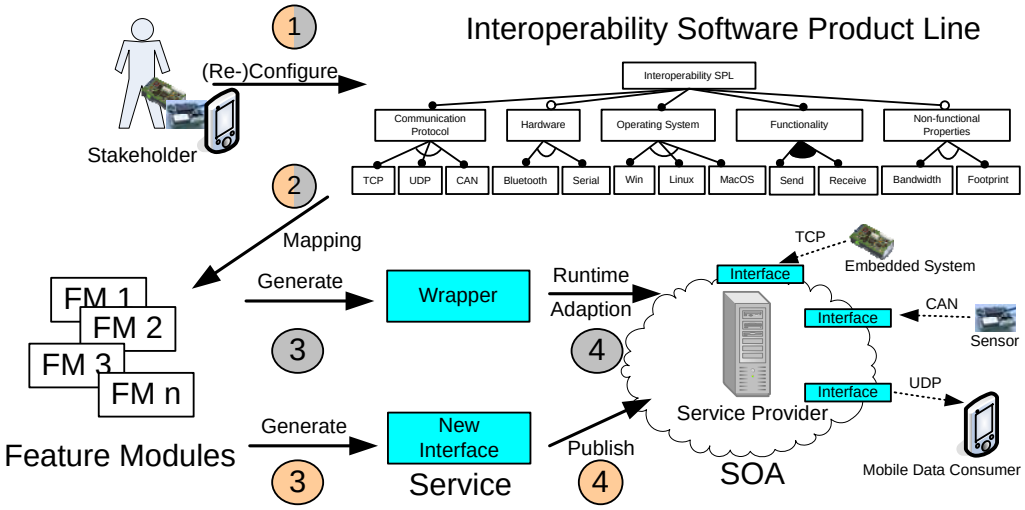

Figure 8: Overview of the architecture.

are beyond the scope of this paper but were investigated in [31].

4.1 Using the Interoperability SPL

In the following, we elaborate the whole architecture to present how the used techniques are combined to achieve the goals of enforcing the interoperability through runtime adaption in a continuously changing environment while reducing the development effort via software product lines. We show schematically the static derivation of a new service (displayed as orange steps) and the reconfiguration of an existing services (depicted as gray steps in Figure 8). Depending on the specification of the devices that are connected to the service provider, a stakeholder configures the interoperability product line (Step 1 in Fig. 8). In the second step, the configuration is mapped to the related feature modules that are composed together in Step 3 in order to generate the new tailored interface service. The service needs to be published using the service provider allowing the communication with the target devices.

Dynamic changes start also with Step 1 and 2. Changes in the environment enforce the reconfiguration of an existing service. All instances of used services can be stored with their feature model configuration. Changes of this configuration are mapped to feature modules which results in a list of deltas that represent new feature modules or commands to remove existing ones. Based on the describe wrapper approach in Section 3.2, wrappers are generated in Step 3 (gray color) and adapted at service provider in Step 4 (gray color) using the runtime adaption process stated in Section 2.1. With this technique the stability of the provided service oriented architecture allow the cooperation of heterogeneous systems that might fail during their lifetime. The reliability of the whole system is improved even when devices needs to be replaced. Using the software product line approach the whole runtime adaption process can be automated via the self (re-)configuration of an interoperability SPL.

5. RELATED WORK

Because of the advantages the ability for runtime program adaptations comes a long with, many approaches and tools to provide this ability have been developed in the recent past. This is particularly true for the Java language. Various approaches exist which solely use Java HotSwap for runtime adaptation purposes, e.g., *AspectWerkz* [10, 9], *Wool* [30], *PROSE* [23, 24], and *JAsCo* [34]. Like Java HotSwap itself, they do not allow to change the schema of already loaded classes.

However, researchers like Kniesel [21], Büchi [11], Hunt [18], and Bettini [7, 8] recommend object wrappings to modify a running program. What the approaches have in common is that the wrapping itself cannot be applied to the running program in an unanticipated way.

There are a number of languages and tools that support static as well as dynamic composition. CaesarJ [4] supports static composition based on collaborations and dynamic deployment of aspects. Object Teams [17] use dynamic composition of teams which represent features. Composition is possible by using statically instantiated activation teams which in turn activate other teams. Both approaches cannot handle unanticipated program changes (changes for what the application was not prepared before start) which is possible with our approach.

With FeatureC++ Rosenmüller et al. [29] implemented an approach to allow static and dynamic composition of features with the same code base. We use a similar combination of feature-oriented programming and runtime adaption. However, we can also change an already instantiated SPL which is currently not supported in FeatureC++.

Services and product lines are used in several studies in combination [33, 2]. For example, Trujillo et al. aim at combining multiple, separately developed product lines in a larger product line. A service oriented architecture is used to represent each single SPL. These approaches do not consider the automatic generation of services nor the runtime reconfiguration as we do. Runtime adaptable or context-aware services are introduced, e.g., by Bastida et al. [5]. The proposed context-aware services can be dynamically composed in order to provide an optimal service. In contrast to our approach, they do not consider software product lines as a basis for variable code. Additionally, the composition process cannot be done automatically which is possible with

our approach.

6. CONCLUSION

In this paper, we outlined an approach to generate services based on software product line technology. These services are used to enforce the interoperability of different systems by abstracting the functionality from the system behind the service. To enable this interoperability also in a running and changing environment, we have to generate these services on demand or adapt existing services at runtime. To enable the needed automatic on demand adaption, we argue that the software product line technology is suitable. We showed that feature-oriented programming is an appropriate implementation technique to implement product lines in our scenario. In future work, we will apply this technique with our already developed runtime adaption approach and currently develop a tool to enforce only correct runtime adaption. We plan to automate the whole service derivation process including the reconfiguration and runtime adaption.

Acknowledgments

Marko Rosenmüller, Norbert Siegmund, Michael Soffner and Veit Köppen are funded by the German Ministry of Education and Science (BMBF), project 01IM08003C. Mario Pukall is funded by German Research Foundation (DFG), project SA 465/31-2. The presented work is part of the ViERforES[3] and RAMSES project[4].

7. REFERENCES

[1] S. Apel, C. Kaestner, M. Kuhlemann, and T. Leich. Pointcuts, Advice, Refinements, and Collaborations: Similarities, Differences, and Synergies. *Innovations in Systems and Software Engineering (ISSE) – A NASA Journal*, 3(3-4), 2007.

[2] S. Apel, C. Kaestner, and C. Lengauer. Research challenges in the tension between features and services. In *Proc. ICSE Workshop on Systems Development in SOA Environments (SDSOA)*, pages 53–58, New York, NY, USA, May 2008. ACM.

[3] S. Apel, T. Leich, and G. Saake. Aspectual Feature Modules. In *IEEE Transactions on Software Engineering*, volume 34, pages 162–180. IEEE Computer Society, 2008.

[4] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. An Overview of CaesarJ. In *Transactions on Aspect-Oriented Software Development I*, volume 3880 of *Lecture Notes in Computer Science*, pages 135–173. Springer Verlag, 2006.

[5] L. Bastida, F. J. Nieto, and R. Tola. Context-aware service composition: a methodology and a case study. In *Proceedings of the International Workshop on Systems Development in SOA Environments*, pages 19–24. ACM, 2008.

[6] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering (TSE)*, 30(6):355–371, 2004.

[7] L. Bettini, S. Capecchi, and E. Giachino. Featherweight wrap java. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 1094–1100, New York, NY, USA, 2007. ACM.

[8] L. Bettini, S. Capecchi, and B. Venneri. Extending java to dynamic object behaviors. *Electr. Notes Theor. Comput. Sci.*, 82(7), 2003.

[9] J. Bonér. AspectWerkz – dynamic AOP for Java. *Invited talk at 3rd International Conference on Aspect-Oriented Software Development*, 2004.

[10] J. Bonér. What are the key issues for commercial AOP use: how does AspectWerkz address them? In *Proceedings of the International Conference on Aspect-Oriented Software Development*, 2004.

[11] M. Büchi and W. Weck. Generic wrappers. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP)*, pages 201–225, London, UK, 2000. Springer-Verlag.

[12] S. Chaudhuri and G. Weikum. Rethinking Database System Architecture: Towards a Self-Tuning RISC-Style Database System. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 1–10. Morgan Kaufmann, 2000.

[13] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[14] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[15] K. R. Dittrich and A. Geppert. Component Database Systems: Introduction, Foundations, and Overview. In *Component Database Systems*, pages 1–28. dpunkt.Verlag, 2001.

[16] A. Geppert, S. Scherrer, and K. R. Dittrich. KIDS: Construction of Database Management Systems based on Reuse. Technical Report ifi-97.01, Department of Computer Science. University of Zurich, 1997.

[17] S. Herrmann. Object Teams: Improving Modularity for Crosscutting Collaborations. In *Proceedings of the International Net.ObjectDays Conference*, volume 2591 of *Lecture Notes in Computer Science*, pages 248–264. Springer Verlag, 2002.

[18] J. M. Hunt and M. Sitaraman. Enhancements - enabling flexible feature and implementation selection. In *Proceedings of the International Conference on Software Reuse: Methods, Techniques and Tools*, volume 3107 of *Lecture Notes in Computer Science*, pages 86–100. Springer, 2004.

[19] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, pages 311–320, New York, NY, USA, May 2008. ACM.

[20] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Verlag, 1997.

[21] G. Kniesel. Type-safe delegation for run-time component adaptation. In *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP)*, pages 351–366, London, UK, 1999. Springer-Verlag.

[22] C. W. Krueger. New methods in software product line

[3] http://vierfores.de

[4] http://wwwiti.cs.uni-magdeburg.de/iti_db/forschung/ramses/

practice. *Commun. ACM*, 49(12):37–40, 2006.

[23] A. Nicoara, G. Alonso, and T. Roscoe. Controlled, systematic, and efficient code replacement for running java programs. *SIGOPS Operating Systems Review*, 42(4):233–246, 2008.

[24] A. Nicoară and G. Alonso. Dynamic aop with prose. In *Proceedings of International Workshop on Adaptive and Self-Managing Enterprise Applications*, pages 125–138, 2005.

[25] D. Nyström, A. Tešanović, M. Nolin, C. Norström, and J. Hansson. COMET: A Component-Based Real-Time Database for Automotive Systems. In *Proceedings of the Workshop on Software Engineering for Automotive Systems*, pages 1–8. IEEE Computer Society, 2004.

[26] A. Popovici, T. Gross, and G. Alonso. Dynamic Weaving for Aspect-Oriented Programming. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 141–147. ACM Press, 2002.

[27] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 419–443. Springer Verlag, 1997.

[28] M. Pukall, C. Kästner, and G. Saake. Towards unanticipated runtime adaptation of Java applications. In *Proceedings of the 15th Asia-Pacific Software Engineering Conference (APSEC)*, pages 85–92. IEEE Computer Society, Dec. 2008.

[29] M. Rosenmüller, N. Siegmund, S. Apel, and G. Saake. Code Generation to Support Static and Dynamic Composition of Software Product Lines. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering (GPCE)*, pages 3–12. ACM Press, Oct. 2008.

[30] Y. Sato, S. Chiba, and M. Tatsubori. A Selective, Just-in-Time Aspect Weaver. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, volume 2830 of *Lecture Notes in Computer Science*, pages 189–208. Springer Verlag, 2003.

[31] N. Siegmund, M. Rosenmüller, M. Kuhlemann, C. Kästner, and G. Saake. Measuring Non-functional Properties in Software Product Lines for Product Derivation. In *Proceedings of the 15th International Asia-Pacific Software Engineering Conference (APSEC)*, pages 187–194. IEEE Computer Society, Dec. 2008.

[32] M. Stonebraker and U. Cetintemel. One Size Fits All: An Idea Whose Time Has Come and Gone. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 2–11, 2005.

[33] S. Trujillo, C. Kästner, and S. Apel. Product Lines that supply other Product Lines: A Service-Oriented Approach. In *SPLC Workshop: Service-Oriented Architectures and Product Lines - What is the Connection?*, Sept. 2007.

[34] W. Vanderperren and D. Suvee. Optimizing JAsCo dynamic AOP through HotSwap and Jutta. In *Proceedings of the 1st AOSD Workshop on Dynamic Aspects*, pages 120–134, 2004.

[35] J. Waldo. The jini architecture for network-centric computing. *Commun. ACM*, 42(7):76–82, 1999.