Respecting Component Architecture to Migrate Product Copies to a Software Product Line

Benjamin Klatt
FZI Research Center for Information Technology
Haid-und-Neu-Str. 10-14
76131 Karlsruhe, Germany
klatt@fzi.de

Martin Küster
FZI Research Center for Information Technology
Haid-und-Neu-Str. 10-14
76131 Karlsruhe, Germany
kuester@fzi.de

ABSTRACT

Software product lines (SPL) are a well-known concept to efficiently develop product variants. However, migrating existing, customised product copies to a product line is still an open issue due to the required comprehension of differences among products and SPL design decisions. Most existing SPL approaches are focused on forward engineering. Only few aim to handle SPL evolution, but even those lack support of variability reverse engineering, which is necessary for migrating product copies to a product line. In this paper, we present how component architecture information can be used to enhance a variabilty reverse engineering process to target this challenge and show the relevance of component architecture in the individual requirements on the resulting SPL. We further provide an illustrating example to show how the concept is applied.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software, Reuse models; D.2.7 [**Software Engineering**]: Distribution, Maintenance and Enhancement, Restructuring, Reverse Engineering, and Reengineering

Keywords

Software Product Line, Reverse Engineering, Component Architecture

1. INTRODUCTION

Parnas et al. have discussed and criticised what they call the "Clone and Own" approach already in 1976 [20]. However, copy and customise existing products is still an approach often used in practice because of reasons, such as time and budget constraints for an initial delivery, or a new and evolving domains for which variability is not yet understood. To still benefit from advantages of the Software Product Line (SPL) concept (Clements et al.[7]), such as the managed reuse and variability, and faster instantiation of new product variants, the initial product copies need to be migrated to such a product line.

However, such a migration is a challenging task because of the required comprehension of product differences, the creation of explicit variability design decisions, and the mapping between the software's features, variability design, and implementation to enable further SPL evolution. While SPLs are of research interest for a long time and there are mature approaches in this area [12], [11], [23], most of them are focused on a forward engineering approach without a specific support for migrating existing product copies to a product line. In a similar way, reverse engineering techniques lack in the specific support to derive SPL variability from existing product copies. The few existing approaches targeting this specific challenge (e.g. [16],[8]) either do not consider any architecture information at all, or are not doing this in the context of a migration process and combined with other, reasonable techniques for analysis and difference comprehension, such as feature location or variation point clustering.

In this paper, we present how component architecture information can be used to improve the migration of product copies to a common product line. In the context of our proposed overall process, we recommend to consider such information i) to optimise the difference analysis between products, ii) to support the variability analysis such as variation point clustering, and iii) to take architecture information into account for the requirements on the resulting software product line. To illustrate the application of our approach and to show how architecture information can be considered, we give an example of two product copies and how they are merged.

The contribution of this paper is a concept of using component architecture to support i) difference analysis of product copies, ii) their variability analysis, and iii) product line design decision. We demonstrate how this concept can be applied to an example system.

The rest of this paper is structured as follows: Section 2 presents the required differentiation of features and variability. In Section 3 we introduce our proposed variability reverse engineering process and describe how this can take component architecture into account, followed by the process application example in Section 4, and our assumptions and known limitations in Section 5. Section 6 presents work related to our approach before we give a conclusion of the paper and an outlook on our future work in Section 7.

2. FEATURES AND VARIABILITY

The concept of features is well-known in the area of requirements engineering and the use of features to describe variability is well established. However, people often do not distinguish between variable features and variability in software design. Feature trees as proposed by Czarnecki et al. [9] are for example often used for all types of variability and different aspects such as requirements, software entities, or others are mixed in the same model. Similar to Bosch et al. [2], [21], we propose a clear differentiation between features, which are on a capability-level and related to requirements management, and variability of software in terms of variation points, which are on the level of software design and architecture. The following subsections explain this differentiation in more detail and introduce how we propose to partition them into different models [15].

2.1 Features

According to Bosch ”features are a logical unit of behaviour that is specified by a set of functional and quality requirements” ([2] p.194). This implies that features are on the same level as requirements and there is an m-to-n relationship between features and requirements. For example, an online banking feature requires a secure login and an account view. In addition, the requirement of a secure connection can be included in an online banking feature as well as in a car rental feature of a company's internal web portal. Due to this relationship, Svahnberg et al. described that a ”Feature typically manifests itself as a set of variation points and may be implemented as a set of collaborating components” ([21] p.7). Because features combine multiple requirements and may represent cross-cutting concerns of software, such as security, they can also be related to one or more variation points realising the variability of software.

2.2 Variability

Svahnberg et al. defined software variability as ”the ability of a software system or artefact to be efficiently extended, changed, customised or configured for use in a particular context” ([21] p.2). We use the same definition in our approach, because it relates variability to the design-level that describes the implementation of software. The variability in a software system is realised by specific variation points. A range of variability realisation techniques has been classified by Svahnberg et al. [21] according to characteristics, such as when, how, and by whom a specific variant is chosen for a variation point. Independent from the realisation technique chosen to implement a variation point, they have also identified that it is preferable to reduce the total number of variation points as much as possible to improve their manageability and the complexity of the software itself ([21] p.7). This awareness is relevant for the variation point design of our approach discussed in Section 3.3.3.

2.3 Model Structure

To reverse engineer the variability between two customised product copies, it is necessary to have representations of the existing software entities (e.g. components, classes, and methods) that can be analysed. This reperesentation needs to be referable by a variation point design as input for the downstream refactoring. The variation point design again needs to be linked with a feature definition necessary for the stakeholders as described above.

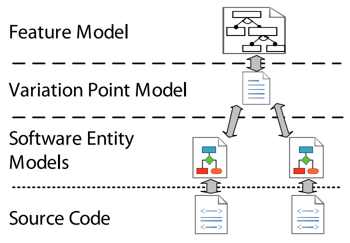

Figure 1: Structure of Feature, Variation Point, and Software Entity Models

Mature models exist for both ends, feature modelling (e.g. Kang et al. [12], or Czarnecki et al. [9]), and representation of software entities (e.g. abstract syntax trees as included in the OMG Knowledge Discovery Model [19], [18]). Due to this, we propose to use a specific variation point model, capable to link those mature models and include the design decision about the variation point realisation as described in [15] and illustrated in Figure 1.

Those three models are able to explicitly capture the required information, and make the necessary design decisions for migrating product copies to the intended product line. From our point of view, it is important to not mix the information of these models into a single one because of their different abstraction levels, purposes, and handling. The software entity models can be reverse engineered from the existing software, and need to be detailed enough for comparison and later refactoring support. Furthermore, the feature model is far more abstract and focused on those variable features relevant for stakeholders interested in the software capabilities, such as product managers. The later model is created and refined during our semi-automatic process as described in Section 3.

2.4 Software Product Line Profile

The approach of software product lines contains many generally applicable concepts, such as managed reuse and explicit variability. However, every realised product line has specific characteristics, such as its maturity level, realisation techniques, or architecture constraints. SPL maturity levels have been specified by Bosch et al. [3] and are related to when an open variability is bound to a specific variant, ranging from shared component repositories to multitenant systems. For different maturity levels, different realisation techniques can be used as surveyed in [21]. In addition, architecture constraints, such as sub-systems that need to be kept encapsulated due to organisational reasons, can be included as well.

We call such a set of requirements on the intended product line a ”Software Product Line Profile” (”SPL Profile”) and compare its relevance for the downstream process to those of architecture styles in the context of general architecture development.

3. VARIABILITY REVERSE ENGINEERING

Our approach is focused on scenarios in which product copies have been created, individually customised, and should now be migrated to a custom product line. To perform such a migration, we do neither assume to have any of the models described above already in place, nor do we require to have an architecture view provided with the implementation of

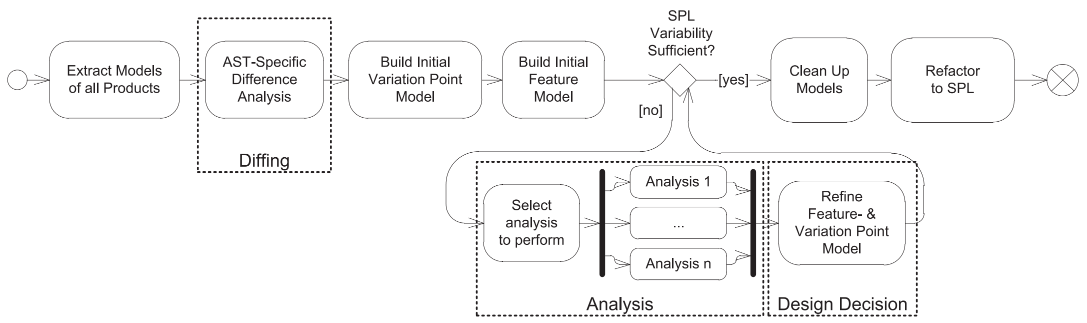
Extract Models of all Products
AST-Specific Difference Analysis
Diffing
Build Initial Variation Point Model
Build Initial Feature Model
SPL Variability Sufficient?
[yes]
[no]
Clean Up Models
Refactor to SPL
Select analysis to perform
Analysis 1
...
Analysis n
Analysis
Refine Feature- & Variation Point Model
Design Decision

Figure 2: Variability Reverse Engineering Process

the product copies.

Instead, we propose a process as shown in Figure 2. This process consists of a chain of activities starting with the extraction of abstract syntax tree (AST) models of the existing implementations. Those AST models are then compared. The next two steps build initial variation point and feature models derived from the identified differences. Those initial models are then iteratively refined to achieve a satisfying variability for the software product line. Each iteration includes one or more analyses individually selected for the iteration and executed in parallel. Their goal is to cluster identified variations, and to make recommendations which and how variations should be merged or modified. The recommended alternatives are the basis for making design decisions in the last step of an iteration. Finally, the models are cleaned up, e.g. merging the implementation models into a single one, and the required refactoring is performed.

The extraction, difference analysis, and initial model creation activities can be carried out fully automatically. The iterative refinement part of the process is semi-automatic and needs to involve a product line engineer. Because the analyses return recommendations in the form of reasonable variation point design alternatives, the engineer will be involved to decide which alternative he prefers. Whether the downstream refactoring can be done automatically, depends on the selected variation point realisation techniques and the individual project.

In addition to the general process, the dotted borders in Figure 2 highlight those activities that can take component architecture information into account and benefit from it. The following subsections describe this in more detail.

3.1 Considered Types of Architecture

We focus on structural architecture information, especially component architectures as a higher level view on object-oriented systems. We distinguish two types of architecture information to be considered: i) architecture of the product copies to be consolidated, and ii) architecture information included in the SPL profile. The former provides additional information to the implementation models extracted from the product copies. For example, this can be strongly related classes, explicit public interfaces, or software entities which belong to a commercial off-the-shell (COTS) component. The latter type of architecture information included in the SPL profile provides architecture information that influences the intended software product line. For example, the requirement to realise variation points with exchangeable components as extension mechanism for customers [14] would require clear, stable, and easy to use interfaces.

3.2 Sources of Architecture

The considered types of component architecture described above can result from different sources. First, we distinguish between explicit and implicit architecture. We call it explicit if it is represented in the implementation, such as package structures, or component descriptors, like OSGi bundle manifests, or web-service interface descriptions. In contrast, we call it implicit if the architecture is not directly visible, but for example described in an additional model or documentation taken into account during software implementation. Both of the sources are relevant for our approach, as long as they contain valid and especially up-to-date information.

According to our goal to also support cases with no or an outdated architecture model, we propose to use reverse engineering techniques to gain a component model of the existing implementation such as developed by Chouambe et al. [6]. Such reverse engineered models are up-to-date and valid by nature. They can include explicit and implicit information as long as it can be extracted or interpreted from the current implementation.

Alternative sources of architecture information are existing documentation and models. These have to fulfill the requirement to be up-to-date, and depending on their application, they might need to be formalised.

Beside those two types of sources, which are focused on the architecture of the product copies under study, the third source are architecture requirements described in the SPL profile. As mentioned before, this profile contains a style guide for the intended product line realisation and might contain architecture constraints to be considered.

All of this architecture information is considered by one or more of the previously highlighted process steps. The following subsections provide deeper insight in which steps which architecture information can be used and how this is done.

3.3 Activity-Specific Influence

Independent from the source of the architecture information, the diffing, analysis, and design decision activities, highlighted in Figure 2, are influenced by the architecture information as described in the following subsections. One might admit that the downstream refactoring is also related to architecture. This is right, but from our perspective, this is a different type of architecture relation, because at this point in the process, the engineer is only implementing architecture decisions already made during former activities and due to this, it is not in the focus of this paper.

3.3.1 *Diffing*

In this activity, abstract syntax trees are compared to identify differences between the product copies, which can be complex and time consuming task for large software systems. A two-phase model-diffing approach, as proposed by Xing et al. [25] is a reasonable technique for this: First, elements which represent the same software entity are matched in the "matching phase". Second, elements without a match and those with modified attributes or children, are identified for each model under study during the "diffing phase". In this step, the analysis can benefit from using architecture information. For example, the diffing phase can be optimised by reducing the size of the search window for finding matching elements, to not cross the border of the component the searched element is located in.

Furthermore, the component structure itself could be analysed for differences between the product copies. If it is intended to realise the variability of the resulting software product line with exchangeable components only, it might be reasonable to perform the differencing on the component level only. However, such a differencing of the component architectures, requires to have the architecture of a comparable (sub-)system of both implementations, to ensure the differencing makes any sense at all.

3.3.2 *Analyses*

The aim of the analyses is to provide recommendations on which variation points to merge or refine. The most relevant and obvious support of architecture for such analyses are the additional clustering capabilities based on the higher-order structure of the implementations. As for the diffing, the intended realisation techniques of the SPL profile might influence the selection of analyses to be perfomed, e.g. due to a different granularity requirement. In addition, product-specific architecture constraints could be taken into account for the recommendations given by the analysis to ensure they will not be broken.

Beside the classes and subcomponents of a component, also its interfaces can be taken into account. For example, if a variation point should be realised with an extension mechanism the interface to the exchangeable components is subject to additional constraints [14]: If multiple alternatives for an interface are offered, the most stable one with the lowest complexity should be preferred.

The architecture information used in this activity has to fulfill slightly different requirements than for the difference analysis. While the diffing requires to have architectures of both product copies, for the analyses it can be sufficient to have a component architecture only for one product copy. For example, if the customisations of one product should be merged into the other, which we call the leading product, it might be enough to have the architecture of this leading product to map the variation points to.

3.3.3 *Design Decision*

The last step influenced by the architecture is the design decision activity. In this step, the SPL engineer selects his preferred variation point design alternatives and decides how they should be realised. The selection is done based on the results of the preceding analyses. Due to this, the activity is indirectly influenced by the architecture information that has influenced the analyses and the resulting recommendations. In addition, the product line engineer can take additional SPL profile requirements or architecture information, not considered by the analyses into account to make a more educated design decision.

4. CALCULATOR EXAMPLE

We have developed an example to illustrate and describe how the approach is applied in general, and component architectures can be taken into account in particular. The software is publicly available for download at [13].

As described in Section 3, the requirements on the SPL, such as the intended variability realisation technique, have a high impact on multiple activities of the variability reverse engineering process. For this, we assume the intended SPL should be realised using exchangeable components bound by dependency injection at system assembly time.

4.1 System Under Study

The example system is a software, which is able to calculate the greatest common divisor of a given number. The case study includes two implementations of the calculator: i) a variant that uses the standard Java API only with the class "BigInteger" available as part of the java.math API, and ii) a variant that makes use of the open source JScience library [10]. As shown in Figure 3 a), the standard Java implementation does not require any additional components apart from the Java API itself. Due to this, it requires less storage (the jscience.jar archive file is about ˜665kb) which might be relevant, e.g. for the deployment on mobile devices. In contrast, the JScience-based implementation requires the additional Java archive file containing the JScience component as shown in Figure 3 b), but according to the JScience website [10], it offers improved quality attributes (e.g. multi-core processor concurrency support, and reduced garbage collection).

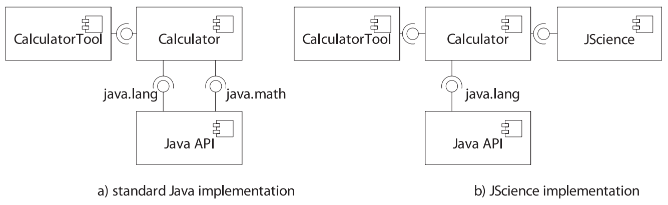

Figure 3: Component Models of Calculator Variants

The differences of the two implementations are located in the imports, the method `gcd()` of the class `Calculator` as shown in the code excerpts below. The JavaDoc comments differ as well but are excluded from this case study.

Listing 1: Standard Java Implementation

```
import java.math.BigInteger;
...
public String gcd(String v1, String v2){
 BigInteger intV1 = new BigInteger(v1);
 BigInteger intV2 = new BigInteger(v2);
 BigInteger gcd = intV1.gcd(intV2);
 return gcd.toString();
}
```

Listing 2: JScience-Based Implementation

```
import org.jscience.mathematics.number.LargeInteger;
...
public String gcd(String v1, String v2){
 LargeInteger intV1 = LargeInteger.valueOf(v1);
 LargeInteger intV2 = LargeInteger.valueOf(v2);
 LargeInteger gcd = intV1.gcd(intV2);
 return gcd.toString();
}
```

4.2 Applying the Approach

As a first step, the abstract syntax tree (AST) models of the product copies are extracted and their difference analysis is performed. The Java packages are interpreted as a component structure and the difference analysis is optimised to search matches of the customised lines in the same package only. Due to the requirement of realising the product line with exchangeable components, it would be reasonable to present the three differing statements not as individual differences but the whole class as an entity that differs in the two product copies. However, we ignore this optimisation in this example to be able to illustrate the subsequent activities.

After the diffing, we build up initial variation point and feature models. Each of them contains three variation points, respectively variable features, because of the three differing statements as the finest granularity of differences. Three variation point elements are linked with the software entity element representing the gcd() method because it is the parent element of all three statements. Each of these variation points contain two ”variant” elements which are linked to the alternative implementations of a statement. In this example, there are also three initial variable features, each linked with one variation point and the alternative features are linked with ”variant” elements.

Next, in the analysis step, we know that we want to achieve an SPL based on exchangeable components. Furthermore, we know that the JScience library is a COTS component that we have to keep separated. Without this component information, it could happen that the system recommends to move the class LargeInteger, which is part of JScience, to the variant component and completely ignore the rest of JScience, but this is prohibited. However, the analysis will return, that all variation points belong to the same component represented by the package org.splevo.examples.calculator and recommends to make the class `Calculator` an interface with the abstract method `gcd()`. Furthermore, the JScience and standard Java-specific implementations of the method are moved to individual components which can be deployed separately.

So in the design decision activity, we decide to merge the individual variation points of the three varying statements into one variation point linked to the gcd() method as a clear interface and two ”variant” elements which are linked to separate components implementing this interface based on JScience respectively standard Java only. At this point of the process, the models of the software will be cleaned up by removing obsolete elements and adapting the class `CalculatorTool` to be able to inject the desired calculator interface implementations into it. As a last step, the code is refactored according to this new product line design.

As shown, the approach can be applied straightforward and it benefits from the guidance achieved by taking the component architecture into account.

5. ASSUMPTIONS AND LIMITATIONS

Our approach is focused on product copies derived from a common code base. We assume individually developed products to have too many differences to find enough matches to be merged. We assume the deviation between the products under study as the major limiting factor for our approach. There may be product copies that are too different to be merged but also individually developed products based on common infrastructures which can benefit from using our approach.

Furthermore, the algorithms potentially used for the diffing and analysis activities can have a high complexity, such as graph matching. To target this challenge, we propose to use optimisations, such as the component-oriented search window reduction described in Section 3.3.1.

Our use of architecture is focused on structural information that can be unambigously mapped to the existing implementations. However, additional information, such as deployment or allocation specifications can always be represented as constraints in the SPL profile but may need to be considered manually by the SPL engineer.

Finally, our approach identifies only variable features which are present in the product copies and differ between them. There will be additional mandatory features from requirements engineering or product management point of view, but their identification is not part of our research.

6. RELATED WORK

As identified by Chen et al. in their survey on variability management approaches [5], only few approaches on variability management handle evolutionary aspects such as Loesch [17], or Deursen [22]. Alves et al. [1] have developed an additional approach on variability-aware software product line refactoring. However, even if Alves et al. have identified the necessaty of identifying the variability in existing software entities and between product-copies, none of these approaches provide any support for it.

Our work focuses on the creation of SPLs from customised product copies with a common initial code base, comparable to the approach by Koschke et al. [16]. However, our approach does not require a pre-existing module architecture as the approach by Koschke et al. does to apply the extended reflexion method. Furthermore, they do not take custom SPL requirements into account, such as the intended maturity level, and the prefered variation point design and realisation technique, which we claim to have an impact on the analysis to be performed.

Cornelissen et al. [8] have presented an approach for product consolidation using feature location techniques based on program graph analysis as done by Rajlich et al. [4], and Wilde et al. [24] to identify varying features to consolidate. They do not take any architecture information into account. However, their approach is complementary to ours and from our point of view it is an analysis to be considered for clustering variability and migrating to a mature product line.

7. CONCLUSION AND OUTLOOK

In this paper, we presented how architecture information can be considered for migrating customised product copies to a software product line. We have introduced our proposed process based on a clear separation of features and variation points and identified which activities can benefit from what type of architecture and the potential information sources.

Finally, we discussed an exemplary process application to illustrate how an SPL engineer responsible for consolidating the product copies, would be supported and guided by our approach.

Our future work will focus on identifying publicly available reference systems that could be used as benchmarks for our approach and by other researchers in this area. We further investigate the automation of our approach to apply the migration in case studies on real systems and to test several differencing and analysis algorithms. Furthermore, we want to prove our approach for raising the maturity level of existing product lines (e.g. migrating configurable products to multi-tenant systems with user-aware feature selection).

8. REFERENCES

[1] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, and C. Lucena. Refactoring Product Lines. In *Proceedings of the 5th international conference on Generative programming and component engineering*, pages 201–210. ACM, 2006.

[2] J. Bosch. *Design and use of software architectures: adopting and evolving a product-line approach*, volume 0. Addison-Wesley Professional, 2000.

[3] J. Bosch. Maturity and Evolution in Software Product Lines: Approaches, Artefacts and Organization. In G. Chastek, editor, *Software Product Lines*, volume 2379 of *Lecture Notes in Computer Science*, pages 247–262. Springer Berlin / Heidelberg, 2002.

[4] K. Chen and V. Rajlich. Case study of feature location using dependence graph. In *8th International Workshop on Program Comprehension. IWPC 2000.*, pages 241–247, Limerick , Ireland, 2000. IEEE.

[5] L. Chen, M. Ali Babar, and N. Ali. Variability management in software product lines: a systematic review. In *Proceedings of the 13th International Software Product Line Conference*, pages 81–90. Carnegie Mellon University, 2009.

[6] L. Chouambe, B. Klatt, and K. Krogmann. Reverse Engineering Software-Models of Component-Based Systems. In *Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering*, pages 93–102, Washington, DC, USA, 2008. IEEE Computer Society.

[7] L. Clements Paul ; Northrop. *Software product lines : practices and patterns.* SEI series in software engineering. Addison-Wesley, Boston, Mass., 6. print. edition, 2007.

[8] B. Cornelissen, B. Graaf, and L. Moonen. Identification of variation points using dynamic analysis. In *1st international workshop on reengineering towards product lines (R2PL'2005)*, pages 9–13, 2005.

[9] K. Czarnecki, K. Ø sterbye, and M. Völter. Generative Programming. In J. Hernandez and A. Moreira, editors, *Object-Oriented Technology ECOOP 2002 Workshop Reader*, volume 2548 of *Lecture Notes in Computer Science*, pages 15–29. Springer Berlin / Heidelberg, 2002.

[10] J.-M. Dautelle. JScience, 2012. http://www.jscience.org.

[11] S. Deelstra, M. Sinnema, J. Nijhuis, and J. Bosch. COSVAM: a technique for assessing software variability in software product families. *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pages 458–462, 2004.

[12] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report November, Software Engineering Institute - Carnegie Mellon University, Pittsburgh, Pennsylvania, 1990.

[13] B. Klatt. GCD Calculator Example, 2012. http://sdqweb.ipd.kit.edu/wiki/GCD_Calculator_Example.

[14] B. Klatt and K. Krogmann. Software Extension Mechanisms. In R. Reussner, C. Szyperski, and W. Weck, editors, *Proceedings of the Thirteenth International Workshop on Component-Oriented Programming (WCOP'08), Karlsruhe, Germany*, number 2008-12, pages 11–18, 2008.

[15] B. Klatt and K. Krogmann. Model-Driven Product Consolidation into Software Product Lines. In *Workshop on Model-Driven and Model-Based Software Modernzation (MMSM'2012)*, Bamberg, 2012.

[16] R. Koschke, P. Frenzel, A. P. J. Breu, and K. Angstmann. Extending the reflexion method for consolidating software variants into product lines. *Software Quality Journal*, 17(4):331–366, Mar. 2009.

[17] F. Loesch and E. Ploedereder. Optimization of Variability in Software Product Lines. *11th International Software Product Line Conference (SPLC 2007)*, pages 151–162, Sept. 2007.

[18] OMG. Architecture-driven Modernization : Abstract Syntax Tree Metamodel (ASTM). Technical Report January, OMG, 2011.

[19] OMG. Architecture-Driven Modernization : Knowledge Discovery Meta-Model (KDM). Technical Report August, OMG, 2011.

[20] D. L. Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, SE-2(1):1–9, 1976.

[21] M. Svahnberg, J. van Gurp, and J. Bosch. A taxonomy of variability realization techniques. *Software: Practice and Experience*, 35(8):705–754, 2005.

[22] A. Van Deursen, M. De Jonge, and T. Kuipers. Feature-based product line instantiation using source-level packages. In *Software Product Lines Conference 2002*, pages 19–30. Springer, 2002.

[23] D. L. Webber and H. Gomaa. Modeling variability in software product lines with the variation point model. *Science of Computer Programming*, 53(3):305–331, 2004.

[24] N. Wilde and M. C. Scully. Software reconnaissance: Mapping program features to code. *Journal Of Software Maintenance Research And Practice*, 7(1):49–62, 1995.

[25] Z. Xing and E. Stroulia. UMLDiff: an algorithm for object-oriented design differencing. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 54–65. ACM, 2005.