Improving Product Copy Consolidation by Architecture-aware Difference Analysis

Benjamin Klatt, Martin Küster
FZI Research Center for Information Technology
Haid-und-Neu-Str. 10-14, Karlsruhe, Germany
{klatt,kuester}@fzi.de

ABSTRACT

Software product lines (SPL) are a well-known concept to efficiently develop product variants. However, migrating customised product copies to a product line is still a labour-intensive challenge due to the required comprehension of differences among the implementations 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 our continued concept on using component architecture information to enhance a variability reverse engineering process. Including this information particularly improves the difference identification as well as the variation point analysis and -aggregation steps. We show how the concept can be applied by providing an illustrating example.

1. INTRODUCTION

Parnas et al. have discussed and criticised code duplication what they call the "Clone and Own" approach already in 1976 [13]. 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 domain for which variability is not yet understood. To still benefit from advantages of the Software Product Line (SPL) concept (Clements et al. [4]), such as the managed reuse and variability, and faster instantiation of new product variants, the initial, customised product copies need to be consolidated to a product line afterwards.

However, such a consolidation is a challenging task because of the required identification and assessment of relevant product differences between the copies, the creation of a reasonable software variability design, and the refactoring of the copies into a single product line. Especially the potentially high number of differences, not all relevant for the resulting product line, makes such a consolidation hard to be done manually.

While SPLs are of research interest for a long time and mature approaches exist in this area [8], [7], [16], most of them focus on forward engineering without specific support for migrating existing product copies to a product line. In a similar way, reverse engineering techniques lack in support specific to derive variability from existing product copies.

The few existing approaches targeting this specific challenge (e.g., [11],[5]) either do not provide support to reduce the manual effort to handle the product differences or make strong assumptions by handling features individually and ignoring relationships between several features.

We have developed an approach to recommend variation point design decisions to a product line engineer, enabling him to derive a product line from customised product copies with reduced manual effort. Our approach facilitates reverse engineering techniques to automatically extract models from the implementations, a model comparison to identify differences, and a variability relationship analysis to derive recommendations for the variation point design.

In this paper, we discuss how software architecture information can be used to improve the overall approach. In the context of our proposed 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 aggregation, and iii) to specify additional requirements on the resulting software product line.

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

In addition to our previous publication [9], that discussed the general concept of using architecture in this process, in this extended paper, we provide details about our developed variation point model, our differencing, and our analysis processes, which are now available in a prototype and benefit the most from utilising architecture information. We also applied this prototype to an extended version of the example already discussed in our previous publication.

The rest of this paper is structured as follows: Section 2 presents our Variation Point Model. Section 3 provides an overview of our overarching SPLevo approach, before discussing the influence of architecture information on this approach in Section 4 and providing an illustrating example of its application in Section 5. Work related to our approach is presented in Section 6 before we give a conclusion of the paper and an outlook on our future work in Section 7.

2. VARIATION POINT MODEL (VPM)

The concept of features is well-known in the area of requirements engineering and widely used to describe variability. However, people often do not distinguish between variable features and variability in software design. In our approach, we clearly distinguish between variable features on a product management and variation points on a software design level as done by Svahnberg et al. [15].

To express the variation points of the product copies under study, respectively those to be present in the resulting product line, we have designed a Variation Point Model (VPM). It specifies Variation Points (VP) describing the location of a variability by linking nodes of an abstract syntax tree model (AST). In addition, VPs contain one or more Variant elements linked with AST nodes representing a variability alternative. VPs are contained in VariationPointGroups to support variation point aggregations as required during the variation point design process (see Section 3.1.2).

In the following section we describe how such a Variation Point Model is initialised and refined during our variability reverse engineering process.

3. SPLEVO APPROACH OVERVIEW

Our SPLevo approach focuses on scenarios with products copied, customised, and now to be migrated to a custom product line. To perform such a consolidation, we do not assume to have any product line or variability-specific information already in place, nor do we require to have any other information or documentation provided with the implementation of the product copies in general. However, our approach benefits from the use of additional architecture information which is this paper's focus.

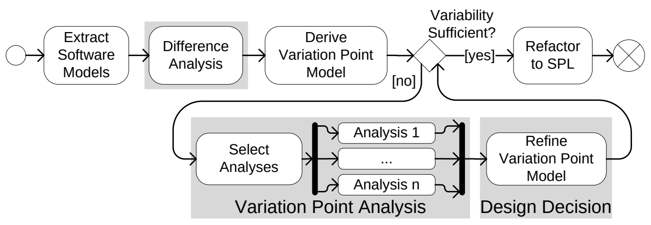

Figure 1: Iterative SPLevo Main Process (architecture-influenced activities marked in grey)

As shown in Figure 1, our SPLevo process consists of a chain of activities starting with the extraction of abstract syntax tree (AST) models from the implementations. Those AST models are then compared to identify differences between the implementations. The next step derives an initial VPM from these differences. The VPM is then iteratively refined to achieve a satisfying variation point design for the SPL. The goal of the analysis is to identify related variation points, and to recommend aggregations to derive an improved variation point design. The recommendations are used to make design decisions in the last step of an iteration. In this step, the product line engineer can decide to accept, decline or modify the recommended aggregations. Finally, the models are cleaned up and the required refactoring is performed.

The extraction, difference analysis, and initial variation point model creation activities can be carried out fully automatically. The iterative refinement part of the process is semi-automatic. A consolidation process will always require manual decisions because of personal preferences on equivalent alternatives in a typically large design space (e.g., because of available alternatives for implementation techniques, code structurings, or feature slicings) or organisational issues. To support this, the analyses return recommendations in the form of reasonable variation point design refinements the product line engineer can decide about.

Whether the downstream refactoring can be done automatically, depends on the selected variability realisation techniques and the individual project.

3.1 Variation Point Analysis

The result of the variation point analysis is a VPM with a satisfying variation point design. One of the product copies can serve as the basis for the product line, integrating the features from the other copies according to the variation point design.

As described in the previous section, an initial VPM is derived from the fine-grained differences between the software entity models. This is done by creating a separate *VariationPoint* element (VP) for each difference. The differing ASTNodes are referenced as software entities of *Variant* elements and their parent ASTNode is referenced as the variation point's location in the software.

This initial Variation Point Model represents all fine-grained differences at the source code level. To achieve a manageable amount of variability in the resulting product line, the VPM must be refined. Manually analysing all variation points to decide about their aggregation requires a lot of effort due to the typically high number of differences.

To support this task, we aim to provide a tool supported analysis to identify related variation points and to derive aggregations. In general, product consolidations cannot be done in a fully automatic fashion. Equivalent alternatives are possible and selected due to non-technical criteria, such as personal preferences or organisational reasons. To cope with this, our analysis returns only recommendations that the product line engineer can accept, decline, or adapt.

As we describe in [10], we use a graph-based approach to combine several basic relationship analyses to derive different types of variation point aggregations. To make this paper self-contained, the following subsections provide an overview of the concept and describe the aggregation and filtering techniques as possible variation point refinements to better understand the discussion of the architecture influence on the process in Section 4.

3.1.1 Graph-Based Analyses Composition

Relationships between variation points can exist due to many different aspects, such as their location or type of modification. To enable our analysis, we consider this scenario as an edge-labeled, undirected graph with the variation points as vertices, and their relationships as edges. The type of a relationship is stored as a label of the according edge (e.g., "CL" because the relationship is derived from the code locations). Multiple different relationships can exist between the same variation points, each represented as an edge between their nodes. In addition, sub-labels can be defined to provide additional information about the relationships (e.g., the name of the method two variation points are located in).

As shown in Figure 2, the VPM is transformed into a graph representation by creating a node for each variation

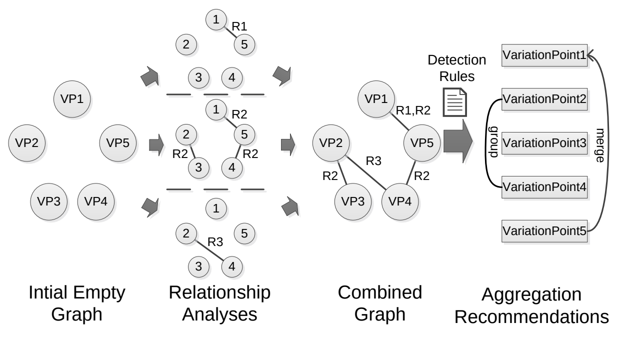

Figure 2: Graph-Based Analyses Composition

point. Next, this graph is handed over to all analyses selected for a specific iteration and performed in parallel. Each analysis creates edges corresponding to the relationships it has identified. Those edges are labelled with the type of relationship identified, e.g., R1, R2, and R3. In the next step, the edges returned from the individual analyses are merged into a single graph.

In the final step of the analysis process, the recommendations to refine the VPM are derived by detection rules inspecting the relationship type combinations between variation points. A detection rule is specified for a set of edge labels as its matching criteria. In addition, it specifies a refinement to recommend in case of a match. If a detection rule matches a sub-graph of variation points, a refinement recommendation is created for all involved variation points according to the rule's specification. This refinement is stored in an overall recommendation set to be presented to the engineer.

Detection rules are always applied in a defined order and if edges are matched by one rule, any rules applied later take the already recommended refinement into account. The set of rules to apply as well as their order depends on the individual product line requirements and need to be aligned to the set of analyses performed in a specific iteration and the consolidation scenario under study. The recommended variation point aggregations are described in the following subsection.

3.1.2 Variation Point Aggregation

A good variation point design is a trade-off between providing variability for as much product options as possible while minimising the number of variabilities to manage.

In our SPLevo approach, we start with a fine-grained VPM on the level of identified code differences and use group and merge aggregations to get to a more coarse-grained manageable one.

VariationPoints are merged by consolidating their Variant elements and the referenced software entities in only one of the *VariationPoint* elements.

The variation point grouping is based on the explicit group structure in the VPM as described in Section 2. Even in the initial VPM, each *VariationPoint* is assigned to a *VariationPointGroup*.

3.1.3 Variation Point Filtering

Complementary to the previously described aggregations, variation points can be filtered from the VPM. This can be done to remove identified variation points which are relevant for the intended product line, such as variation points located in a component which is out of scope. In such a case, the variation point is simply removed from the VPM and will no longer lead to a refactoring advice in the downstream process.

4. ARCHITECTURE INFLUENCE

Figure 1 shows not only the activities of our general process, but also highlights those that benefit from taking architecture information into account (grey areas). We focus on component architectures as higher level structures of object-oriented systems. This information has to be up-to-date and should be formalised to be processed. Otherwise, it could be used only manually by the engineer accepting or declining design recommendations. The following subsections describe the architecture's influence per activity.

4.1 Diffing

The diffing activity compares the extracted AST models to identify differences between the product copies. Typically, this is a complex and time consuming task for large software systems if done manually. A two-phase model-diffing approach, as proposed by Xing et al. [17] can be applied to automate this: The *matching phase*, matches elements representing the same software entity. The *diffing phase*, handles elements without a match and identifies modified attributes, references, or children of the others. In both phases, the architecture information itself can be analysed and used to optimise the diffing phases for better results and reduced processing complexity.

If the variability should be realised in the resulting software product line with exchangeable components, it might be reasonable to perform the differencing on the component level only. This allows to ignore the detailed differences on lower implementation levels and to mark only the components as added, removed, or changed if they contain any changed elements. It is also possible to combine various levels of difference abstraction depending on the degree of differences within a component. If the differences can not be abstracted to the component level completely and a more fine-grained diffing is required, the diffing phases split up in the following sequential steps:

Match Elements connects elements representing the same software entity. This is done by comparing an element with matching candidates. By default, all software elements of the same type are compared. Taking component borders into account, the list can be reduced significantly in a first run by comparing only elements in the same component (e.g., classes realising a component). If no match was found, a second run can try to match elements in other components to not miss elements moved between components. However, moving elements between components is assumed to be a proportionally rare operation when customising product copies.

Check Attributes and **Check References** identify matched elements with changed attribute values respectively references. These checks can be completely skipped for elements or references to elements of components that are out of scope (e.g., third party components).

Process Unmatched Elements handles elements present in one copy only. They are either filtered or converted into more reasonable diffing types such as *Import Added* or *Method Added* instead of a generic *Element Added*. Architecture information can define scopes of irrelevant elements to filter and allows for identifying additional difference types (i.e. *Component Added* and *Component Deleted*).

Similar to the previous step, **Derive Specific Difference Types** produces more reasonable diffing elements but not only for added or removed elements, but for any type of difference. If the product line engineer prefers to work on the component level only, all differences could be encapsulated by *Component Change* elements.

4.2 Variation Point Analyses

The aim of the overall variation point analysis is to recommend variation points to be aggregated. The most relevant and obvious architecture-based support for such an analysis is the additional clustering capability based on the higher-order structures of the implementations. As for the diffing, the intended variability realisation techniques might influence the selection of analyses to be performed (e.g., variation points can only be realised on a class or component level). In addition, product-specific architecture constraints could be taken into account by the analysis to not recommend refinements invalidating these constraints.

For our approach of combining several basic relationship analyses, as presented in Section 3.1, we have identified several types of relationships between variation points and according strategies to find them. In the following, we discuss, how each of these relationship analyses can benefit from using component architecture information. The current paper refines the naming of the analyses proposed and discussed in detail in [10].

The **Code Location** analysis considers variation points as related if located in the same parent software element (e.g., variation points of varying statements in the same method). With architecture information available, higher-level structures can be taken into account. Variation points of varying classes, interfaces or packages contained in the same component can be related to each other.

The **Program Structure Dependency** analysis identifies dependencies between variation points' software artifacts (e.g., a variable declaration and the import of the declared type). A component architecture defines additional structural dependencies based on the components' interfaces and their composition. Such explicit dependencies could be analysed in addition to programming language dependencies. For example, if variation points in two independent classes are both adding dependencies to members of an external component, this architectural related dependency can be used to identify an additional relationship between the variation points. Especially in systems making use of techniques for loose coupling (e.g., dependency injection), such composition information provides a benefit for identifying dependencies.

The **Data Dependency** analysis identifies relationships between variation points of software elements which potentially influence each other because of common data objects they manipulate or access. In a similar way, the architecture of a system identifies components which indirectly exchange data with each other (e.g., through a common data store). Again, there might be no obvious connection in the implementation due to a loose coupling.

Program Execution Traces represent program flows monitored during the execution of one or more features. Such traces can be gathered, for example, by instrumenting a software before its execution. As variation points represent locations of variability, relationships are identified between variation points with locations reached by the same execution trace.

Execution traces can become very large depending on their granularity. The information which components are out of scope can be used to filter out parts of the execution traces — during the trace recording or before the analysis — and can speed up the analysis.

A **Change Type** describes a modification of a software entity (e.g., a constructor added to a class). Considering the differences between variants of a variation point as a change, variation points containing differences with the same standardised type of change can be related to each other (e.g., all variation points with a constructor added to a class). The architecture of a system allows for improved analysis constraints, such as relating all variation points with added constructors within the same component. Similarly, changes to component specifications can be related to each other as well (e.g., method signatures added to component interfaces).

A **Change Pattern** is similar to a change type and describes the modification between the variants of a variation point, but can involve multiple software entities and can be specific to the system under study. For example, added constructors which require a boolean parameter, that is checked for a null value in a conditional statement within the constructor. Such specific rules can also involve or be about component structures.

The **Cloned Change** analysis applies a clone detection to the code changes of the variation points. For example, if variation points are about a set of added statements, the clone analysis is applied to those statements to check if the same functionality has been added at several locations.

Clone detection typically requires a lot of processing effort. Component structures can reduce the scope and optimise the performance of the detections. Furthermore, clone detection can also be used on the component structure itself (e.g., identify similar component structure modifications).

The **Semantic Relationship** analysis benefits from developers often introduce semantics not only in comments but also in names of variables, methods and classes. Semantic code clustering techniques can be used to find related variation points based on the names of their varying software elements. Component architecture provides additional semantic information for the analysis (e.g., component or interface names) respectively their contained elements.

The **Common Modifications** analysis investigates in changes performed for the same intention and identifies relationships between variation points linked with those changes. Common modifications are typically committed at once into a revision control system (e.g., svn or git). The analysis can cluster based on single commits or even multiple commits with equal commit messages or contained issue identifiers. A change on the architecture level, such as changing a component structure, often manifests in several individual code refactorings. Interpreting the association of architecture changes to the individual refactorings allows to identify additional common modifications and further variation point relationships.

4.3 Variation Point Design Decision

The design decision activity is the third activity able to benefit from architecture information. During this activity, the product line engineer has to decide about the recommendations provided by the analysis. He can either accept, decline or modify them and decide how they should be realised. By nature, this activity is indirectly influenced by the architecture information's impact on the analysis that created the presented recommendations. In addition, the variation point and recommendation presentation can be improved by arranging them according to the component structure. Furthermore, the product line engineer can manually take additional — probably less formalised — architecture information into account to make a more educated decision.

5. CALCULATOR EXAMPLE

In [9], we have presented an illustrating example to describe how the approach is applied in general. We have extended this example, to include differing external dependencies [1]. The extended example shows how variation point analysis and design can be improved by taking component structures into account. Our prototype implementation [2], is already capable to run this example.

5.1 System Under Study

The example software is a calculator, able to determine the greatest common devisor of two integers as well as the square root of a floating point number. As shown in Figure 3, the CalculatorTool class makes use of the classes CalculatorGCD and CalculatorSqrt to perform the calculations.

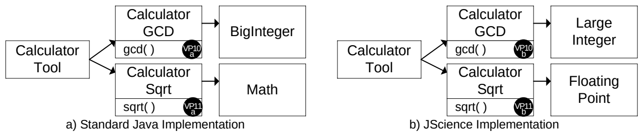

Figure 3: Calculator Class Diagram

The native implementation of the Calculator (a) requires the standard Java library only. The JScience implementation (b) uses external classes (LargeInteger and FloatingPoint) which promise to provide improved quality attributes (e.g., multi-core processor concurrency support, and reduced garbage collection) according to the JScience website [3].

The copied code, adapted to use the classes FloatingPoint and LargeInteger, contains code modifications in the methods `CalculatorGCD.gcd()` and `CalculatorSqrt.sqrt()` as well as changed imports for the differing dependencies. The JavaDoc comments differ as well but are irrelevant for this case study. The listings on the next page show the two differing implementations of the method `gcd()`.

5.2 Applying the Approach

First, the abstract syntax tree (AST) models of the product copies are extracted from their source code and the difference analysis is performed on them. 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. In the `gcd()` listings, the analysis detects four differences, line 4 to 7 (`toString()` is called for different static type definitions). Similar, differences are detected for the method `sqrt()`.

Second, we build an initial VPM based on the difference model. It contains 4 variation points for method `gcd()`, three for method `sqrt()`, and three for added or removed imports referring to the classes `LargeInteger`, `FloatingPoint`, and `BigInteger`.

Next, a VPM analysis is performed, facilitating the code location and program structure dependency analyses. It returns sub-graphs for VP6–VP9 representing the changed statements in `gcd()`, for VP3–VP5 representing the statements in `sqrt()`, and a sub-graph for VP1 and VP2 representing the relationship between the added and removed imports in `CalculatorGCD`. All these sub-graphs lead to merges recommended to the product line engineer. If he accepts the recommendations, a second iteration facilitating the program structure dependency analysis is performed. This second iteration recommends to group the variation points resulted from the merge of the changed imports in `CalculatorGCD` (i.e. VP1 and VP2) and the merged variation points describing the changed statements in `gcd()` (i.e. VP6–VP9) because the statements refer to the imported classes. Finally, accepting these second recommendations leads to the variation points VP10 and VP11. VP10 and VP11 contain two variants as marked in Figure 3 (i.e. VP10a/b and VP11a/b). Each variation point allows to choose between a native Java variant (a) and a JScience-based variant (b).

The resulting native variants require less storage (i.e. ~665kb for the JScience library) which might be relevant (e.g., for the deployment on mobile devices). In contrast, the JScience-based variants require additional storage, but offer improved performance and precision. Focusing on the code-level only, as done up to now, would result in a product line with two independent variation points. It would allow to configure mixed setups with a native GCD and a JScience-based sqrt calculation and vice versa. Both variants make no sense for the design rationale described above because they neither reduce the required storage nor fully utilise the JScience advantages.

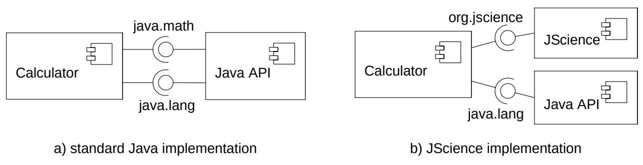

Figure 4: Calculator Component Architecture

Figure 4 shows the component architecture of the product copies. Taking this into account, a further analysis iteration identifies an additional relationship between VP10 and VP11. This relationship results from the JScience component containing the classes `LargeInteger` and `FloatingPoint` and leads to a group recommendation for VP10 and VP11. Accepting this recommendation results in a single variation point for the final product line allowing to choose between a fully native or fully JScience-based variant.

As shown, the approach can be applied straightforward and benefit from the guidance achieved by taking the architecture into account. Final decisions are always up to the engineer to incorporate hidden design knowledge.

[1]http://sdqweb.ipd.kit.edu/wiki/Calculator Example
[2]www.splevo.org
[3]www.jscience.org

Listing 1: Standard Java Implementation

```java
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

```java
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();
}
```

6. RELATED WORK

As surveyed by Chen et al. [3], only few approaches on variability management handle evolutionary aspects such as Loesch [12] or Alves et al. [2]. However, Alves et al. claim the necessity of identifying the variability between product-copies' implementations, none of the approaches provide any support for it.

Similar to Koschke et al. [11], we investigate customised product copies with a common initial code base but do not require a pre-existing module architecture.

Cornelissen et al. [5] used program graph analyses to identify varying features to consolidate. Their approach is complementary to ours and one type of a basic analysis to consider.

Furthermore, there are many approaches to build and analyse feature models from a single system's existing variability (e.g., Acher et al. [1] and She et al. [14]). There feature model analyses are complimentary to our approach. But they do not intent to consolidate product copies and their approaches are not designed to derive how to combine differing code locations to a single variable one.

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 and identified which activities can benefit from what type of architecture. We discussed the architecture influence on the differencing and variation point analysis activities in more detail.

Finally, we discussed an exemplary application of our approach to illustrate how it is applied and how it benefits from architecture information.

Currently, we are working on a case study on the ArgoUML-SPL software provided by Couto et al. [6]. In our future work, we investigate in the value of the individual relationship analysis strategies. Furthermore, we are working on automated support for the downstream refactoring for individual variability realisation techniques.

Acknowledgment This work was supported by the German Federal Ministry of Education and Research (BMBF), grant No. 01IS12012B.

8. REFERENCES

[1] M. Acher et al. Reverse Engineering Architectural Feature Models. In *Proc of ECSA'11*. Springer, 2011.

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

[3] L. Chen, M. Ali Babar, and N. Ali. Variability management in software product lines: a systematic review. In *Proc of SPLC'09*, 2009.

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

[5] B. Cornelissen, B. Graaf, and L. Moonen. Identification of variation points using dynamic analysis. In *Proc of R2PL'05*, 2005.

[6] M. V. Couto, M. T. Valente, and E. Figueiredo. Extracting Software Product Lines : A Case Study Using Conditional Compilation. In *Proc of CSMR'11*, 2011.

[7] S. Deelstra, M. Sinnema, J. Nijhuis, and J. Bosch. COSVAM: a technique for assessing software variability in software product families. *Proc of CSMR'04*, 2004.

[8] K. C. Kang et al. Feature-oriented domain analysis (FODA) feasibility study. Technical Report November, Software Engineering Institute - Carnegie Mellon University, 1990.

[9] B. Klatt and M. Küster. Respecting component architecture to migrate product copies to a software product line. In *Proc of WCOP '12*. ACM Press, 2012.

[10] B. Klatt, M. Küster, and K. Krogmann. A Graph-Based Analysis Concept to Derive a Variation Point Design from Product Copies. In *Proc of REVE'13*, 2013.

[11] R. Koschke et al. Extending the reflexion method for consolidating software variants into product lines. *Software Quality Journal*, 17(4), 2009.

[12] F. Loesch and E. Ploedereder. Optimization of Variability in Software Product Lines. In *Proc of SPLC'07*. IEEE, 2007.

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

[14] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. Reverse engineering feature models. In *Proc of ICSE'11*. IEEE, 2011.

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

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

[17] Z. Xing and E. Stroulia. UMLDiff: an algorithm for object-oriented design differencing. In *Proc of ASE'05*. ACM, 2005.