Towards a Language-Independent Approach for Reverse-Engineering of Software Product Lines

Tewfik Ziadi  
UPMC LIP6  
Paris, France  
tewfik.ziadi@lip6.fr  

Christopher Henard  
SnT, University of Luxembourg  
Luxembourg, Luxembourg  
christopher.henard@uni.lu  

Mike Papadakis  
SnT, University of Luxembourg  
Luxembourg, Luxembourg  
michail.papadakis@uni.lu  

Mikal Ziane  
LIP6  
Paris, France  
mikal.ziane@lip6.fr  

Yves Le Traon  
SnT, University of Luxembourg  
Luxembourg, Luxembourg  
yves.letraon@uni.lu  

ABSTRACT

Common industrial practices lead to the development of similar software products. These products are usually managed in an ad-hoc way which gradually results in a low product quality. To overcome this problem, it is essential to migrate these products into a Software Product Line (SPL). Towards this direction, this paper proposes a language-independent approach capable of reverse-engineering an SPL from the source code of product variants. A prototype tool and a case study show the feasibility and the practicality of the introduced technique. This paper concludes the paper and presents our future work.

Categories and Subject Descriptors

D.2.7 Categories and Subject Descriptors: Distribution, Maintenance, and Enhancement—Restructuring, reverse engineering, and reengineering

General Terms

Algorithms, Experimentation, Languages

Keywords

Reverse-engineering, Software Product Lines

1. INTRODUCTION

Software Product Line Engineering (SPLE) is a software development paradigm designed to handle software products variants that share a common set of features. While objects and components enhance the reusability of software, SPLE performs a step further by allowing the reuse and the management of software features. The benefits of SPLE include the reduction of both the maintenance effort and the time to market. SPLE involves the creation and the management of a Software Product Line (SPL). The products of an SPL are usually represented by a variability model called Feature Model (FM) which allows building tailored products by combining the features.

SPLE can be implemented as a top-down approach. In that case, features and variability are first specified at the design stage and then software products are built. The top-down process is useful when SPLE is adopted from the beginning. However, current practices are different. Indeed, as reported by Berger et al., most of the industrial practitioners first implement several software products and then try to manage them. These product variants are created using ad-hoc techniques, e.g., copy-paste-modify. This is a bad practice leading to a complex management and a low product quality. Thus, migrating such products to an SPL is the challenge faced by extractive approaches in SPLE.

Over the past decade, several studies have investigated such approaches. Some of them deal only with the extraction of features from textual requirements, architectural artifacts, or product descriptions. We advocate that these techniques can go beyond the feature identification step. Indeed, they can refactor and migrate a set of software products into an SPL. In other words, a full implementation of an SPL can be reverse-engineered. Therefore, not only features but also their associated assets, e.g., code units, should be identified and extracted.

Towards this direction, this paper introduces an automated technique called ExtractorPL, capable of performing a reverse-engineering of an SPL. ExtractorPL infers a full implementation of an SPL given the source code of product variants. The main challenge of this task is to analyze the source code of the products in order to (a) identify the variability among the products, (b) associate them with features, (c) regroup the features into a variability model, and (d) map the code units to each feature. The proposed approach is language-independent and only uses as input the source code of the product variants. In addition, ExtractorPL is implemented as a publicly available prototype tool and a case study performed on existing SPL assesses the feasibility and the practicality of the introduced technique.

The remainder of this paper is organized as follows. Section 2 and 3 respectively present the ExtractorPL approach and a case study to evaluate it. Section 4 discusses the benefits and the limitations of the approach. Then, Section 5 presents work related to the present one. Finally, Section 6 concludes the paper and presents our future work.
2. THE EXTRACTORPL APPROACH

ExtractorPL is a language-independent approach which extracts an SPL from the source code of product variants. The re-engineered SPL is a full implementation since it allows (a) building specific products by composing the source code units of the identified features and (b) managing the resulting SPL and its products through an FM.

To achieve this re-engineering, three main steps are considered. These steps are depicted by Figure 1. First, ExtractorPL abstracts each product variant into a set of atomic pieces called Set of Construction Primitives (SoCPs). Each Construction Primitive (CP) represents a node in an abstract syntax tree for features called Feature Structure Tree (FST). Then, following the lines suggested in [26], features are identified as SoCPs and translated to FSTs. Finally, the code units are generated from the obtained features’ FSTs.

The following subsections introduce an example of product variants from a banking system that can be migrated into an SPL. Then, the three main steps of the ExtractorPL approach are detailed through the example.

2.1 The Banking System Example

As a concrete example of software product variants, consider a set of banking systems [26]. Each variant proposes a simple banking application. The variability between these product variants is related to the limit on the account and to the currency exchange, which are optional features. These banking products were manually developed using a copy-paste-modify approach.

Figure 2 illustrates the source code of the Account class in the three variants denoted as Product1Bank, Product2Bank, and Product3Bank. Following this figure, consider the Product1Bank product depicted by Figure 2(a). In this product, the Account class defines a basic banking account without the limit and currency information. On the contrary, since the Product2Bank of Figure 2(b) supports an account limit feature, its corresponding Account class defines the limit field. In addition, the withdraw method is refined to check the limit. Finally, in the Product3Bank variant of Figure 2(c), the Account class is defined with information related to both the limit and currency exchange.
inner modules, e.g., packages and classes. Terminal nodes
are used: each FST. In this work, the following construction primitives
to decompose each FST depend on the type of nodes within
them as a SoCPs. The construction primitives (CPs) used
and terminal
Each product variant is thus abstracted as a set of FSTs.
For each node in each obtained FST, a CP is created and
added to the SoCPs. This means that each product variant
is defined as a set \( P = \{ cp_1, cp_2, \ldots, cp_n \} \), where each \( cp_i \in \) SoCPs. In the following, we consider \( \text{AllP} = \{ P_1, P_2, \ldots, P_N \} \)
as the set of product variants available to perform the reverse-
engineering of the SPL, i.e., the input of the ExtractorPL
approach.

The left part of Figure 3 depicts the three FSTs obtained
from the source code of the banking products. For instance,
the Account class is represented by a node with the name
Account and the type Class.

The following subsection introduces the algorithm that
compares and analyzes the extracted SoCPs. Equivalence
between the construction primitives relies on the equivalence
between nodes in the corresponding FST [4], as defined be-
low.

1. A non-terminal node \( n_1 \) is equivalent to a non-terminal
node \( n_2 \) if and only if \( n_1 \) and \( n_2 \) have the same name,
the same type and the same node child.

2. A terminal node \( n_1 \) is equivalent to a terminal child
\( n_2 \) if and only if they have the same name, the same
type, the same parent and the same body.

\[ cp_1 \in \{ \text{CreateNonTerminal(name, type, child),} \]
\[ \text{CreateTerminal(name, type, parent, body)} \}. \]

\[ \text{SoCPs} = \{ \text{CreateNonTerminal(name, type, child),} \]
\[ \text{CreateTerminal(name, type, parent, body)} \}. \]

We distinguish two types of nodes within an FST: non-
terminal and terminal ones. A non-terminal node denotes
inner modules, e.g., packages and classes. Terminal nodes
store the module’s content, e.g., method bodies [4].
2.3 Automatic Identification of the Software Product Line Features

The second step of the reverse-engineering process performed by our approach aims at comparing the SoCPs of the product variants in order to identify their features. To this end, ExtractorPL first uses the algorithm proposed in [26] to represent features as sets of SoCPs. Then, these features are transformed into FSTs.

The feature identification process is based on a formal definition of a feature that uses the notion of interdependent CPs. This notion is defined as follows.

**Definition 1 (Interdependent CPs).** Given the set of product variants that can be used by ExtractorPL, AllP, two CPs (of products of AllP) $cp_1$ and $cp_2$ are interdependent if and only if they belong to exactly the same products of AllP. In other words, $cp_1$ and $cp_2$ are interdependent if the two following conditions are fulfilled.

1. $\exists P \in AllP \; cp_1 \in P \land cp_2 \in P$.
2. $\forall P \in AllP \; cp_1 \in P \Leftrightarrow cp_2 \in P$.

Since interdependence is an equivalence relation on the set of CPs of AllP, it leads us to the following definition of a feature.

**Definition 2 (Feature).** Given AllP a set of products, a feature of AllP is an equivalence class of the interdependence relation of the CPs of AllP.

The application of this algorithm to the SoCPs of the banking products provides the features depicted by Figure 4. This includes one mandatory feature and three optional ones. The Base feature gathers all the CPs that are present in all the product variants. The feature F1 concerns the limit information. Indeed, it contains primitives to create the limit field, its getter and the withdraw method with the body defining limit checking. F2 is related to the currency exchange since it contains the CPs related to the currency field. The F3 feature is related to the withdraw method without limit checking. Finally, ExtractorPL also organizes the obtained features into a FM. This model is depicted by Figure 5.

Figure 5: The Feature Model (FM) built by ExtractorPL for the banking example. It represents the variability between the extracted features of the reengineered SPL.

The algorithm for identifying the features of a given set of products [26] is based on this definition. In brief, it takes as input a set of SoCPs, i.e., one per input product variant and returns a single mandatory feature called Base and a set of optional features for these product variants. Finally, once the features of the product variants have been identified, they are represented as FSTs in order to be useful for generating the code of each feature.

2.4 Feature Code Generation

In addition to the feature identification, ExtractorPL aims at extracting a full compositional implementation of an SPL from the source code of product variants. It means that our approach includes the generation of the code units associated to each feature once the SPL is extracted. As a result, one is able to build tailored software products by automatically generating the source code of the features selected.

To perform the code generation, ExtractorPL uses the FSTs of the features obtained in the previous step. Then, it generates their code units by using the pretty printers proposed by FeatureHouse. Indeed, the FeatureHouse framework includes a set of pretty printers for different programming languages such as C or Java. These printers produce the code from FSTs. Using this framework within our approach allows ExtractorPL being a language-independent technique.

Finally, to demonstrate the result of the code generation, Figure 6 depicts the code units generated from the FSTs of the features. Following this figure, the code associated with the Base feature represents the common implementation of the Account class. The code units of the remaining optional features express the variability and refine the common code by adding elements related to each feature. Given the code of each feature, a product variant of the resulting SPL can be built using the compositional operators between FSTs. The following section evaluates ExtractorPL via a case study.

**Figure 4:** The identified features for the banking product variants. Base gathers the code common to any banking system. F1 concerns the limit information. F2 represents the currency exchange and F3 denotes the withdraw method without the account limit checking.

**Figure 5:** The Feature Model (FM) built by ExtractorPL for the banking example. It represents the variability between the extracted features of the reengineered SPL.

**Figure 6:** The code units generated from the FSTs of the features.
3. CASE STUDY

In this section, ExtractorPL is assessed. The question raised here is how this evaluation can be performed. Generally, an automated reverse-engineering task can be considered as successful when it provides similar results to a manually performed one. Therefore, one way to measure the accuracy of our approach is to compare its result with the result from a developer. Another way is to measure whether the extracted SPLs provide a minimum quality standard as defined by the two following requirements.

1. The extracted SPL allows building the products that have been used to perform the re-engineering.
2. The approach identifies the features that must appear in all the possible variants of the SPL. These features are usually called mandatory features.

If a re-engineered SPL cannot fulfill the first condition, it is obviously an erroneous approach. Similarly, if the second condition cannot be satisfied, then there is no hope to identify optional features. Following the above-mentioned concerns, a controlled experiment is conducted based on existing SPLs. We use products from existing SPLs in order to establish a comparison basis between the existing SPLs and the extracted ones. As a result, this case study aims at answering the two following research questions:

• [RQ1] How close the SPL re-engineered by ExtractorPL is to the original one?
• [RQ2] Does the re-engineered SPL has a minimum quality standard?

The first research question amounts to evaluate whether the SPL extracted with ExtractorPL conforms to the original one. To this end, we manually compare the original SPL with the extracted one. In particular, we check whether extracted features correspond to those defined in the original version of the SPL. The second research question aims at checking whether the above-mentioned minimum requirements are fulfilled by the SPL extracted by our approach.

The evaluation of ExtractorPL is divided into two parts. The first one is performed manually and aims at answering to the RQ1. The second one performs automatically in order to answer to the RQ2.

3.1 Relation Between the Re-Engineered SPL and a Manually Produced One (RQ1)

In this section, we compare the SPL resulting from ExtractorPL with the original one. This is a manual step and thus it requires a lot of effort to be accomplished. Therefore, in order to complete the experiments with reasonable resources, the evaluation is limited to one benchmark.

3.1.1 Setup

We use a notepad SPL written in Java [21]. This SPL is detailed in Table 1. It contains 14 Java classes for a total of 806 lines of code. It proposes three optional features: copy/cut/paste, undo/redo and find. By combining these three features, up to 8 different notepad applications can be built. These 8 product variants are used by ExtractorPL to reverse-engineer an SPL. We manually compare the features extracted by ExtractorPL using the 8 product variants with the features of the notepad SPL.

3.1.2 Evaluation

ExtractorPL has extracted 4 features. The first one, Base, contains the 7 classes related to the core of any notepad variant: About, Actions, Center, ExampleFileFilter, Fonts, Notepad and Print. The three other features, F1, F2, and F3 are related to the optional features of the original SPL. Indeed, F1 contains the Notepad and Actions classes. This latter defines the copy, cut and paste methods. As a result, F1 is related to the copy/cut/paste feature. The second feature, F2, encompasses the Notepad and Actions with the find method and attributes. Finally, F3 contains the Notepad, Redo and Undo actions. Finally, we manually checked that the extracted SPL allows generating the code of all the products used as input and that they can be executed without encountering any problem.

| Lines of code | 806 |
|---------------|-----|
| Classes       | 14  |
| Optional features | Copy/Cut/Paste (CCP)  |
|               | Undo/Redo (UR)       |
|               | Find (F)             |
| Product variants | Basic Notepad (BN)  |
|               | BN + CCP             |
|               | BN + UR              |
|               | BN + F               |
|               | BN + CCP + UR        |
|               | BN + CCP + F         |
|               | BN + UR + F          |
|               | BN + CCP + UR + F    |

Table 1: The Notepad Software Product Line (SPL) used to evaluate ExtractorPL towards RQ1.
3.1.3 Answering RQ1
The extracted SPL conforms with the original one. We compared the obtained features and variants and found that our approach is able to accurately retrieve the features and to re-generate the products used as input. As a result, given a set of product variants ExtractorPL is able (a) to retrieve the variability among these products and (b) to extract a SPL that is representative of the original one.

3.2 Minimum Quality Standard Evaluation of the Re-Engineered SPL (RQ2)

The second part of this study aims at evaluating whether the re-engineered SPL achieves a minimum level of quality.

3.2.1 Setup
We use the two following SPLs from FeatureIDE [21]:

1. E-Mail. This SPL gathers a family of systems managing mails. Examples of optional features in this SPL include encryption or the address book.
2. GPL. The Graph Product Line is a family of graph manipulation algorithms [17].

We choose these particular SPLs as they are considered to be standard benchmarks. Table 2 gathers detail regarding these two SPLs. In particular, for each SPL, it presents the programming language in which it is implemented, the number of features of the corresponding FM, the number of possible product variants that can be built according to the FM and the number of input products that have been used by ExtractorPL to extract the SPL. Indeed, we used a sample of products to extract the SPL since hundreds of products can be build from these SPLs.

The configuration of the products used as input by our approach are configurations randomly selected from the space of all the possible configurations that can be generated from the FM [17]. For each configuration randomly generated, we use FeatureHouse to construct the corresponding software product variant. The resulting products are then used by ExtractorPL to reverse-engineer the SPL.

For each SPL and for each number of products used by our approach, the extraction of the SPL has been independently performed 10 times. In the following, we present two approaches to automatically evaluate the resulting SPL. The first one compares the products generated by ExtractorPL with the ones used to extract the SPL. We expect the resulting SPL to be able to build the products that were used as input. The second steps automatically evaluates the mandatory features. Here, we expect the mandatory features of the original SPL to be included in the mandatory features of the extracted SPL.

3.2.2 Regeneration of the Input Products
The objective is to check whether the extracted SPL allows building the products that were used by ExtractorPL. To this end, we check whether the SoCPs of a given input product (of the original SPL) matches the SoCPs of the corresponding product built from the re-engineered SPL. More formally, if \( \text{AllP}_{\text{in}} \) denotes the set of \( N \) input products used as input and if \( \text{AllP}_{\text{out}} \) denotes the set of \( N \) products generated from the extracted SPL, we check that:

\[
(\forall P \in \text{AllP}_{\text{in}})(\exists P' \in \text{AllP}_{\text{out}}) | P = P',
\]

where \( P \) and \( P' \) are products represented as a SoCPs. The equivalence between two products \( P \) and \( P' \) is defined as an equivalence between their SoCPs. For each of the two SPL of Table 2, and for each number of input products, all the 10 runs of the approach produced an SPL which allows regenerating the inputs products, thus validating the above-mentioned condition.

3.2.3 Evaluation of the Mandatory Features
ExtractorPL extracts one mandatory feature called Base and a set of optional features. In this section, we evaluate whether the mandatory features of the original SPL are included in the mandatory feature of the SPL extracted with ExtractorPL. To this end, we check whether the SoCPs of the original mandatory features are included in SoCPs of the mandatory feature of the re-engineered SPL. Thereby, if \( \text{SoCP}_{\text{in}} \) denotes the SoCPs of the input mandatory features and if \( \text{SoCP}_{\text{out}} \) denotes the SoCPs of the extracted mandatory feature, we check that:

\[
\text{SoCP}_{\text{in}} \subseteq \text{SoCP}_{\text{out}}.
\]

The evaluation has been performed 10 times independently per SPL. For each SPL and for each number of random products used as an input by the approach, we observed that the original mandatory features are included in the mandatory feature of the extracted SPL. It is noted that all the mandatory features are always validated on both the E-Mail (C) and GPL (Java) SPLs, fact which demonstrates the ability of ExtractorPL to retrieve the mandatory features.

3.2.4 Answering RQ2

From the results presented in the previous sections, we found that (a) the extracted SPL allows building the products used to extract this SPL, and (b) all the mandatory features of the original SPL are included in our extracted SPL. It means that our approach does not miss any information, thus fulfilling the minimum quality requirements defined in the beginning of this section.

3.3 Threats to Validity

The conducted study involves three existing SPLs. As a consequence, there is a threat regarding the generalization of the results. Indeed, using different SPLs might lead to different results. To both reduce this threat and to provide a good sample of applications, we used three SPLs considered as standard benchmarks. These three SPLs are of different size and programming languages. Other threats can be

| Language | Features | Product variants | Classes/Files | Lines of code | Products used by ExtractorPL |
|----------|----------|-----------------|--------------|--------------|-----------------------------|
| E-Mail   | C        | 23              | 39           | 816          | 1, 5, 10, 50 and 100        |
| GPL      | Java     | 38              | 840          | 1929         | 1, 5, 10, 50 and 100        |

Table 2: The Software Product Lines (SPLs) used for the evaluation of the approach towards RQ2.
identified due to the employed evaluation metrics. In other words, there is a risk that the quality measures are irrelevant towards the “real” quality of an SPL. To reduce this threat, we evaluate the approach using manual and automatic metrics. Additional threats can be due to our implementation. Indeed, potential errors in it might affect the presented results. To overcome this issues, we divided our implementation into modules to minimize the potential errors. We also make the prototype tool publicly available. Finally, there is a threat regarding results that could happen by chance. To minimize the risks attributed to random effects, we repeated the experiments 10 times independently.

4. DISCUSSION

This section first discusses the reasons to migrate existing product variants to an SPL. In this respect, several profits are provided by our tool. Then, some limitations regarding the proposed approach are highlighted.

4.1 Benefits

Migrating a set of existing products to an SPL can lead to the following profits. First, in can reduce the development costs. Indeed, an SPL allows building tailored software products by combining the features. It thus allows reusing existing code within different products. This can be performed automatically and without adapting the code.

Second, it can bestow a faster time to market. A full implementation of an SPL as proposed by our approach easily allows building the products by only selecting the desired features. The products are then generated by composing the code of the selected features. This allows configuring easily the products depending on the targeted market. It also greatly decreases the time to build these products and enables a flexible productivity.

Another outcome is the higher quality in the products. Migrating existing products to an SPL leads to a reduced risk to introduce errors when new products are created. Indeed, creating product variants with ad-hoc techniques like copy-paste-modify can lead to an introduction of errors in some variants. If the code of each feature is centralized within the SPL and shared in all the product proposing these features, it allows testing each feature independently. This allows using SPL testing techniques which aim at testing the whole SPL in an efficient way.

Finally, moving products to an SPL provides a higher quality in the products developed, thus leading to an easier management and maintenance of the products. In particular, the variability model such as the FM allows managing and tailoring the products. Besides, the code contains less redundancy and is refactored according to the underlying model.

Regarding ExtractorPL, it is the first approach to the authors’ knowledge which allows building an SPL from a set of product variants. Indeed, existing approaches require additional information to perform the reverse-engineering, like annotations in the code. On the contrary, our approach is fully automated and requires only the source code of the products variants (Section 5) gives more detail regarding other techniques. In addition to the extraction of the code units of the features, our approach also extracts a FM. Such a model provides a high-level view of the variability within the product line. It also allows visualizing the features, their dependencies and paves the way to reasoning and model-based testing of the SPL.

4.2 Limitations

ExtractorPL does not consider variability within the body of methods or functions, i.e. the statement level. We are working on the extension of the approach to remove this limitation. The idea is to modify the FSTs grammar to allow defining nodes for the statements of functions. Besides, the current implementation of ExtractorPL only infers an FM with a single mandatory feature and a set of optional features. This model does not encompasses constraints among the features, e.g. implications or exclusions. We are working on an extension of the proposed approach to infer a possible list of constraints from the features that are observed in the product variants. These constraints will then be proposed to the user to be accepted and added to the FM.

5. RELATED WORK

Whether extracting a product line is useful has been assessed in [6]. In this work, Berger et al. investigated the assessment of product variants to extract a product line. They propose a set of metrics that enable the software architects and project managers to estimate whether it is beneficial or not to construct a product line. This work is complementary to our and can be done as a prior step to our approach.

There are few work related to the extraction of an SPL from the source code of product variants. Yoshimura et al. propose a reverse-engineering process for variability. This work also abstracts input product variants using Abstract Syntax Tree (AST) models. The extraction of the SPL implementation is not considered in these work.

Zhang et al. present a framework for re-engineering variability. However, this work only focus on the extraction of variability from the source code with conditional compilation directives. Xi et al. propose an approach based on Formal Analysis Concept (FCA) for the identification of code units that are associated with a set of existing features. Indeed, in addition to the source code of product variants, this approach also takes as input data the list of features associated to each product variant. It then tries to locate the code units associated to each feature. The difference between this work and our approach is that ExtractorPL only considers the source code of the product variants as input data, without any additional information.

Extracting the variability from other assets than source code has been investigated in several work, e.g., [8]. In [19], Rubin et al. propose an approach to extract a product line from architectural artifacts. Frenzel et al. use the reflexion method to refactor a collection of product architectures into an SPL architecture. In their work, variability is specified using annotations. In [1], FMs are extracted from plugin dependencies and architecture fragments of product variants. Yssel et al. consider the extraction of FMs from a set of similar models that represent function blocks, a kind of architectural models for embedded systems. ExtractorPL can be modified to use architectural artifacts. Indeed, FSTs can be employed to abstract architectural models.

Finally, while ExtractorPL extracts an SPL from the source code of a set product variants, Valente et al. propose a semi-automatic approach where an SPL is extracted from a single software product.
6. CONCLUSION

Automatically migrate a set of software product variants to an SPL is not an easy task. It requires to perform several non-trivial steps including (a) the identification of the features in the source code of the products, (b) the extraction of the features as code units, (c) the extraction of a variability model and (d) once the SPL is extracted, the correct composition of these features in order to build tailored software products. We tackled this problem with ExtractorPL, a language-independent approach which provides a quick automatic front-end to refactor a set of similar product variants into a SPL. ExtractorPL has been implemented in a prototype tool based on which several experiments have been conducted. Our technique bestows two main outcomes.

- **It is a full extractive approach.** From the source code of a set of product variants, ExtractorPL extracts a full implementation of an SPL. This includes the features, their code units, and a variability model.

- **It is a language-independent approach.** ExtractorPL only manipulates FSTs to extract a SPL. To integrate new languages or artifacts, it only requires to implement a parser for FSTs related to this language.

Finally, to enable reproducibility of our results, our implementation of ExtractorPL is publicly available at:

http://pagesperso-systeme.lip6.fr/Tewfik.Ziadi/sac14/

7. REFERENCES

[1] M. Acher, A. Cleve, P. Collet, P. Merle, L. Duchien, and P. Lahire. Extraction and evolution of architectural variability models in plugin-based systems. In *SoSyM*, pages 1–28, 2013.

[2] M. Acher, A. Cleve, G. Perronin, P. Heymans, C. Vanbeneden, P. Collet, and P. Lahire. On extracting feature models from product descriptions. In *VaMoS*, pages 45–54, 2012.

[3] V. Alves, C. Schwanninger, L. Barbosa, A. Rashid, P. Sawyer, P. Rayson, C. Pohl, and A. Rumrull. An exploratory study of information retrieval techniques in domain analysis. In *SPLC*, pages 67–76, 2008.

[4] S. Apel, C. Kästner, and C. Lengauer. Featurehouse: Language-independent, automated software composition. In *ICSE*, pages 221–231, 2009.

[5] D. Benavides, S. Segura, and A. R. Cortés. Automated analysis of feature models 20 years later: A literature review. *Inf. Syst.*, 35(6):615–636, 2010.

[6] C. Berger, H. Rendel, and B. Rumpe. Measuring the ability to form a product line from existing products. In *VaMoS*, volume 37, pages 151–154, 2010.

[7] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wasowski. A survey of variability modeling in industrial practice. In *VaMoS*, page 7, 2013.

[8] J.-M. Davril, E. Delfosse, N. Hariri, M. Acher, J. Cléland-Huang, and P. Heymans. Feature model extraction from large collections of informal product descriptions. In *ESEC/FSE*, pages 290–300, 2013.

[9] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel. An empirical study of the effect of time constraints on the cost-benefits of regression testing. In *FSE*, pages 71–82, 2008.

[10] P. Frenzel, R. Koschke, A. P. J. Breu, and K. Angstmann. Extending the reflexion method for consolidating software variants into product lines. In *WCRE*, pages 160–169, 2007.

[11] C. Henard, M. Papadakis, G. Perronin, J. Klein, P. Heymans, and Y. L. Traon. Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test suites for large software product lines. *CoRR*, abs/1211.5451, 2012.

[12] C. Henard, M. Papadakis, G. Perronin, J. Klein, and Y. L. Traon. Multi-objective test generation for software product lines. In *SPLC*, pages 62–71, 2013.

[13] C. Henard, M. Papadakis, G. Perronin, J. Klein, and Y. L. Traon. Pledge: a product line editor and test generation tool. In *SPLC Workshops*, pages 126–129, 2013.

[14] K. Kang, J. Lee, and P. Donohoe. Feature-oriented product line engineering. *Software, IEEE*, 19(4):58–65, jul/aug 2002.

[15] B. Klatt and M. Küster. Respecting Component Architecture to Migrate Product Copies to a Software Product Line. In *WCO’12*, June 2012.

[16] C. W. Krueger. Easing the transition to software mass customization. In *PFE*, volume 2290, pages 282–293, 2001.

[17] R. E. Lopez-Herrejon and D. S. Batory. A standard problem for evaluating product-line methodologies. In *GCSE*, volume 2186, pages 10–24, 2001.

[18] K. Pohl, G. Böckle, and F. J. v. d. Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. 2005.

[19] J. Rubin and M. Chechik. Combining related products into product lines. In *FASE*, volume 7212, pages 285–300, 2012.

[20] U. Ryssel, J. Ploennigs, and K. Kabitzsch. Automatic variation-point identification in function-block-based models. In *GPCE*, pages 23–32, 2010.

[21] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich. Featureide: An extensible framework for feature-oriented software development. *Science of Computer Programming*, 2012.

[22] M. T. Valente, V. Borges, and L. T. Passos. A semi-automatic approach for extracting software product lines. *IEEE Trans. Software Eng.*, 38(4):737–754, 2012.

[23] Y. Xue, Z. Xing, and S. Jarzabek. Feature location in a collection of product variants. In *WCRE*, pages 115–154, 2012.

[24] K. Yoshimura, F. Narisawa, K. Hashimoto, and T. Kikuno. Fave: factor analysis based approach for detecting product line variability from change history. In *MSR*, pages 11–18, 2008.

[25] B. Zhang and M. Becker. Recovar: A solution framework towards reverse engineering variability. In *Proc. PLEASE*, pages 45–48, 2013.

[26] T. Ziadi, L. Frias, M. A. A. da Silva, and M. Ziane. Feature identification from the source code of product variants. In *CSMR*, pages 417–422, 2012.