Managing Cloned Variants: A Framework and Experience

Julia Rubin[1,2], Krzysztof Czarnecki[3], Marsha Chechik[1]

[1]University of Toronto, Canada
[2]IBM Research at Haifa, Israel
[3]University of Waterloo, Canada

mjulia@il.ibm.com, kczarnec@gsd.uwaterloo.ca, chechik@cs.toronto.edu

ABSTRACT

In our earlier work, we have proposed a generic framework for managing collections of related products realized via cloning – both in the case when such products are refactored into a single-copy software product line representation and the case when they are maintained as distinct clones. In this paper, we ground the framework in empirical evidence and exemplify its usefulness. In particular, we systematically analyze three industrial case studies of organizations with cloned product lines and derive the set of basic operators comprising the framework. We discuss options for implementing the operators and benefits of the operator-based view.

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*

General Terms

Design, Management

Keywords

Legacy software product lines, cloned product variants, industrial case studies.

1. INTRODUCTION

Software Product Line Engineering (SPLE) promotes strategic, well-managed software reuse [7, 24]. In reality, however, reuse might be ad-hoc, often realized via artifact cloning (the "clone-and-own" approach). Over the past decade, several strategies for dealing with cloned product variants have been proposed. Some [9, 21, 39, 30] advocate refactoring them into software product line (SPL) representations (the product line *merge-refactoring*), whereas others [35, 38, 37] propose mechanisms for maintaining multiple variants without attempting to refactor them. Yet, these product maintenance and reverse engineering solutions are often "monolithic" and are designed with a specific project context or particular application domain in mind, making it difficult to reuse existing work for a new case at hand. For example, an approach for merge-refactoring Matlab/Simulink models [30] is not directly applicable to code handwritten in C++.

Some approaches, often developed in seemingly unrelated contexts, can be used as "building blocks" for cloned product line management tasks – both merge-refactorings and clone maintenance. Examples of such approaches include *feature location* [28] aimed at tracing feature declarations to their corresponding implementations, as well as *feature interaction detection* [40] aimed at spotting conflicts caused by feature compositions. However, sufficient guidance on which approaches are needed and which of them are applicable for a given scenario does not exist. Selecting the right implementation for an approach is also challenging: even though more than 20 feature location techniques are available [28], it is unclear how to select one that works best in a particular context as assumptions made by these techniques are not always explicated. Moreover, existing approaches cover only a subset of activities related to the development, maintenance and refactoring of cloned product variants; complementary approaches also need to be identified and developed.

The goal of our research is to support the management of cloned variants by *providing a framework for organizing knowledge related to the development, maintenance and merge-refactoring of product lines realized via cloning*. We empirically analyze a number of industrial case studies and devise a framework that specifies and organizes a set of basic management operators – those that are required to support both *the case when refactoring of the cloned variants into SPL representations is performed* and *the case when such a refactoring is not desired or possible* [8]. In fact, we show that these two cases share a substantial number of common operators.

In our earlier 4-page paper [26], we gave an initial version of the framework and its operators, focusing on the landscape of existing implementation approaches and the gaps that remain. The goal of this work is both to ground the framework on empirical evidence and exemplify its usefulness. In particular, we systematically analyze three industrial case studies that rely (or have relied) on cloning to realize their product lines, derive the set of the required management activities and demonstrate their decomposition into a set of basic operators.

We do not attempt to produce a complete set of operators yet, but rather to explore and demonstrate the benefits of the operator-based view for a more systematic organization of development tasks. We aim to provide a common vocabulary to describe and compare existing techniques, helping identify remaining gaps and eventually moving towards a recommender system that can assist users in selecting an approach that best fits their purpose.

SPLC 2013 August 26 - 30 2013, Tokyo, Japan

Contributions. This paper makes the following contributions.

1. We perform a detailed analysis of development activities in three industrial organizations that employ (or have employed) cloning to realize their product portfolio, decomposing them into operator instances.
2. We present a refined version of the cloned product line management framework and its operators, incorporating lessons learned from the analysis.
3. We propose the operator-based view for organizing the management tasks and creating a *body of knowledge* around their implementations.

The reminder of the paper is structured as follows. In Section 2, we describe our research methodology and give an organizational context of the three case studies being analyzed. Section 3 describes the activities that are performed as part of the transition from cloning to an SPL approach and identifies the relevant operators. Section 4 identifies the operators required for the interim coping with clones. We describe our vision of building the body of knowledge around the operators in Section 5. Section 6 discusses related approaches, while Section 7 concludes the paper with a summary and an outline of future research directions.

2. RESEARCH METHODOLOGY AND CASE STUDIES

In this section, we describe our research methodology and the industrial case studies that we analyzed (see also Table 1 for a summary). We selected case studies based on the following criteria:

1. The company employs (or used to employ) cloning to realize its product portfolio.
2. We have access to a significant amount of company-specific data and a deep understanding of the development process.

These criteria lead us to choose three partner companies, diverse enough to draw insightful conclusions. The first one is currently in the advanced stage of the transition from cloning to a managed SPLE approach (case study #1 – Section 3.1). The transition of the second one, Danfoss Drives, is completed and has been reported in [13, 12] (case study #2 – Section 3.2). In the third case, the company still relies on cloning to implement its commercial products, and the transition to SPLE has just been initiated (case study #3 – Section 3.3). Confidentiality issues prevent us from sharing the names of the first and the third companies, which are both from the aerospace and defense domains.

We performed semi-structured interviews with the employees of the studied companies and completed our understanding of the Danfoss Drives case using published data [13, 12]. Our analysis focused on each company's development environment, including the tools it uses, the development artifacts it maintains, the processes it follows, the target SPLE approach the company aims to adopt and the challenges it faces, as described in the remainder of this section. We then analyzed in detail the activities that the company performs, identified their atomic steps, mapped those steps to instances of the operators and discussed automation opportunities (see Sections 3 and 4).

2.1 Case study #1

For the first case study, we analyze a software product line of a large aerospace company, developed over the course of 15-20 years by approximately 300 engineers, half of them system- and half software engineers. As summarized in the second column of Table 1, the product line of this company is realized using four types of artifacts: system requirements, software requirements, design models and tests. System and software requirements are maintained in an internal requirements management tool similar to IBM Rational Doors[1]. Design models are specified in SCADE[2], with code being automatically generated from the design models using model-driven technologies [32]. Due to regulatory requirements[3], a detailed vertical traceability is established between the artifacts: from system to software requirements and further to the design models and code.

The development process strictly follows a waterfall model. Over 50 products of the product line are implemented by cloning (captured by a complex clone derivation graph) with the cloned variants kept as separate branches in a Software Configuration Management (SCM) system. The goal of the company is to establish a component library as well as a configurable framework architecture connecting the components. Those are to be used as a basis for all future products of the product line. However, the company does not intend to re-implement existing products on top of the common framework – these products are to be maintained as is.

2.2 Case study #2

Danfoss Drives is one of the largest producers of frequency converters – electronic power conversion devices used to control shaft speed or torque of a three-phase induction motor. According to the reports in [13, 12], the company produces three main product series – for manufacturing industry, for HVAC (Heat, Ventilation and Air-conditioning), and for the water segment. In addition, the company provides many specialized products, e.g., for crane and specific textile applications. It employs 1200 individuals globally and approximately 200 in the R&D department. About 60 of them are embedded software developers, working in four development sites located in four different countries. Almost all of the functionality in a frequency converter is handled by software.

The main characteristics of the Danfoss development environment are summarized in the third column of Table 1. Product development is addressed by a matrix organization where projects are carried out by the integrated product development with dedicated personnel and target markets. Line organizations provide skilled developers for projects to draw upon to ensure application and domain knowledge for timely product releases. Exceptions are the testing organization which is "global" to all products, and an Embedded Software Platform team responsible for enforcing reuse.

The company follows an iterative development approach, with most artifacts being C++ code and test cases. Since late 1990s, the idea of managed reuse was promoted within the Danfoss development organization, resulting in creation of an object-oriented framework architecture written in C++. Yet, the evolution of the code base was not really controlled, and reuse between products was done in a "clone-and-own" way: code was taken from one project branch to another via merging in an SCM system.

In 2005, the company decided to migrate its recently launched new series of products to an SPLE approach, specifically, a configurable platform ("the 150% view") – a common code base in which variability points are selectable via compiler switches. The new series of products initially contained four cloned variants, two of which were selected for a pilot study. The code of the selected variants was analyzed to identify common and variable parts, and refactored into a common platform.

In the following years, the remaining two products of the new series were integrated into the common platform, and all additional products were developed on top of it. For the platform-managed

[1] http://www-01.ibm.com/software/awdtools/doors/
[2] http://www.esterel-technologies.com/products/scade-suite/
[3] DO-178B

Table 1: Analyzed Case Studies.

	Case Study #1	Case Study #2	Case Study #3
Domain	Aerospace	Electric motor controllers	Aerospace and Defense
Process	– V model (strictly waterfall) – Model-centric, with full requirements-to-code traceability – DO-178B certified	– Iterative – Code-centric	– Iterative – Code-centric – Requirements managed by a requirements management tool but no traceability to code is maintained – Requirements-based testing
Artifacts	– Textual requirements – Executable design models (code is generated) – Tests	– C/C++ code – Tests	– Textual requirements – C/C++ code – Tests
Transition Process	– Over 50 product variants before transition – Complex derivation graph – Six products as initial input to transition	– At least four product variants in the new series – Two products used as input to transition	– Five commercial product variants and a couple of prototypes before transition – Two products as initial input to transition
Target	Component library and a configurable framework architecture connecting the components	Configurable platform (150% view)	Component library
Status	Transition to SPLE in progress	Transition to SPLE completed	Transition to SPLE initiated

code, each product team could either use the code "as is" or create a branch from it with product-specific changes and additions. Products of other series were still maintained as distinct clones, with some being phased out.

At the time of writing, all products of the new series are derived from a shared platform in a feature-oriented manner. The company uses pure::variants[4] to manage its platform containing over 1100 features and hundreds of configurations used by customers.

2.3 Case study #3

The third case study, just like the first one, comes from a large aerospace and defense company (see the last column of Table 1). We analyzed a relatively recent software-intensive product line, developed over the last five years. It stems from two similar products which became successful and grew into a family containing five commercial products and a couple of in-development prototypes. Software is responsible for almost all of the functionality of these products.

The product line was developed by around 30 R&D professionals involved in different development stages – from requirements engineering to testing, with a centralized development team responsible for all products of the product line. Since the products are supportive rather than safety-critical, the company spends less time on regulatory issues, compared to the first case. For the same reason, design artifacts are rather informal and not always synchronized with code which is hand-written in C. The company uses IBM Rational Doors to maintain requirements. It complies to rigid testing procedures at all development stages including customer sites.

The product line management team indicated that they realized a they were developing a product line only after the success of the initial products, when customers started to ask for additional variants. The company made an effort to establish a library of reusable components early in the process, yet this library was often bypassed, and developers used cloning to rapidly serve their customers' needs. Moreover, together with the disadvantages of cloning, developers saw several advantages of the approach: it did not require any upfront investment; it was a rapidly available and easy to use practice, and it gave the developers freedom to make necessary changes in their code, without any need to synchronize their work with others [8]. For these reasons, the team mostly continued to clone.

With an increase in the number of products, a transition to SPLE received a higher priority. The company thus aims to inspect the existing implementations and identify reusable configurable components with well-defined interfaces. These components will then form a shared component library, allowing new products / projects to pick components from it. Each component should be independently testable so that the focus of the new product testing becomes integration testing. Thus, the library of components is anticipated to speed up both the development and the testing. Since the entire product line is developed by one centralized team, the team is able to redefine the development processes such that this time the library is put into use and kept up-to-date.

At the time of writing, an initial process of identifying commonalities and variabilities in the developed variants and building a library of reusable components has started.

3. TRANSITION TO SPLE

In this section, we describe the development activities that the companies performed as part of the transition process. We start by fixing some terminology (see also Figure 1).

We define a *variant* (a.k.a. *product*) as a well-formed set of *artifacts*, such as requirements, model elements and code statements. A variant's artifacts implement *features*. Inspired by Rajlich and Chen [6], we represent a product feature as a pair consisting of a *feature declaration (FD)* – a label and a short description that identifies the feature, and a *feature implementation (FI)* – a subset of product artifacts (requirements, model elements, code statements, etc.) that realize the feature declaration. A feature declaration is

[4]http://www.pure-systems.com/

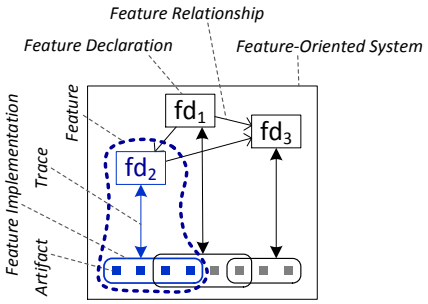

Figure 1: Notation.

traced to the feature implementation that realizes it. Features can have *relationships* with each other, e.g., one feature can depend on another in order to operate. A *feature model* is a set of feature declarations and relationships between them. A *feature-oriented system* (a.k.a. *system*) is a feature model and a set of artifacts traced to feature declarations from that model. A feature-oriented system can correspond to an individual product variant or a set of related products (a software product line).

In what follows, we use the above terminology to define a set of basic operators required for each of the case studies (see Table 2). Due to space limitations, we present a detailed analysis only of the first case study and a more brief analysis of the other two.

3.1 Case Study #1

The transition to SPLE proceeded in a top-down fashion, i.e., it started by first analyzing requirements documents, then creating a common architecture, and then building common assets, as schematically shown in Figure 2. At the time of writing, the architecture creation step has concluded and its verification is yet to commence. It terms of implementation, the created component libraries cover about 10% of the architecture. The assets were built incrementally using three major activities, described below.

Activity 1: Variability and commonality analysis.

This activity involves a comparison between a few variants in order to assess the variability scope and build an initial feature model.

1.1 Compare requirements documents.

As the first step, requirements documents are analyzed and compared to each other at the structural level. In this case study, all requirements documents follow the same structure: major capabilities are described in separate document sections which are grouped hierarchically. Each section contains a set of requirements statements. Every variant has its own requirements document. If a variant does not have a certain capability, the corresponding section in the requirements document for that variant is missing.

`findFD`: In effect, each document outline represents a feature tree of the corresponding variant, revealed using two conceptual operations. The first, captured by the `findFD` operator (line 1 in Table 2), returns the set of all feature declarations (*FD*s) realized by a given variant. In this case study, the implementation of the operator simply collects all sections in the corresponding document and treats them as feature declarations: section titles are treated as *FD* labels while section descriptions – as *FD* descriptions. The result has to be reviewed by a domain expert: while the majority of sections represent features, a few may not. Also, some section titles are rather long, and shorter feature labels have to be created.

`dependsOn`?: The second operation retrieves dependencies between feature declarations and is captured by the `dependsOn`? operator (line 3 in Table 2). There are several possible forms of such

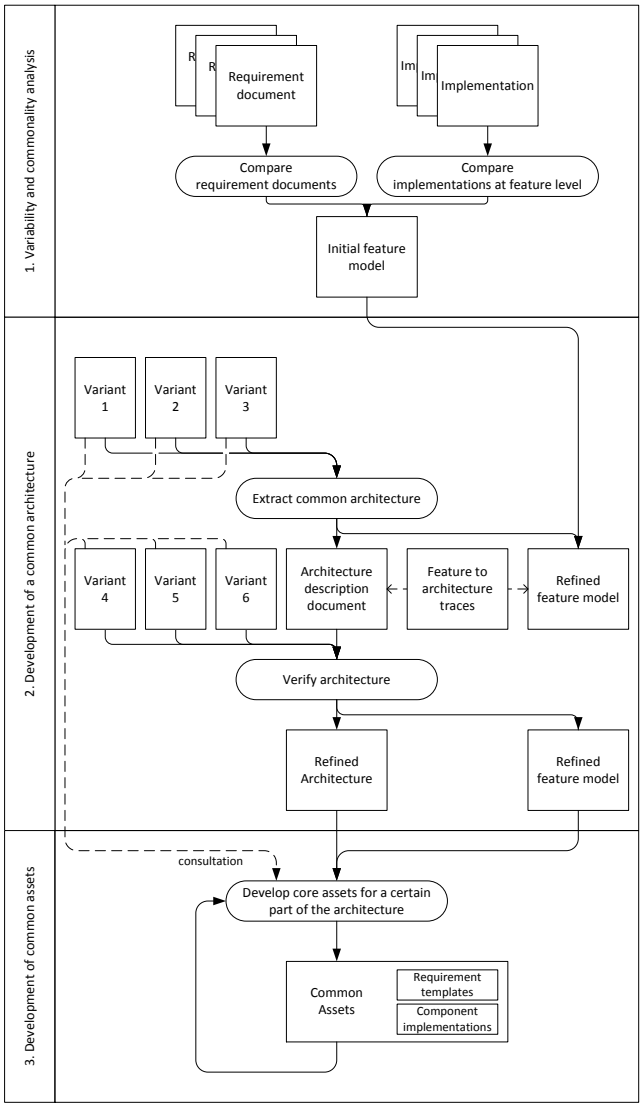

Figure 2: Transition Activities in Case Study #1.

dependencies, declaratively specified by the input *property* parameter. For example, one feature can require another in order to compile. Behavior dependencies, when one feature requires the other in order to operate correctly, are more complex. In our case, the input *property* of the operator is configured to determine feature declarations as dependent if their corresponding subsections have a parent / child relationship. The operator returns a *set of witnesses*, each demonstrating the `dependsOn` relationship between the artifacts of FD_1 and FD_2 (or *none* if the features are independent). In this case, a *witness* is just a pair of section numbers where one is a subsection of the other.

The result of applying these two operators is a tree of feature declarations containing mandatory features of each variant. Note that in this case study, the particular implementations of the two conceptual operators can easily be merged and optimized, to produce a candidate feature tree in one pass, but this is not always true in general.

`same`?: The requirements documents are further compared at the statement level to find pairs of features from distinct variants that are considered similar. Conceptually, this can be seen as an application of the `same`? operator (line 4 in Table 2), which determines whether two features are equivalent by considering their specifica-

Table 2: Operators for Managing Cloned Variants.

	Operator	Input	Output
1.	`findFD`	*variant*	*set of FDs*
2.	`findFI`	*variant* × *FD* × *property*	*FI*
3.	`dependsOn`?	$<FD_1, variant> \times <FD_2, variant> \times property$	*set of witnesses*
4.	`same`?	$<FD_1, variant_1> \times <FD_2, variant_2> \times property$	*set of witnesses*
5.	`interact`?	*set of* $<FD_i, variant_i> \times property$	*set of witnesses*
6.	`compose`	$system_1 \times \ldots \times system_n \times matches \times resolution$	*system*
7.	`reorganize`	*system*	$system'$

tions and implementations. Like `dependsOn`?, this operator uses a *property* that specifies equivalence criteria and returns a *set of witnesses* exemplifying the disagreements (or none if the features are equivalent).

In this case study, the implementation of `same`? matches feature declarations based on their lexical similarity (the Levenstein distance metric [18]) of their corresponding sections in the requirements documents as well as the individual statements of these sections. Feature declarations that correspond to sections containing a significant number of similar statements are considered similar as well. This implementation works well since in this case documents are mostly produced by cloning. Yet, the result still needs to be reviewed and refined by a domain expert.

1.2 Compare implementations.

`findFI`: Next, the design models of several variants are compared to each other. In this case, due to the regulatory constraints, every element of design models contains traceability links to the requirements it implements. Thus, establishing traceability between a feature declaration and its corresponding feature implementation – an instance of the `findFI` operator (line 2 in Table 2) – is trivial, and the *property* input parameter of `findFI` is simply configured to look for existing traceability links.

`same`?: Once the traceability is established, the feature declarations of distinct variants are compared again, this time considering design-level functions. This operation can be seen as a separate application of the `same`? operator. While conceptually both comparison operations could have been performed at once, in this case study they were executed separately, mostly because the comparison at the implementation level was done by manual inspection. We envision that this step can be automated using model matching techniques [36]. Regardless of how the implementation-level matching is done, its result still has to be reviewed by a domain expert and modified where necessary. For example, some of the differences can correspond to new, rather than already defined, feature declarations and hence should be lifted to the feature tree level.

1.3 Create an initial version of a feature model.

`compose`: This step involves merging the feature trees of the analyzed variants, unifying those declarations that are deemed similar. This can be seen as an instance of the conceptual `compose` operator (line 6 in Table 2): for n input systems, a set of *matches* between the corresponding input elements, and a *resolution* parameter defining how to handle conflicts that occur when input elements disagree. The operator produces a merged system. In our case, we are only interested in feature trees which are given to the `compose` operator as a parameter. The set of *matches* contains feature declarations decided by a process similar to the operator `same`?, and the *resolution* parameter specifies which variant to prefer if the two variants disagree. Again, while it is performed manually in this case study, we can envision automation of the operator, relying on existing works on feature model composition [1].

Activity 2: Development of a common architecture.

In this activity, three recent variants are analyzed in order to create a common architecture, define modules, interfaces and connections. The variants are picked so that they appear far apart in the cloning derivation graph and implement a diverse set of features, ensuring a sufficient scope coverage. The architecture is then validated using another three variants.

2.1 Create a common architecture.

`reorganize`: The chosen variants are first reorganized and "normalized" to create modules that cluster related artifacts together, usually by their functionality and also by applying certain architectural patterns. To support this task, we use the `reorganize` operator (line 8 in Table 2) which receives a system and a *property* that declaratively defines the nature of the required reorganization, as defined above. The operator returns a refined version of the system, after the reorganization has been performed.

When possible, the implementation of `reorganize` should ensure that the traceability between implementation artifacts and their corresponding feature declarations, established using `findFI`, is preserved. Alternatively, new traceability relationships are to be created. In this case study, the reorganization was performed manually while keeping the traceability relationships. Automation relying on existing model and code refactoring techniques [22] might also be possible and should be explored further. Yet, we do not envision a fully automated process but rather a set of automated techniques that assist the domain expert as necessary.

`compose`: Following the reorganization, the artifacts (both feature trees and implementations) of distinct variants are combined using `compose`, creating a candidate architecture and a refined feature model, with traceability between them. This step was also performed manually, involving architects with experience in building some of the previous variants. The outcome was a document describing the resulting architecture.

2.2 Verify the common architecture.

`reorganize`: The created candidate architecture and the feature model are further reviewed to validate their fitness and the ability to support additional three variants, distinct from those that were used as input to the previous step. This part of the review, conceptually captured by an instance of `reorganize` applied on the generated feature-based system, was also performed manually. This step re-

veals the need of applying the `reorganize` operator for both the cloned variants before the transition and the product line architecture after the transition, thus obtaining a *feature-oriented system* as a parameter rather than a concrete *variant*.

Activity 3: Development of common assets.

The development of common assets is done incrementally, with each increment covering a different part of the architecture. The created assets, which include requirements templates (text documents with placeholders) and component implementations, are intended to be used as libraries in the development of new variants.

`findFI`, `same`?: The activity relies on the ability to trace feature declarations from the generated feature model to their implementations in the analyzed variants, as established by the `findFI` operator, as well as the ability to compare and identify disagreements at the implementation level, using the `same`? operator. Then, existing variants are manually inspected and consulted during the development of the common assets: most of the assets are built from scratch by a domain expert to fit the more general context, to distill robust abstractions and ensure a high degree of modularity.

3.2 Case Study #2

In contrast to case study #1, the transition to SPLE in the second case proceeded in a bottom-up fashion, starting from code differences and working up to a feature model. The process was carried out incrementally, refining the common platform to improve its quality and modularity by refactoring along the way. The process involved four major activities, schematically shown in Figure 3 and described below.

Activity 1: Merge initial set of variants.

`compose`: As the first step, the implementations of two subsystems is compared to each other on the code level using a textual diff tool, and further unified. During the unification, conditional compilation directives with #`ifdef` PRODUCT_IS_XXX commands are introduced wherever the source code files disagreed with each other. This activity is conceptually represented by an instance of the `compose` operator, where the *matches* parameter is empty (i.e., only identical elements are considered similar) and the *resolution* is to insert conditional compilation directives that represent original variants.

`findFD`, `findFI`, `interact`?: The implementations of `findFD` and `findFI` are trivial: a feature declaration is created for each inserted directive and traced to its corresponding code. A *mutually-exclusive* relationship is defined between each pair of feature declarations that correspond to distinct variants. That provides a trivial resolution to the potential interaction of features that were not designed to work together. Feature interactions, ranging from purely syntactical to behavioral, are detected using the `interact`? operator (line 5 in Table 2) which obtains as input a *property* specifying the form of interactions to be checked.

As the result of applying the above operators, a *feature-oriented system* containing a common "150% view" representation of the code artifacts is created. The system contains a "primitive" feature model with *mutually-exclusive* features that correspond to input variants. This feature model drives the definition of two different makefiles for building the two original products from the unified code base.

Activity 2: Refactor to introduce meaningful features.

`reorganize`: In this activity, product-specific #`ifdef` statements are manually inspected by the domain expert and replaced with feature-specific statements #`if` HAS_FEATURE_XXX == 1. This activity, seen as an instance of the conceptual `reorganize` operator, includes refining the set of feature declarations, the traceability relationships between them, and the corresponding code.

In the same step, product-specific makefiles are also replaced by feature-specific counterparts: for each feature declaration, one makefile holds the list of artifacts to build when the feature is enabled, whereas the other holds the list of artifacts required when the feature is disabled (in most cases – an empty list).

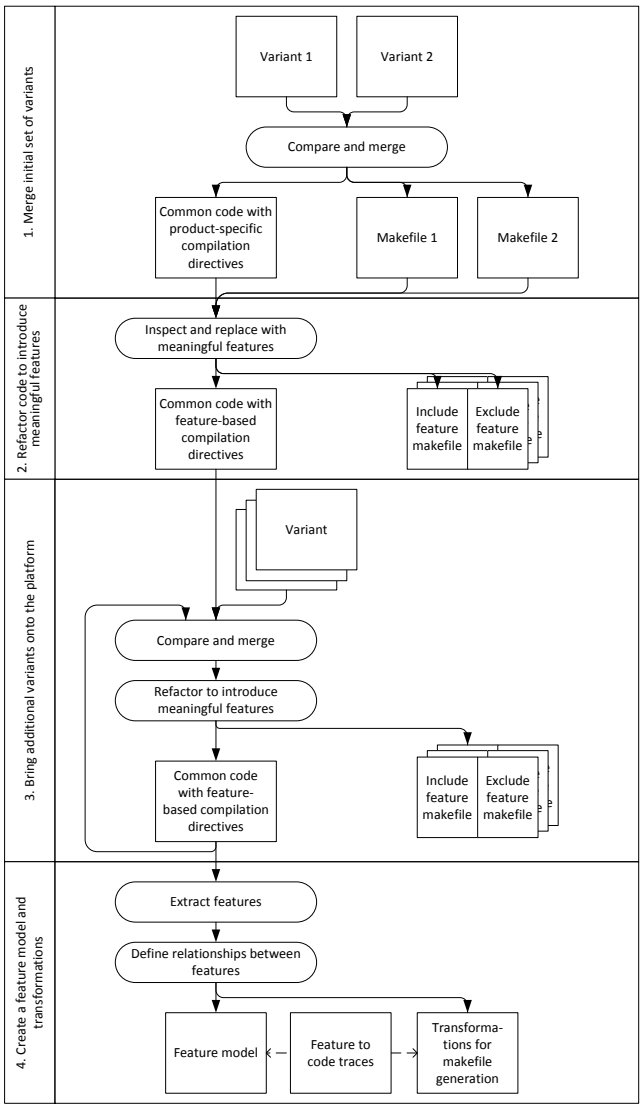

Figure 3: Transition Activities in Case Study #2.

Activity 3: Bring additional variants onto the platform.

The remaining variants are incrementally integrated into the constructed platform, one-by-one, until the full coverage is achieved. First, the code of a variant is combined with the existing platform, using product-specific #`ifdef` directives (via the operators `compose`, `findFD`, `findFI` and `interact`?, as described above). Later, the code is manually refined (via `reorganize`) to include feature-specific statements.

Activity 4: Create a feature model and transformations.

As the final step, compiler directives used to configure the code are extracted into a feature model that is managed by pure::variants.

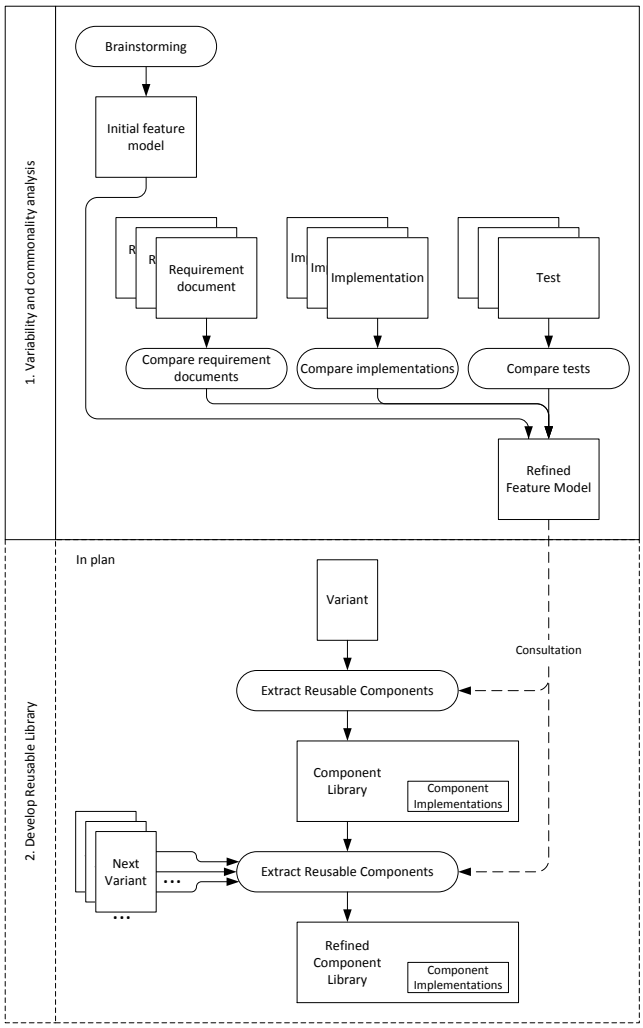

Figure 4: Transition Activities in Case Study #3.

4.1 Extract features.

findFD, findFI: This step can be seen as an instance of the findFD operator whose implementation trivially extracts the existing compilation directives. In the Danfoss case, the extracted directives were listed in a text file which was further imported to pure::variants. In this case, detecting traceability between the extracted feature declarations and the code that corresponds to them (the instance of the findFI operator) is also trivial.

4.2 Define relationships between features.

dependsOn?, same?, interact?, reorganize: Since the above process does not produce dependencies between features, the created feature model lists all features as optional. Refining it to include richer relationships such as grouping or alternatives conceptually relies on operators that can determine such relationships between feature declarations – dependsOn?, same?, interact?, as well as on the ability to reorganize the resulting feature model and improve its structure [34]. In the Danfoss case, this step was carried out manually by domain experts. Finally, transformations for creating makefiles for a specific feature configuration were developed using pure:variants, again, in a straightforward manner.

3.3 Case Study #3

In the third case, the transition process to SPLE has only just began. The company started from the analysis of the product portfolio and the existing product artifacts, aiming at capturing commonalities and variabilities among the current set of variants. Later on, the development of a reusable component library is planned, as shown in Figure 4. At this stage, all activities in this case study are performed manually, with the goal to identify and investigate automation opportunities. Thus, we do not discuss automation of the operators yet.

Activity 1: Variability and commonality analysis.

The goal of this activity is to create a feature model that scopes the product portfolio and explicates the list of supported features. Unlike the case study #1, there is no clear documentation listing the set of all existing feature declarations. Thus, the activity is performed in two steps.

1.1 Brainstorming.

findFD?: During a few brainstorming sessions, the development team members holding various organizational roles (i.e., managers, architects, developers and testers) proposed an initial set of feature declarations (findFD?), focusing mainly on capabilities that are perceived to be distinguishing between the developed product variants and those that customers use to describe the products. The set of elicited feature declarations is captured in a text document.

dependsOn?, same?, interact?: Identifying *requires* and *mutually exclusive* relationships between the found feature declarations is again performed manually based of the team's familiarity with the developed products. A *requires* dependency is introduced every time one feature depends on the presence of the other (as represented by the dependsOn? operator), while a *mutually exclusive* relationship is introduced every time features are not designed to work together (interact?) or implement functionality that is perceived similar (same?).

1.2 Comparing artifacts.

findFD: In this step, the initial set of feature declarations is further refined by inspecting development artifacts such as requirements documents, code and test descriptions. Artifacts are compared to each other manually and, similarly to the case study #1, disagreements "hint" at additional variation points, which become feature declarations. This is an instance of the findFD operator, performed manually.

dependsOn?, same?, interact?: Relationships between features are also detected manually. Conceptually, *require* dependencies between features (dependsOn?), distinct implementations of similar features (same?) and conflicting features (interact?) are considered. The set of the discovered feature declarations and relationships between them produce a version of a feature model.

Activity 2: Development of reusable library (proposed).

compose, reorganize: The company plans to build a library of reusable components in an incremental manner, by analyzing product variants one by one. Each analyzed variant either augments the library with additional components (an instance of the compose operator) or refines parameters of the components already in the library (an instance of compose followed by reorganize).

reorganize, findFI: After each iteration, the assets might be extended, refined and refactored (reorganize) in order to improve encapsulation and ensure that components can be usable in a diverse set of variants. The feature model is to be consulted at this stage to ensure that the components are easily configurable to provide the functionality perceived as important. If possible, traceability between feature declarations and the artifacts that implement them is to be established (findFI).

4. COPING WITH CLONES

The transition from cloning to SPLE is an incremental process which might easily take several years, as in case study #2. In case study #3, the transition is only in its initial stage, so immediate customer needs still have to be addressed by cloning. Moreover, in all three case studies considered here, the created SPL architecture only targets future products, while existing ones are still maintained as distinct clones.

In this section, we focus on the problem of maintaining existing cloned product variants. In most cases, little automation is available to support the required maintenance activities [25]. In fact, in our case studies, such activities are performed manually. We thus discuss these activities collectively, mapping them to instances of the conceptual operators in Table 2.

Activity 1: Propagating changes between variants.

`findFD`, `findFI`, `same`?: Changes made in one cloned variant might be useful in another. To locate such changes, correspondences between feature declarations of a variant (detected using `findFD`) and the artifacts that implement them (detected using `findFI`) are established. Differences between distinct implementations of the same feature declaration (detected using `same`? and represented by a set of *witnesses*) are inspected and propagated between variants. In the simplest form, the differences can be detected using a textual difference tool, as was done in case study #2. Detecting more sophisticated behavioral differences is also possible, e.g., using the technique in [11].

Activity 2: Sharing features between variants.

`findFD`, `findFI`: Like individual changes, complete features can be shared between distinct product variants. Here, again, a list of feature declarations, together with traces to implementation-level artifacts, is identified and maintained (instances of `findFD` and `findFI`).

`same`?, `dependsOn`?: Different implementations of the chosen feature of interest are compared to each other (using `same`?) selecting the one found most appropriate. Further, the set of other features it requires (detected using `dependsOn`?) is inspected. If those features are not part of the target product, some of their artifacts have to be transferred to the target product together with the selected feature, to ensure its correct operation, as discussed in [29].

`interact`?, `compose`: Next, the `interact`? operator verifies whether the new feature interferes with the functionality of the existing ones in the target product variant. Following that, `compose` integrates the selected feature and those that it requires in the target system, resolving the conflicts identified by `interact`?.

Activity 3: Retiring features.

`findFD`, `findFI`, `dependsOn`?: While new features are added, some of the existing features might no longer be needed. Like in the previous activities, the set of feature declarations and their corresponding implementations is detected (using `findFD` and `findFI`) and features that depend on the one being removed are identified (using `dependsOn`?). Since the functionality of such features should not be affected by the feature retirement, artifacts that these features use are not removed.

Activity 4: Establishing new variants.

`findFD`, `same`?, `dependsOn`?, `interact`?, `compose`: Depending on the maturity of the transition process, there might still be a need to create new variants following the existing cloning practices, like in the case study #3. In such cases, the feature portfolio of all existing variants (detected using `findFD`) is inspected, and the variant with the most similar functionality is used as a starting point for cloning. Then, features that are not required in this variant are removed, as described in Activity 3, while additional features are either developed from scratch or "borrowed" from other variants, as described in Activity 2.

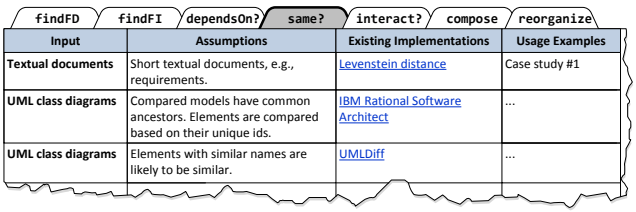

Figure 5: An Initial Sketch of the Knowledge-Based Library.

5. TOWARDS BUILDING THE BODY OF KNOWLEDGE

Our empirical analysis of the case studies demonstrated the applicability of the operators and their ability to support a variety of development activities related to the management of cloned product variants, as summarized in Table 3. By focusing on the operators, we broke processes down into well-studied logical components, thus promoting the reuse of existing component implementations when dealing with cloned product variants.

For example, the implementation of the `same`? operator can be based on analyzing lexical similarities between textual documents, e.g., using the Levenstein distance metric [18], as was done in case study #1. This implementation is applicable to short and unstructured text, such as requirements descriptions. When comparing design artifacts (i.e., models of different types), a variety of model comparison and matching techniques can be applied. These are surveyed in depth in [36]. Comparison techniques for code artifacts have also been studied extensively. These can rely either on textual diff tools, e.g., [20], as in case study #2, on tools that attempt to detect semantic differences, e.g., [11], or on more sophisticated implementations based on code clone detection [3].

Numerous implementations of `findFI` (a.k.a. feature location) have been developed (see [28] for a survey). Feature interaction techniques, implementing `interact`?, have also received a lot of attention, especially in the telecommunications domain [40].

We believe that the operator-based view enables a better understanding of existing implementations and their applicability. Ultimately, creating a library of possible implementations for each operator, together with the assumptions that the implementation makes and the properties it supports, can facilitate an efficient management of cloned variants and enable reuse across organizations and domains. A sketch of a vision for such a library in shown in Figure 5. "Crowdsourcing" of existing implementations can eventually lead to an increased quality and a larger spectrum of available solutions.

6. RELATED WORK

Several works [10, 2, 15, 14, 16] capture guidelines and techniques for manually transforming legacy product line artifacts into SPLE representations. Some also introduce automatic approaches to reorganize product variants into annotative representations [17, 21, 31, 27]. Works on feature-oriented refactoring [19, 23] focus on identifying the code for a feature and factoring the code out into a single module or aspect aiming at decomposing a program into features. Our work differs from those as we do not propose a specific

Table 3: Applicability of the Operators.

		findFD	findFI	dependsOn?	same?	interact?	compose	reorganize
Case 1	1. Variability and commonality analysis.	✓	✓	✓	-	-	✓	-
	2. Development of a common architecture.	-	✓	-	-	-	✓	✓
	3. Development of common assets.	-	✓	-	✓	-	-	-
Case 2	1. Merge initial set of variants.	✓	✓	-	-	✓	✓	-
	2. Refactor to introduce meaningful features.	-	-	-	-	-	-	✓
	3. Bring additional variants onto the platform.	✓	✓	-	-	✓	✓	✓
	4. Create a feature model and transformations.	✓	✓	✓	✓	✓	-	✓
Case 3	1. Variability and commonality analysis.	✓	-	✓	✓	✓	-	-
	2. Development of reusable library.	-	✓	-	-	-	✓	✓
Cloning Activities	1. Propagating changes between variants.	✓	✓	-	✓	-	-	-
	2. Sharing features between variants.	✓	✓	✓	✓	✓	✓	-
	3. Retiring features.	✓	✓	✓	-	-	-	-
	4. Establishing new variants.	✓	✓	✓	✓	✓	✓	-

refactoring approach or technique but rather capture and classify common tasks required during such refactoring.

Other authors also looked at systematic classifications of programming tasks: Chen and Rajlich [6] identified six fundamental program comprehension operators that trace feature label, description and implementation to each other. We incorporated some of them into our work, as these operators are also applicable in the context of cloned product variants. However, our main focus was on cases of *multiple* variants rather than single-copy systems and involved *manipulations* on the variants rather than only *comprehension* activities. Borba et al. [4] suggested a theory of product line refinement. This is a special case of variant management – the more general problem that we consider here. She et al. [33] classified several software re-engineering scenarios involving feature model synthesis. These can be seen as detailed scenarios for our `findFD` and `dependsOn`? operators, while our work has a broader scope. Brunet et al. [5] identified model merging operators and specified their algebraic properties. Our work is not limited to models and considers a broader set of necessary maintenance activities, focusing specifically on cloned product variants.

7. CONCLUSIONS AND FUTURE WORK

Software Product Line Engineering is gaining increasing popularity in industry due to its promises to enable a significant improvement in time-to-market and quality, reduction in engineering costs, increase in portfolio size, and more. However, in reality, many companies still employ cloning to realize their product variants. In this work, we conducted an empirical study involving three such companies and analyzed in detail the development activities these companies perform.

We broke the activities down into individual clone management operators and showed that our operators support both the case when a company transitions to a structured SPL-based artifact management approach and the case when existing clone variants are maintained as is. Taking the operator-based view allowed us to study and organize the landscape of development tasks for managing cloned variants as well as to map these tasks to a broad set of existing solutions.

While one can come up with a different set of domain-specific operators that closer reflect the necessary development activities for each case and exploit the increased degree of specialization, we believe that the generic operator-based view leads to more efficient development and maintenance practices. Specifically, it allows organizations to locate and reuse existing work as well as to share experiences with each other using a common vocabulary.

This is a first step in exploring the space of clone management operators. While we showed that the current set is reasonable, it is most likely incomplete. In the future, we aim to refine the set of operators and their interfaces. Moreover, identifying and classifying contexts in which the operators are automatable as well as quantifying the cost of providing such automation is still required. "Smart" implementations of the operators that can work incrementally, to help address incremental changes in the development artifacts, would also be very useful.

We invite the product line community to join our efforts and further improve the framework by refining the set of the operators, studying their applicability and organizing existing work around the established common terminology. We believe that such an effort will assist practitioners, by allowing a greater generality of what the field offers, as well as solution developers, by systematizing and organizing the support that is to be provided.

8. REFERENCES

[1] M. Acher, P. Collet, P. Lahire, and R. France. Comparing

Approaches to Implement Feature Model Composition. In *Proc. of ECMFA'10*, pages 3–19, 2010.

[2] J. Bayer, J.-F. Girard, M. Würthner, J.-M. DeBaud, and M. Apel. Transitioning Legacy Assets to a Product Line Architecture. In *Proc. of FSE'99*, pages 446–463, 1999.

[3] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE TSE*, 33:577–591, 2007.

[4] P. Borba, L. Teixeira, and R. Gheyi. A Theory of Software Product Line Refinement. *Theoretical Computer Science*, 455:2–30, 2012.

[5] G. Brunet, M. Chechik, S. Easterbrook, S. Nejati, N. Niu, and M. Sabetzadeh. A Manifesto for Model Merging. In *Proc. of GaMMa'06*, pages 5–12, 2006.

[6] K. Chen and V. Rajlich. Case Study of Feature Location Using Dependence Graph. In *Proc. of IWPC'00*, pages 241–249, 2000.

[7] P. C. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[8] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki. An Exploratory Study of Cloning in Industrial Software Product Lines. In *Proc. of CSMR'13*, 2013.

[9] D. Faust and C. Verhoef. Software Product Line Migration and Deployment. *J. of Software Practice and Experiences*, 30(10):933–955, 2003.

[10] S. Ferber, J. Haag, and J. Savolainen. Feature Interaction and Dependencies: Modeling Features for Reengineering a Legacy Product Line. In *Proc. of SPLC'02*, pages 235–256, 2002.

[11] D. Jackson and D. A. Ladd. Semantic Diff: A Tool for Summarizing the Effects of Modifications. In *Proc. of ICSM'94*, pages 243–252, 1994.

[12] H. P. Jepsen and D. Beuche. Running a Software Product Line: Standing Still is Going Backwards. In *Proc. of SPLC'09*, pages 101–110, 2009.

[13] H. P. Jepsen, J. G. Dall, and D. Beuche. Minimally Invasive Migration to Software Product Lines. In *Proc. of SPLC'07*, pages 203–211, 2007.

[14] K. C. Kang, M. Kim, J. Lee, and B. Kim. Feature-oriented Re-engineering of Legacy Systems into Product Line Assets. In *Proc. of SPLC'05*, pages 45–56, 2005.

[15] K. Kim, H. Kim, and W. Kim. Building Software Product Line from the Legacy Systems: Experience in the Digital Audio and Video Domain. In *Proc. of SPLC'07*, pages 171–180, 2007.

[16] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi. Refactoring a Legacy Component for Reuse in a Software Product Line: a Case Study: Practice Articles. *J. of Software Maintenance and Evolution*, 18(2):109–132, 2006.

[17] R. Koschke, P. Frenzel, A. P. Breu, and K. Angstmann. Extending the Reflexion Method for Consolidating Software Variants into Product Lines. *Software Quality Control*, 17(4):331–366, 2009.

[18] V. I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10, 1966.

[19] J. Liu, D. Batory, and C. Lengauer. Feature Oriented Refactoring of Legacy Applications. In *Proc. of ICSE'06*, pages 112–121, 2006.

[20] D. MacKenzie, P. Eggert, and R. Stallman. *Comparing and Merging Files with GNU diff and patch*. Network Theory Ltd., 2003.

[21] T. Mende, R. Koschke, and F. Beckwermert. An Evaluation of Code Similarity Identification for the Grow-and-Prune Model. *J. of Soft. Maintenance and Evolution*, 21(2):143–169, 2009.

[22] T. Mens and T. Tourwé. A Survey of Software Refactoring. *IEEE TSE*, 30(2):126–139, 2004.

[23] G. C. Murphy, A. Lai, R. J. Walker, and M. P. Robillard. Separating Features in Source Code: an Exploratory Study. In *Proc. of ICSE'01*, pages 275–284, 2001.

[24] K. Pohl, G. Boeckle, and F. van der Linden. *Software Product Line Engineering : Foundations, Principles, and Techniques*. Springer, 2005.

[25] B. Ray and M. Kim. A Case Study of Cross-System Porting in Forked Projects. In *Proc. of FSE'12*, 2012.

[26] J. Rubin and M. Chechik. A Framework for Managing Cloned Product Variants. In *Proc. of ICSE'13 NIER track*, pages 249–252, 2013.

[27] J. Rubin and M. Chechik. Quality of Merge-Refactorings for Product Lines. In *Proc. of FASE'13*, pages 83–98, 2013.

[28] J. Rubin and M. Chechik. A Survey of Feature Location Techniques. In I. Reinhartz-Berger et al., editor, *Domain Engineering: Product Lines, Conceptual Models, and Languages*. Springer, To appear.

[29] J. Rubin, A. Kirshin, G. Botterweck, and M. Chechik. Managing Forked Product Variants. In *Proc. of SPLC'12*, pages 156–160, 2012.

[30] U. Ryssel, J. Ploennigs, and K. Kabitzsch. Automatic Variation-Point Identification in Function-Block-Based Models. In *Proc. of GPCE'10*, pages 23–32, 2010.

[31] U. Ryssel, J. Ploennigs, and K. Kabitzsch. Extraction of Feature Models from Formal Contexts. In *Proc. of SPLC'11*, pages 4:1–4:8, 2011.

[32] D. Schmidt. Guest Editor's Introduction: Model-driven engineering. *IEEE Computer*, 39(2):25, 2006.

[33] S. She, K. Czarnecki, and A. Wasowski. Usage Scenarios for Feature Model Synthesis. In *Proc. of VARY'12*, pages 13–18, 2012.

[34] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. Reverse Engineering Feature Models. In *Proc. of ICSE'11*, 2011.

[35] M. Staples and D. Hill. Experiences Adopting Software Product Line Development without a Product Line Architecture. In *Proc. of APSEC'04*, pages 176–183, 2004.

[36] M. Stephan and J. R. Cordy. A Survey of Methods and Applications of Model Comparison. Technical report, Queen's University, Kingston, Canada, 2011.

[37] C. Thao, E. Munson, and T. Nguyen. Software Configuration Management for Product Derivation in Software Product Families. In *Proc. ECBS'08*, pages 265–274, 2008.

[38] J. van Gurp and C. Prehofer. Version Management Tools as a Basis for Integrating Product Derivation and Software Product Families. In *Proc. of VaMoS'06*, pages 48–58, 2006.

[39] K. Yoshimura, F. Narisawa, K. Hashimoto, and T. Kikuno. FAVE: Factor Analysis Based Approach for Detecting Product Line Variability from Change History. In *Proc. of MSR'08*, pages 11–18, 2008.

[40] P. Zave. FAQ Sheet on Feature Interaction. `http://www2.research.att.com/~pamela/faq.html`, 2004.