Variability Evolution and Erosion in Industrial Product Lines - A Case Study

Bo Zhang
Software Engineering Research Group
University of Kaiserslautern
Kaiserslautern, Germany
bo.zhang@cs.uni-kl.de

Martin Becker,
Thomas Patzke
Fraunhofer Institute Experimental Software Engineering (IESE)
Kaiserslautern, Germany
{martin.becker,thomas.patzke}@iese.fraunhofer.de

Krzysztof Sierszecki,
Juha Erik Savolainen
Danfoss Power Electronics A/S
Global Research & Development
Graasten, Denmark
{ksi, JuhaErik.Savolainen}@danfoss.com

ABSTRACT
Successful software products evolve continuously to meet the changing stakeholder requirements. For software product lines, modifying variability is an additional challenge that must be carefully tackled during the evolution of the product line. This bears considerable challenges for industry as understanding on how variability realizations advance over time is not trivial. Moreover, it may lead to an erosion of variability, which needs an investigation of techniques on how to identify the variability erosion in practice, especially in the source code. To address various erosion symptoms, we have investigated the evolution of a large-scale industrial product line over a period of four years. Along improvement goals, we have researched a set of appropriate metrics and measurement approaches in a goal-oriented way, applied them in this case study with tool support, and interpreted the results including identified erosion symptoms.

Categories and Subject Descriptors
D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement – *Restructuring, reverse engineering, and reengineering.* D.2.8 Metrics – *complexity measures.* D.2.9: Management – *software quality assurance.* D.2.13: Reusable Software – *domain engineering.*

General Terms
Experimentation, Measurement, Documentation.

Keywords
Product Line Evolution, Variability Erosion, Static Code Analysis, Industrial Case Study.

1. INTRODUCTION
The goal of software product line (SPL) engineering is to create similar products in a cost-efficient way so that their common and varying characteristics are fully utilized. Ideally, a SPL should contain a majority of commonality, which is shared and reused all products, and a minority of variability, that is currently required by existing products. However, trying to maximize the commonality tends to fail in evolution. When being exposed to a number of products, application engineers tend to invent new combinations of features, new improvements to existing features, and completely new features that are difficult to predict during domain engineering. This drives the increase of variability in SPL evolution to allow matching the changing development requests.

On the downside, the newly added variability makes the variability realization in the product line more complex, and the increased complexity tends to reduce the overall SPL productivity. If no corrective actions are done, the SPL becomes less competitive and new products will eventually start to use solutions beyond the product line core or effort to create a new product line will be started. Based on our practical experience, it is often the better choice to evolve an existing asset base than try to create a new product line, especially in the light of long-lived products. However, there is a *lack of methods and tools to support efficient SPL evolution* in current industrial practice, e.g. in analyzing existing asset bases to detect erosion problems and planning and executing necessary countermeasures. Also a lot of research results focus on engineering new product lines rather than supporting evolving existing product lines to target new requirements.

Once SPL evolution is not conducted appropriately, variability realizations will gradually erode over time. The concept of erosion was introduced by Perry and Wolf in [18] to indicate software architecture problems. In our context of SPL evolution, erosion means that realization artifacts become overly complex due to unforeseeable changes (also known as software aging [15]). Eroded variability realizations comprise overly scattered, fine-grained Variability Elements (VEs) with complex interdependencies. They lead to practical problems in evolution as it becomes more difficult to assess change impact on core assets and to adapt the change with other related VEs [12] in a consistent manner. This increases the risk for missing, obsolete or incorrect variability realizations in core assets. Besides change propagation, the effort of quality assurance after change is also non-trivial, especially when the products fall under some regulation. If a code change involves multiple independent variable features and each feature has multiple optional values, then it will cause tremendous quality-assurance effort because all possible configurations have to be checked.

The overly complex variability realizations cause a serious impact on productivity of SPL maintenance, and put the expected SPL advantages more and more at risk. Detecting and fixing such problems is difficult because these problems do not only involve the current SPL, but often emerge in the past and develop along

SPLC 2013, August 26 - 30 2013, Tokyo, Japan

ACM 978-1-4503-1968-3/13/08 $15.00.

the SPL evolution history. The *goal of this paper* is to raise awareness for problems and solution strategies in the context of SPL evolution, and improve the necessary support in industrial practice. To this end, we have investigated the evolution of a large-scale industrial product line over a *period of 4 years* in this case study. The underlying business motivation for this is to sustain the productivity of the evolving product line, i.e. to keep it on a sufficiently good level while meeting economic constraints. At the heart this requires to avoid an overly (i.e. more than necessary) decrease of productivity. Driven by improvement goals, we have researched a set of appropriate metrics and measurement approaches, applied them on *31 versions* of the product line infrastructure analyzed the huge data set and discussed the main findings.

In this paper we provide the following contributions:

1. We present industrial experience with product line evolution, taken improvement steps and their actual effects.
2. We discuss analysis goals in the context of PL evolution.
3. We derive a set of metrics for identifying and classifying variability implementation via Conditional Compilation.
4. We introduce the novel concept of Variation Point Group (VPG) that helps to understand the alignment of VEs between problem space and solution space.
5. We describe how to collect the metrics in a pragmatic way with our tool suite.
6. We present some of the data, which we have collected on 31 versions of the evolving product line.
7. We analyze variability evolution based on obtained data, especially the erosion of variability implementation over time.

The paper is organized as follows: Section 2 provides the background knowledge of the state of art and practice in SPL evolution, and investigates open issues and related research work. Section 3 introduces the SPL development and organization in Danfoss Power Electronics. Section 4 gives an overview of our evolution analysis approach, which includes a GQM model and various variability measurements on the code level with tool support. Section 5 presents the case study following the analysis approach, where different variability erosion issues are investigated in detail. Main findings in the text are emphasized and indexed with F-n. Finally the conclusion and future work is presented in section 6.

2. BACKGROUND AND RELATED WORK

This section provides the necessary background for the paper. It presents general findings of SPL evolution in practice and introduces related research work.

2.1 SPL Evolution in Practice

Over time, variability tends to continuously increase in successful industrial product lines. More and more features are added and more products are instantiated from the product line infrastructure. This continuously increases the number of configurations in which the core assets belong to. This tends to lead into culture where variability is only added but never removed. When complexity grows it is hard to know if one can actually remove a variation point. However, since the continuously added variability significantly contributes to the increasing complexity, ability to remove variability is a key enabler for successful product line evolution [23].

Nowadays SPL development is often conducted in an incremental way, in which the variability artifacts evolve both in space and in time [10]. During evolution the variability supported by the SPL infrastructure is changing over time, which requires iterative development and maintenance of variability specifications and realizations. However, experience in evolving SPLs shows that variability specifications and realizations erode in the sense that they become overly complex and inconsistent with each other. Along this line, Livengood addressed in [12] the practical challenge that if there is a change in the variability model, then it is often very difficult to assess the change impact on the downstream development artifacts. The reason is that as a SPL evolves, the variability model is not traceable to (probably also inconsistent with) the variability realizations.

Experience from industry [2][16] shows that variation points (e.g., #ifdef blocks in C/C++ code) and their interdependencies are often inexplicit, and such information is difficult to detect from code manually. As a SPL evolves, there might be obsolete or even erroneous code in core assets, and fixing such problem is time-consuming. This causes a serious impact on SPL productivity and puts the expected SPL advantages more and more at risk. Refactoring the core assets would be appropriate and feasible, but too expensive if done manually. Sometimes there are several code refactoring activities possible, but it remains unclear what is the best one because the respective methods and tools are still in infancy.

2.2 Related Research

Many research publications have been concerned with product line evolution. Product line evolution categories and their interdependencies in different product line engineering phases have been studied by Svahnberg and Bosch [28]. In contrast to the current paper, most of the proposed scenarios are not specific to product lines, and in particular are not specific to variability. McGregor claimed in [14] that the difference between evolution of single systems and evolution of products in product lines is that for single systems, anticipated evolution is possible and unanticipated evolution is very likely, whereas for product lines, anticipated evolution is very likely and unanticipated evolution is less likely. We do not distinguish between anticipated and unanticipated evolution, that is, all evolution is seen as unanticipated, a view consistent with non-proactive product line methods [10]. Elsner et al. investigated in [4] the different notions of evolution, or variability in time, in product line engineering. Three different categories of variability in time were found (variability of linear change over time, multiple versions at a point in time, and binding time over time).

Passos et al. [20] also addressed the challenge of understanding software evolution and hypothesized that changes can be effectively managed in a feature-oriented manner. They envisioned organizing software evolution by automatically recovering traceability links among features and their realization artifacts. Based on the traceability links, they intended to conduct analysis for making strategic decisions and to provide further recommendations for feature refactoring. However, to the best of our knowledge automated traceability recovery is technically still an open issue, especially when features are not fully modularized or implicit in code. In this paper, we mitigate this challenge by automatically extracting VEs and their interdependencies which are implemented by C-preprocessor (CPP) code.

Regarding the analysis of CPP code, Liebig et al. [11] investigated the issues of #ifdef nesting, tangling, and scattering in forty variability-aware systems. Although these systems evolve in diverse application domains, they are all open source

systems and might not reflect the representative setting and corresponding issues in the industry. Moreover, in their analysis any macro constant used in #ifdef expressions is considered as a feature name. However, from our experience most #ifdef blocks (e.g., 87.6% in the Danfoss SPL) are actually not variability-related, but for other purposes such as include guards or macro substitution. This might be another threat to external validity. In our case study, all variabilities follow a special naming convention, which enables us to extract the variable code accurately and exclusively.

In our previous study [32], we analyzed variability-related CPP code in both core assets and product realizations, and extract a realization variability model including variabilities, variation points, and a graphical tree of variation points based on #ifdef nesting. Moreover, we also presented a feature correlation mining approach [33] to identify implicit variability interdependencies from existing product configurations. Both of these two approaches help to understand complex variability realizations in a reverse-engineering style. Although we conducted quantitative measurement on extracted VEs as well as their interdependencies, the analyses were only on one SPL version, thus it is not clear how these VEs are evolved over time. Besides, neither the variability impact on code nor the variability complexity in each file was considered in our previous work.

3. DANFOSS POWER ELECTRONICS SPL

Danfoss Power Electronics develops frequency converters known as VLT® drives, to control speed, torque, acceleration, synchronization, positioning, and the overall performance of electrical motors. Software coming along with the hardware is a significant asset that provides the needed variability in order to realize a wide range of possible applications. Almost any functionality in a frequency converter is now handled by corresponding software block. This allows for great flexibility and ease of reprogramming, but unfortunately, it comes at the expense of higher complexity.

It becomes apparent that due to the increasing variability induced by specific customer needs, it becomes too complex, too time consuming, and too expensive to build new products using the "clone-and-own" approach, or from "scratch". That's why Danfoss introduced SPLs in their engineering process in 2006 [6][7]. The product line has been quite successful in providing product variants that met the market demands: with a shared code base of a few MLOC and a large number of features it supports over 15 main and 30+ optional product variants. An internal study has revealed that the number of defects has been reduced by more than 30%, and the defect severity has decreased.

The overall organizational structure of the development department at Danfoss Power Electronics has been project-centric for many years. Product development is addressed by a matrix organization where projects are carried out through integrated product development with dedicated personnel provided by line organizations. In order to improve the coordination between projects, a Platform Coordination team has also been established with participation from each project together with the platform release manager who acts as chairman. The coordination team has monthly meetings where defects and change requests are coordinated between the projects.

At the beginning, the main focus was limited to migrating software assets to a common platform. In the first attempt independent products, derived by the "clone-and-own" approach, were unified into a single set of jointly usable information to achieve a unified code base. Initially, the variation points were based on products, since it was a quick way to extract them, however later they were refactored into variable features: so-called HAS_FEATUREs. These are used to form a feature model representing actual variability and implement variation points as CPP statements in C/C++ source code.

The HAS_FEATURE mechanism is used to implement various kinds of variability in space: the canonical features representing functionality provided by certain subsystems, and variable features configuring the subsystems as a result of different customer needs and discovered defects. Over time, HAS_FEATUREs scope may also change: from product-specific, application-specific, to finally become a common part of higher-level feature, so being removed. E.g. defect fixes will follow this path, nevertheless, it must be admitted that adding variability is far more usual that removing it.

Development of HAS_FEATUREs is well defined: rules how to establish them are given, variation mechanism is described, naming convention for various types of features is provided, and examples of do's and don't's guiding the development are presented. Software engineers have been trained, feature development is reviewed, and yet, we have found out that automated checks must be made in order to ensure feature implementation follows the HAS_FEATURE specification.

4. ANALYSIS APPROACH

The underlying business motivation of this case study is to sustain the productivity of the evolving product line, i.e. to keep it on a sufficiently good level while meeting economic constraints. At the heart this requires to avoid an overly (i.e. more than necessary) decrease of productivity. One of the main root causes to this end is the "overly erosion" of the variability-related part of the product line architecture. We call this *variability erosion* in the following. In order to cope with the engineering challenge of variability erosion, there are 4 basic *tactics* [1]:

1. **Removal** – to reduce erosion in the current version;
2. **Tolerance** – to minimize the impact of erosion without reducing it;
3. **Forecasting** – to predict future incidence and likely consequence of erosion;
4. **Prevention** – to prevent further erosion in future versions.

In this paper, we focus on erosion detection and forecasting. While detection serves as a prerequisite of removal and tolerance, forecasting forms the basis of prevention.

Along our PuLSE-E method [17] we have identified typical problems, root causes, and code smells for variability erosion in a previous study [16]. However, in the beginning of this case study it was unclear which metrics could be used to detect erosion symptoms in an automated way in the data set and how to interpret the results. Given the various VEs and their interdependencies, a plethora of different metrics can be easily measured from SPL realizations. Among them are fan-out, fan-in, cyclomatic complexity, nesting levels on the different VEs and their change over time, to name a few. While some of them might be helpful indicators for variability erosion, others are not.

In order to conduct the analysis systematically and concentrate on helpful metrics, we first have investigated the used variability realization mechanisms for typical erosion problems and then have followed the GQM approach [26] to get a clear

understanding of the analysis goals and derive corresponding metrics from them. In the following, we present main findings of the the respective activities.

4.1 Variation Realization Mechanism

In variability realizations, Conditional Compilation (CC) is a frequently used mechanism to enable or disable variant code [5] [17]. To be specific, macro constants are defined in product configurations and used in #ifdef statements (in this paper #ifdef also subsumes other similar directives such as #if, #ifndef, and #elif) in code core assets. Based on the defined values of macro constants, a C-preprocessor adapts the core assets by enabling or disabling enclosed code fragments in an intuitive way. However, the usage of CPP code also brings negative effects. Liebig et al. [11] analyzed the CPP code of forty open source SPLs and addressed the issues of #ifdef nesting, tangling and scattering. These issues obfuscate the layout of the common code, making the core assets overly complex and difficult to understand.

Danfoss is using CC as the main mechanism to realize variabilities (i.e. HAS_FEATUREs), specified in pure::variants [21], in their code core assets. CC is also used for other reasons [16], e.g. definition of constants, include guards, macro substitution, which actually yields the majority of the CPP statements (*about 87.6% in this Danfoss SPL*).

F1. In order to analyze variability realizations, it needs to be distinguished, whether the CPP directives are used for the realizations of variabilities or something else. This complicates the analysis in general and typically requires extensive involvement of domain knowledge.

In the case of Danfoss, the clear and strictly followed naming convention allowed an automatic classification without expert involvement. With that, variable features can be traced from the problem space into all respective variations points in the solution space easily. By analyzing the structure of CPP statements, also the size and nesting of variation points can be analyzed automatically.

F2. Naming conventions like "HAS_FEATURE" help to identify variation points automatically, if they are strictly followed.

One typical problem of variability erosion is the proliferation of variant elements over time [9]. In this case, the realization of a variable feature scatters over several modules, files, and variation points [8]. Further problems are variation combinatorics and unintended interdependencies of variant elements. A cursory inspection of used realization constructs revealed that the majority of variation points is realized by means of #ifdef statements. A considerable number of them use expressions that depend on two or more variabilities, which is called #ifdef tangling in [11]. As a consequence, the relation between variabilities and variation points is not 1:n but m:n. There are two interesting aspects in this: 1) the combined usage of HAS_FEATUREs can result in fairly complex expressions, which are difficult to understand; and 2) the usage of recurring expressions is an indicator that the structure of the variability problem space and solution space are misaligned.

Therefore, we have decided to consider three types of variability concepts in our analysis as illustrated in Fig. 1, i.e., Variability (Var), Variation Point (VP), and Variation Point Group (VPG). A Var represents a variable feature in the problem space and is realized in VPs (solution space) to enable or disable enclosed code fragments. In CC a VP is a #ifdef block. Moreover, VPs with logically equivalent #ifdef expressions are clustered as a VPG because they have the same effect on their enclosed code fragments. As shown in Fig. 1, each Var can be used in multiple VPGs, while each VPG may contain multiple VPs. These VPs may be scattered in either one or multiple code files.

F3. The concept of VPG helps to understand the alignment of VEs between problem space and solution space.

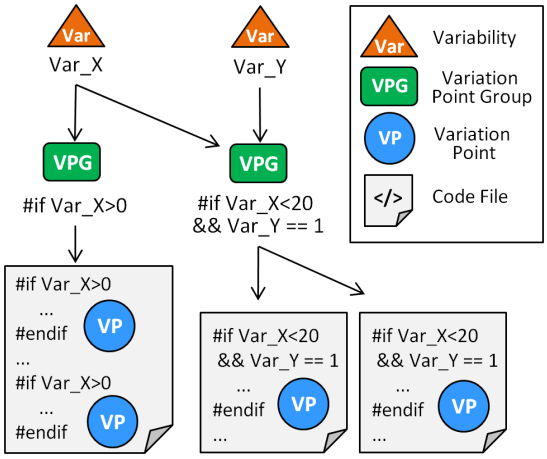


Figure 1. Variability elements and interdependencies.

4.2 GQM Model

Given the various VEs and their interdependencies, a plethora of different metrics can be easily measured from SPL realizations. While some of them might be helpful indicators for variability maintenance and improvement, others are not. In order to conduct the analysis systematically and concentrate on helpful metrics, we have followed the Goal-Question-Metric (GQM) approach [26] to get a clear understanding of the research goals and derive corresponding metrics from them. The GQM model supports goal-oriented software measurement with three levels, i.e., goal, question, and metric. The goal level defines tasks to be achieved, which are refined into questions on the question level to assess the achievement of these goals. Moreover, related metrics are defined on the metric level to answer each question in a quantitative way.

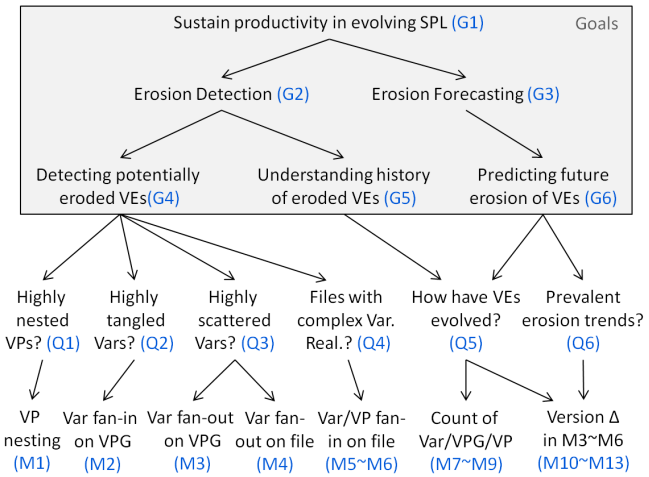


Figure 2. GQM model of our approach.

Given the practical challenges discussed in section 2, we have developed a GQM model for our analysis approach as illustrated in Fig. 2, and the goals, questions, metrics are introduced respectively as follows.

4.2.1 Goals

In the light of these practical challenges, the top-level goal is to sustain the productivity of SPL maintenance in the context of an evolving product line from the viewpoint of SPL developers (G1). As mentioned above, this can be achieved by detecting existing erosion (G2) and by forecasting future erosion (G3). G2 means to identify current erosion problems and understand their root causes. Then countermeasures such as erosion removal or tolerance can be taken against these problems. G2 can be decomposed into detecting potentially eroded VEs (G4) and understanding history of eroded VEs (G5). Besides, the goal of erosion forecasting (G3) is achieved by predicting future erosion of VEs (G6) based on evolution history.

4.2.2 Questions

In order to detect potentially eroded VEs, we investigate complex variabilities in term of nesting (Q1), tangling (Q2), and scattering (Q3). Besides, we also analyzed code files with complex variability realizations (Q4). These are the main issues of CC-based variability realizations in practice that cause understanding and maintenance problems. Therefore, they are the most suspicious in-depth causes of variability erosion, and should be identified and verified with domain knowledge.

Moreover, assuming that some overly complex VEs are verified as eroded ones, it is still unclear when they are introduced for what purpose. To learn how to cope with eroded VEs in the current SPL, it is necessary to trace back along the evolution history and investigate how the relevant VEs have evolved (Q5). It also helps to identify prevalent erosion trends in the future (Q6).

4.2.3 Metrics

To identify highly nested VPs (Q1), we calculate the nesting level of their represented #ifdef blocks (M1). The highly nested #ifdef blocks are difficult to understand due to their complex logic. Then we measure variability tangling (Q2) via the metric Var fan-in on VPG (M2). It is defined as the number of Vars that are used in the #ifdef expression of a given VPG (containing VPs with the same Vars). For instance, in Fig. 1 the Var fan-in on VPG of the VPG "#if Var_X<20 && Var_Y==1" is two, since it has two Vars. If the #ifdef expression of a VPG uses too many Vars, then its logic would be very complicated to understand.

Besides, variability nesting and tangling, we further measure variability scattering (Q3) in different VPGs as well as in different files. The metric of Var fan-out on VPG (M3) is defined as the number of VPGs that use a given variability in their #ifdef expressions. For instance, in Fig. 1 the Var fan-out on VPG of "Var_X" is two, since "Var_X" is used in two different #ifdef expressions. It indicates the complexity of interdependencies between macro constants and related #ifdef blocks in CC. The metrics of Var fan-out on file (M4) is defined as the number of code files containing a given Var. For instance, in Fig. 1 the Var fan-out on file of "Var_X" is three, since "Var_X" is used in three files. In case a Var is changed, the relevant Var fan-out on file metric indicates the maintenance effort, because the change needs to be propagated on all involved files.

To investigate the file complexity in terms of variation realization (Q4), we further calculate Var fan-in on file (M5) and VP fan-in on file (M6). Var fan-in on file is defined as the number of used Vars in a given file, while VP fan-in on file is defined as the number of VPs in a given file. For instance, the VP fan-in on file of the left file in Fig. 1 is two, since it includes two #ifdef blocks. As these metric values increase, a file includes more VEs and becomes difficult to understand for adapting changes in SPL maintenance.

To investigate the evolution of VEs (Q5), we calculate the number of different Vars (M7), VPGs (M8), and VPs (M9) in each version. Moreover, we also calculate the version differences of some fan-out and fan-in metrics (i.e., M3~M6) as M10~M13, which indicate the evolution of VE interdependencies and their code impact over time. All these delta metrics not only help to understand variability erosion in the past, but also help to predict prevalent erosion trends in the future. The delta value of Var fan-in on VPG (M2) is not calculated because it only depends on the VPG and does not change.

4.3 Implementation and Tool Support

In order to conduct the qualitative measurements over the large raw data set of 31 versions (avg. 3.6 MLOC each) in the Danfoss SPL, appropriate tool support is required. General purpose analysis tools exist to provide metrics and other facts about single systems, especially on the code level. Unfortunately, there is only little support available to analyze existing variability realizations, e.g. by means of CC, in the diverse assets and to compare it between different versions.

In order to provide sufficient tool support we have analyzed existing technology such as ifnames, grep, CDT, AST parsers as well as commercial analysis tools for their suitability in our context. Factors that we have considered are the accuracy of the results, performance and interoperability of the tools, and the adoption and integration efforts. We have also considered the fact that analysis and assessment demands differ considerably between different SPL settings and thus tool support must provide the respective customization, automation and extension mechanisms. The result of our inquiry was that there are tools that can be used for specific issues in the analysis, however, to our best knowledge there is no full-fledged tool support that allows analyzing and assessing all the issues discussed in the previous sections in a sufficient way.

To process N versions of a SPL infrastructure and support inter-version analyses, we have developed the VITAL (Variability ImprovemenT AnaLysis) tool set. VITAL is centered around a simple and extensible analysis data model, which represents relevant aspects of the whole product line (e. g., variability model, core assets, CPP directives, variation points) along respective measuring, analysis, and assessment tools. The integration of the various tool components is realized in MS Access 2010, as it allowed us to integrate and use various tools under a simple UI. Furthermore, the analysis capabilities can be adapted with the respective system owners efficiently by defining new queries in a graphical way and standard tools as MS Excel can be used for further analysis.

Basic facts about the CPP directives can be either extracted with SrcML [27] (which annotates source code with respective xml tags), SciTools Understand [24], or simple parsing scripts in Python or Perl. Obviously this is not a high performance solution, however it allows us to extract facts in a straight-forward way. If more sophisticated dependencies or metrics are required, we can get additional facts by means of the Fraunhofer SAVE tool [3] or SciTools Understand. Based on the gathered basic facts, VITAL derives aggregated facts, e.g. the structure of CPP directives, VPs and VPGs, and calculates respective metrics. In order to analyze the data, we have provided a series of SQL queries and Excel templates. Results can be visualized by means of Excel, the

Fraunhofer SAVE tool, or general-purpose graph visualization tools, such as Cytoscape, Treeviz and Gephi.

Regarding deriving VPGs, ideally VPs should be clustered into VPGs based on the semantics of their #ifdef expressions. However, comparing the semantic equivalence of two arbitrary expressions is theoretically undecidable. In our approach, VPGs are clustered syntactically by their #ifdef expressions. To increase the correctness of clustered VPGs, some standardization of #ifdef statements are conducted before clustering: 1) each #ifdef statement is rewritten as "#if defined(...)", and each #ifndef statement is rewritten as "#if !defined(...)"; 2) extra spaces in #ifdef expressions are removed.

In this case study, a Danfoss directive and the size of the data set prevented us from running the tools on-site on the code. To cope with this, we have first extracted all CPP statements from the different files and then reconstructed the CPP skeletons of the files at Fraunhofer. VITAL has then been applied on the skeleton data set. With this, we followed the "no purpose – no data" principle and have extracted only necessary base facts about the systems.

F4. Separating the basic fact extraction from the rest of the analysis helps to run analysis in restricted settings.

The overall analysis of the 187,674 files in 31 SPL versions with 110,857,457 LOC took after several performance improvements about 1.5 days and was performed by VITAL automatically. 6,929,265 CPP directives were analyzed, the respective information about VEs was gathered, and 3,713,575 metrics were calculated. The overall produced data set is 3.232 GB big – a considerable size, which raised some challenges during the analysis.

5. DANFOSS SPL EVOLUTION ANALYSIS

5.1 Versions of Variable Code

In this case study, we analyze the variability evolution of the Danfoss SPL from October 2008 until January 2013. Finally we manage to collect 31 versions within this period, although they are not equally distributed in time. The interval between two successive versions is generally one or two months, and 51 days on average. Each version uses CC as its variation mechanism. Since in this SPL each variability is defined with the naming convention "HAS_FEATURE", the variable code is actually VPs (#ifdef blocks) using HAS_FEATURE names. While most variable code is developed manually, there are a few automatically generated files that contain VPs for specific products. Since we are investigating variability erosion in the context of SPL maintenance, these generated files are excluded because they can be re-generated during change propagation and do not cause any maintenance problem.

We calculate the size of the variable code in each version as shown in Fig 3. In this large-scale industrial SPL, the size of variable code was about 42 KLOC at beginning, and has been increased up to about 120 KLOC in four years. As pure::variants was adopted in Danfoss from January 2012 for variability modeling, it facilitates variability realization maintenance, and *the code size increase has been significantly accelerated thereafter*.

F5. Improved variability management capabilities (e.g. tool support) can drive the increase of variability.

This fast growth might facilitate efficient product development in the early stages, but it also leads to SPL erosion and maintenance difficulties that affect productivity. To investigate these problems, we need to further analyze the evolution of various VEs.

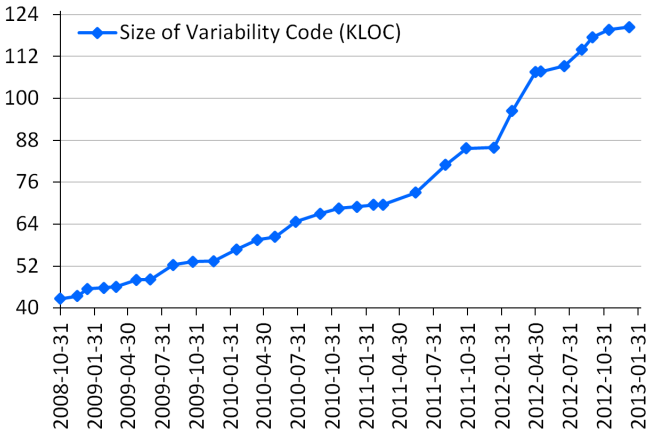


Figure 3. Evolution of variable code over 31 versions.

5.2 Evolution of Variability Elements

According to our definition of three types of VEs (i.e., Var, VPG, and VP), we have analyzed variable code and investigated evolution of these VEs along 31 versions. Fig. 4 shows that both the number of Vars (M7) and the number of VPGs (M8) have been increasing overall and approximately in a linear way. The grown was quite slow in the first several versions, because Danfoss conducted some code refactoring against increasing Vars and VPs at that time. Moreover, it seems that the number of Vars increases in proportion to the number of VPGs. This phenomenon reflects their interdependencies and will be discussed later in the measurement of Var fan-out on VPG (M3). According to our analysis the number of VPs (M9) also keeps increasing, which is not presented for the sake of brevity. Finally the last version in January 2013 contains 834 Vars, 1223 VPGs, and 13969 VPs. Although these measurements provide an overall knowledge of how VEs increase or decrease over time, it is still unclear how each individual VE evolves with respect to its interdependencies with other VEs or files. Therefore, we continue to investigate variability interdependencies and code impact with in-depth analysis.

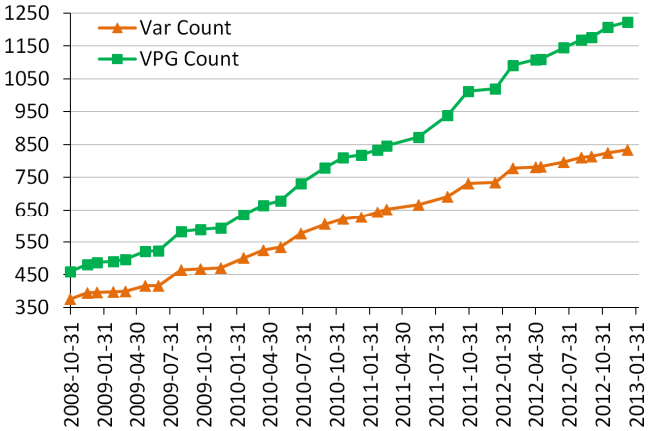


Figure 4. Evolution of Vars and VPGs.

5.3 Variability Nesting

As VPs are realized by #ifdef blocks in Danfoss SPL, they are sometimes presented in a nested layout. If a VP is deeply nested,

then its logic would be very complex, which impair program comprehension and make the code maintenance error-prone. Since this nesting issue does not only involve variability related #ifdef blocks (i.e., VPs), but also involves #ifdef blocks of other purposes, the nesting level of a VP is calculated in the context of all parent #ifdef blocks if existing. We have analyzed the nesting level (M1) of all 13969 VPs in the last version of Danfoss SPL, and Table 1 shows the grouping of these VPs based on their nesting level. It indicates that most VPs are not nested (nesting level equals one) or only nested in one degree (nesting level equals two). *The average nesting level of all variation points is 1.50*, which is not very complex in general.

Table 1. Grouping of VP nesting levels in the last version

Nesting Level (M1)	1	2	3	4	5
VP Count	7988	5082	833	64	2

However, there are a few VPs with a high nesting level (up to five), and maintaining such code is very expensive. After inspecting these highly nested VPs, our domain experts found that these VPs are mostly developed by an external team (probably lack of training) and are not well-reviewed afterwards. To mitigate this problem, one potential solution is to refactor the variability architecture so that the hierarchically structured VPs at a deep nesting level are encapsulated, using module replacement for example.

5.4 Variability Tangling

Besides nested VPs, another potential erosion problem is multiple tangled Vars used in the #ifdef expression of a VP. The tangling degree is measured by the metric Var fan-in on VPG (M2). Since VPs of the same VPG usually have the same Vars, we have conducted this measurement on all 1223 VPGs in the last version of Danfoss SPL, and Table 2 shows the grouping of these VPGs based on their tangling degree. It shows that most VPGs only contain one or two Vars in their #ifdef expressions. *The average tangling degree of all VPGs in the last version is 1.21*, which also indicates a low tangling level. However, there are a few VPGs with a high tangling degree (up to 11), which are very difficult to understand. This is probably due to inexperienced developers with insufficient training.

Table 2. Grouping of Var Fan-In on VPs in the last version

Var Fan-in on VPG (M2)	1	2	3	4	7	11
VPG Count	1010	185	22	4	1	1

F6. VPGs with more than five tangled Vars are difficult to understand and shall be considered as overly complex.

5.5 Variability Scattering

5.5.1 Var scattering in VPGs

Var scattering in VPGs depicts the situation when a Var is used in the #ifdef expressions of multiple VPGs. It is measured by the metric Var fan-out on VPG (M3), which indicates the change impact of a Var on VPGs in the scenario of SPL maintenance. We have conducted this measurement on each of the 834 Vars in the last version of Danfoss SPL, and Table 3 shows the grouping of these Vars based on their scattering degree. It shows that most Vars are scattered in one or two VPGs. *The average scattering degree is 1.68*, which also indicates a low scattering level.

However, there are a few Vars with a high scattering degree (up to 21), which means each of these Vars crosscuts the variability realizations in many different #ifdef blocks. Note that since different VPGs have different #ifdef expressions, the change propagation of a Var on different VPGs might be different. Therefore, the Vars with high scattering degree in VPGs would cause a tremendous maintenance effort if they are changed. Besides, our domain experts also confirm that,

F7. Vars that are scattered in more than two VPGs are not recommended and shall be considered as overly complex.

Table 3. Grouping of Var Fan-out on VPG in the last version

Var Fan-out on VPG (M3)	1	2	3~5	6~9	13~21
Var Count	548	188	70	23	5

5.5.2 Var scattering in files

Besides analyzing Var scattering in VPGs, we also investigate Var scattering in files. It is measured by the metric Var fan-out on File (M4), which is defined as the number of relevant code files which contain the scattered VPs of a given Var. This metric indicates the change impact of a Var on files. Since each file shall be quality-assured after change, this metric also indicate the QA effort during SPL maintenance. We have conducted this measurement on each of the 834 Vars in the last version of Danfoss SPL, and Table 4 shows the grouping of these Vars based on their scattering degree. Although 438 Vars (53%) are scattered in only one or two files, there are still 278 Vars (33%) scattered in three to ten files, and a few Vars have even higher scattering degree (up to 144). *The average scattering degree on file is 6.20*, which is also high compared to the scattering degree in VPGs. It implies that,

F8. Vars that are scattered in many VPGs are probably located in even more files (with replicated VPs).

Table 4. Grouping of Var Fan-out on Files in the last version

Var Fan-out on Files (M4)	1	2	3~10	11~46	54~144
Var Count	247	191	278	105	13

To sum up, we find out that,

F9. Var scattering in both a large number of VPGs and files leads to high maintenance cost, if the respective Vars have a high probability to evolve, e.g. by adding new variants.

One solution to this issue is to adopt certain architectural refactoring using a different variability realization mechanism, e.g., Aspect Orientation.

5.6 Complexity of Variability Files

Besides the issues of variability nesting, tangling, and scattering, another case of complex variability realizations is variable code files or modules with a large number of VEs. In the last version of the Danfoss SPL, there are 1322 variability-related code files that contain Vars and VPs. We investigate the number of Vars and VPs in these files in the following subsections.

5.6.1 Vars in variability files

The number of Vars in variability files is measured by the metric Var fan-in on File (M5). In this case study, we simply measure the number of HAS_FEATURE names that are used in VPs to enable or disable feature code. We have conducted this

measurement on all the 1322 variability files in the last version of Danfoss SPL, and Table 5 shows the grouping of these files based on the number of included Vars. It shows that most files only contain one to three Vars. *The average number of Vars per file is 3.91*, which also indicates a low complexity level.

Table 5. Grouping of Var Fan-in on File in the last version

Var Fan-in on File (M5)	1	2~3	4~10	11~49	54~118
File Count	679	316	229	89	9

However, there are a few files including up to 118 Vars, which would be very difficult to understand and error-prone during maintenance. Our domain experts have manually inspected these extremely complex files, and it turns out that most of them are due to non-optimal architectural design. E.g., the most complex file with 118 Vars is actually a configurator of a subsystem, where all relevant variable features are listed in the file without any hierarchical encapsulation.

In order to visualize the complex files in the SPL, we present the measurement result of Var fan-in on File in a rectangular treemap using the Treeviz tool [29], as shown in Fig. 5. Each block represents a variability file, and size of the block indicates size of the file. The file blocks are colored in gradient from blue, yellow, to red, which is based on their metric value of Var fan-in on File. The red file blocks represent files with a large number of Vars (Max. 118), while the blue file blocks represent files with a small number of Vars (Min. 1). The files in red color are the hotspots in variability realizations that are extremely complex and difficult to understand.

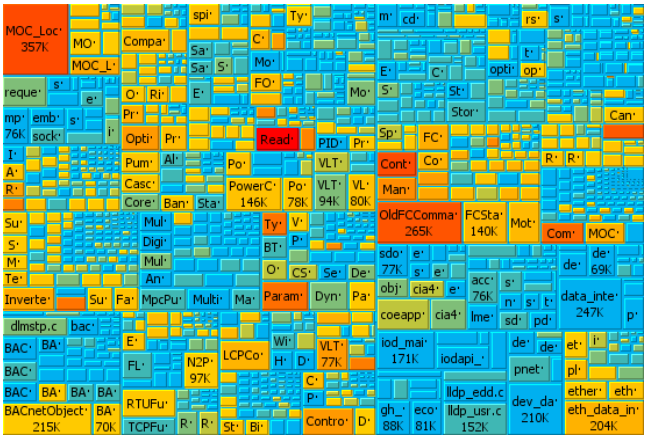


Figure 5. Visualization of Var fan-in on File.

5.6.2 VPs in variability files

The number of VPs in variability files is measured by the metric VP fan-in on File (M6). Since each VP is a #ifdef block in this Danfoss SPL, the metric indicates the cyclomatic complexity of a code file (If VPs are independent and not nested in the file, then the cyclomatic complexity is propositional to the number of VPs). We have conducted this measurement on all the 1322 variability files in the last version of the Danfoss SPL, and Table 6 shows the grouping of these files based on the number of included VPs. Although 676 files (51%) have only one to three VPs, there are 326 files with four to ten VPs and 273 files with 11 to 50 VPs. Additionally, there are 47 files with even 51 to 407 VPs in the extreme case. *On average each variability file has 10.57 VPs*, which is definitely a non-trivial problem. Actually Danfoss is aware of such problem, and they already conducted some improvement in the past to refactor some of the most complex variability files.

Table 6. Grouping of VP Fan-in on File in the last version

VP Fan-in on File (M6)	1	2~3	4~10	11~50	51~407
File Count	385	291	326	273	47

The measurement result of VP fan-in on File can also be visualized in a rectangular tree map, which is similar to Fig. 5 and is not shown in this paper for the sake of brevity.

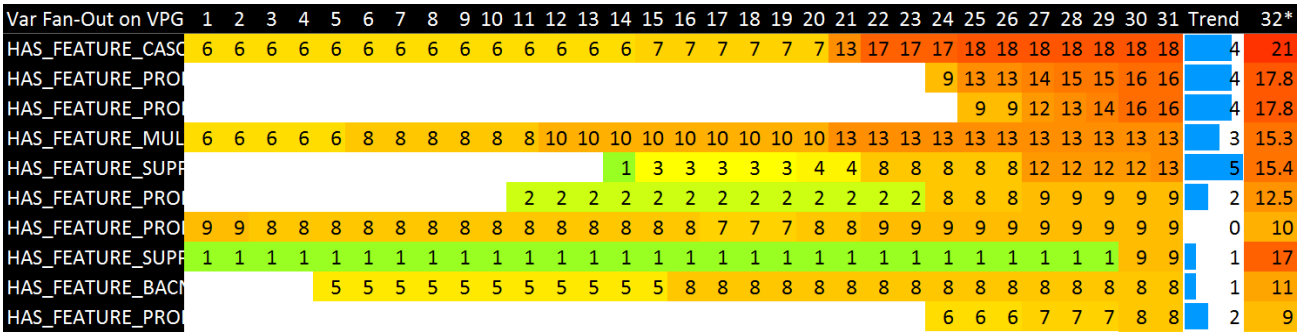
5.7 Variability Erosion in the Past and Future

So far we have investigated several scenarios of variability erosion in the last version, including variability nesting, tangling, scattering as well as code files with complex VEs. In these scenarios, variability realizations are overly complex and difficult to understand. However, these problems do not only exist in the last version, but are often introduced in an earlier version and typically become more serious during SPL evolution. Therefore, investigating the evolution history of VEs would help domain experts to understand why erosion occurs and how to cope with it.

To this end, we conduct the measurement of aforementioned fan-in and fan-out metrics not only on the last version, but also on all earlier versions. For instance, Fig. 6 shows a data table of ten out of 834 Vars with the measurement result of Var fan-out on VPGs (M3) over all 31 versions. Each column in the table indicates a version, while each row indicates a Var (with the naming convention "HAS_FEATURE") and its fan-out values in corresponding versions. The data cells are filled in gradient colors from green, yellow to red based on their fan-out values. If a Var is not used in a version at all, then the corresponding data cell is colored in white. Therefore, the colorful line strip in each row indicates the life-span of a Var during evolution, and the red data cells (with high values) indicate complex interdependencies where a Var is used in the many different VPGs.

Moreover, we further measure the delta value of fan-in and fan-out metrics (i.e., M3~M6) between two successive versions, namely the metrics M10~M13. In each delta metric a positive delta value (increment) implies that the complexity of a corresponding interdependency or code file increases, while a negative delta value (decrement) implies that the complexity decreases. Based on these delta measurements (the results are not presented for the sake of brevity), domain experts can easily find out when and why an erosion is introduced in the SPL.

The knowledge of variability evolution does not only help to understand erosion in the past, but also helps to identify prevalent erosion trends in the future. Our idea is that VEs and files with an increasing complexity are more likely to become more complex in the next version. Based on this assumption, the future erosion trend (i.e., the probability that a VE or file becomes more complex) can be calculated by the number of increments deducting the number of decrements in the delta metrics. For instance, as illustrated in Fig. 6, the first Var in the data table of Var fan-out on VPG (M3) was introduced in the first version, and



Var Fan-Out on VPG	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Trend	32*
HAS_FEATURE_CASC	6	6	6	6	6	6	6	6	6	6	6	6	6	6	7	7	7	7	7	7	13	17	17	17	18	18	18	18	18	18	18	4	21
HAS_FEATURE_PROI																								9	13	13	14	15	15	16	16	4	17.8
HAS_FEATURE_PROI																									9	9	12	13	14	16	16	4	17.8
HAS_FEATURE_MUL	6	6	6	6	6	8	8	8	8	8	8	10	10	10	10	10	10	10	10	10	13	13	13	13	13	13	13	13	13	13	13	3	15.3
HAS_FEATURE_SUPF														1	3	3	3	3	3	4	4	8	8	8	8	8	12	12	12	12	13	5	15.4
HAS_FEATURE_PROI											2	2	2	2	2	2	2	2	2	2	2	2	2	8	8	8	8	9	9	9	9	2	12.5
HAS_FEATURE_PROI	9	9	8	8	8	8	8	8	8	8	8	8	8	8	8	8	7	7	7	8	8	9	9	9	9	9	9	9	9	9	9	0	10
HAS_FEATURE_SUPF	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	9	9	1	17
HAS_FEATURE_BACI					5	5	5	5	5	5	5	5	5	5	5	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	1	11
HAS_FEATURE_PROI																								6	6	6	7	7	7	8	8	2	9

Figure 6. Var Fan-out on VPG in the past and future.

its value of has been increased from 6 to 18 with four increments and no decrements. Therefore, the erosion trend value is four, which is indicated by a bar in the data cell in blue color.

Besides predicting the erosion trend of a fan-in or fan-out metric, we further predict the most probable value of the metric in the next version. This predicted value equals the corresponding value in the last version plus the average of increments in the past versions. For instance, in Fig.6 the increments of the first Var in the data table are one, six, four, and one. Thus the average increment is three, and the predicted Var fan-out on VPG value of that Var in version #32 is 18+3=21. This value prediction does not consider the decrements because we are only predicting the metric value in case it increases in the future. As shown in Fig 7, the data table is ranked by the predicted value in descending order, so that domain experts can find out which VEs or files will be seriously eroded in the future.

5.8 Threats to Validity

The validity of the case study results can be threatened along five dimensions, i.e., internal, external, construct, conclusion, and reliability validity [30] [31]. Internal validity is threatened if the investigated systems contain defects that lead to incorrect analysis results. Luckily, in this case study all Vars follow a special naming convention, and all VPs that are implemented in CC. This enables us to accurately separate variability realizations with other code assets in the SPL. Besides, since we are only focusing on variability erosion in the context of software maintenance, automatically generated code files are excluded from this case study. An external threat is that the analyzed systems are not representative. We analyzed a real-world SPL in this case study, however from a limited application domain. Construct validity is threatened by selecting inappropriate metrics. We reduced this threat by using a goal-oriented approach which leads to the identification of just the required facts and metrics in the particular context. Conclusion validity is threatened if there are issues that affect the justification of case study conclusions, e.g., low statistical power. As a countermeasure to such threats, we analyzed an industrial SPL with large number of VEs. In order to investigate their evolution and especially their erosion history, we analyzed 31 version of this SPL over the recent four years.

6. CONCLUSION AND FUTURE WORK

As successful industrial companies need to sustain their productivity of SPL maintenance, the challenge of changing variability must be carefully tackled during the evolution of the SPL infrastructure. In this paper, we present industrial experience of SPL evolution and related open issues together with a case study. In order to investigate various symptoms of variability erosion, we derive a set of metrics in a goal-oriented way, and conduct corresponding measurements along the evolution history of a large-scale industrial SPL with tool support. Based on these measurements, potentially eroded VEs are detected automatically and inspected manually by domain experts. Moreover, the extracted variability evolution data is further used to predict prevalent erosion trend in the future.

The following lessons are learned: 1) in practice only a small portion of CPP directives are variability-related, and to recognize and extract these variability realizations automatically can be challenging and requires domain knowledge; 2) some VEs and related code files tend to become overly complex and erode during SPL evolution, which makes variability realizations difficult to understand and maintain; 3) there are several erosion symptoms in CC-based variable code, including complex interdependencies between VEs (e.g., variability nesting, tangling, and scattering) and complex variable code files; 4) complex VEs and related code files can be identified automatically, which should be inspected by domain experts for erosion detection; 5) the history of variability evolution helps to understand the cause of erosion and to predict prevalent erosion trends in the future.

Our future work includes: 1) further investigating variability erosion symptoms and their causes comprehensively with the help of domain experts; 2) deriving (semi-) automatic approaches to variability improvement by tolerating or correcting eroded VEs without affecting the semantics of variability realizations; 3) validating the effectiveness of variability improvement with controlled experiments.

7. ACKNOWLEDGMENTS

This work is within the MOTION project of “Innovationszentrum Applied System Modeling”, which is sponsored by the German state of Rhineland-Palatinate and Fraunhofer IESE. See http://www.applied-system-modeling.de/.

8. REFERENCES

[1] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," Dependable and Secure Computing, IEEE Transactions on, vol. 1, no. 1, pp. 11-33, Jan. 2004.

[2] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wkasowski, "A survey of variability modeling in industrial practice," in Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems, ser. VaMoS '13. New York, NY, USA: ACM, 2013.

[3] S. Duszynski, J. Knodel, and M. Lindvall, "SAVE: Software architecture visualization and evaluation," in Software Maintenance and Reengineering, 2009. CSMR '09. 13th European Conference on, Mar. 2009, pp. 323-324.

[4] C. Elsner, G. Botterweck, D. Lohmann, and W. Schröder-Preikschat, "Variability in time - product line variability and evolution revisited," in Proceedings of the Fourth International Workshop on Variability Modelling of Software-intensive Systems, ser. VaMoS '10, 2010.

[5] C. Gacek and M. Anastasopoules, "Implementing product line variabilities," SIGSOFT Softw. Eng. Notes, vol. 26, no. 3, pp. 109-117, May 2001.

[6] H. P. Jepsen, J. G. Dall, and D. Beuche, "Minimally invasive migration to software product lines," in Proceedings of the 11th International Software Product Line Conference, ser. SPLC '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 203-211.

[7] H. P. Jepsen and D. Beuche, "Running a software product line: standing still is going backwards," in Proceedings of the 13th International Software Product Line Conference, ser. SPLC '09. Pittsburgh, PA, USA: Carnegie Mellon University, 2009, pp. 101-110.

[8] C. Kästner, "Virtual separation of concerns: Toward preprocessors 2.0," Ph.D. dissertation, 2010.

[9] 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. Softw. Maint. Evol., vol. 18, no. 2, pp. 109-132, Mar. 2006.

[10] C.W. Krueger, "New Methods behind a New Generation of Software Product Line Successes", In K.C. Kang, V. Sugumaran, and S. Park, "Applied Software Product Line Engineering", Auerbach Publications, 2010, pp. 39-60.

[11] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze, "An analysis of the variability in forty preprocessor-based software product lines," in Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 105-114.

[12] S. Livengood, "Issues in software product line evolution: complex changes in variability models," in Proceeding of the 2nd international workshop on Product line approaches in software engineering, ser. PLEASE '11. New York, NY, USA: ACM, 2011, pp. 6-9.

[13] A. Lozano, "An overview of techniques for detecting software variability concepts in source code," in ER Workshops, ser. Lecture Notes in Computer Science, O. D. Troyer, C. B. Medeiros, R. Billen, P. Hallot, A. Simitsis, and H. V. Mingroot, Eds., vol. 6999. Springer, 2011, pp. 141-150.

[14] J. D. McGregor, "The evolution of product line assets," Tech. Rep., 2003.

[15] D. L. Parnas, "Software aging," in Proceedings of the 16th international conference on Software engineering, ser. ICSE '94, vol. 7, no. 4. Los Alamitos, CA, USA: IEEE Computer Society Press, Dec. 1994, pp. 279-287.

[16] T. Patzke, M. Becker, M. Steffens, K. Sierszecki, J. E. Savolainen, and T. Fogdal, "Identifying improvement potential in evolving product line infrastructures: 3 case studies," in Proceedings of the 16th International Software Product Line Conference - Volume 1, ser. SPLC '12. New York, NY, USA: ACM, 2012, pp. 239-248.

[17] T. Patzke, "Sustainable evolution of product line infrastructure code," Ph.D. dissertation, 2011.

[18] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," SIGSOFT Softw. Eng. Notes, vol. 17, no. 4, pp. 40-52, Oct. 1992.

[19] K. Pohl, G. Böckle, and F. J. Linden, Software Product Line Engineering: Foundations, Principles and Techniques. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

[20] L. Passos, K. Czarnecki, S. Apel, A. Wkasowski, C. Kästner, and J. Guo, "Feature-oriented software evolution," in Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems, ser. VaMoS '13. New York, NY, USA: ACM, 2013.

[21] Pure::variants. http://www.pure-systems.com/pv (Mar 2012)

[22] J. Savolainen and J. Kuusela, "Violatility analysis framework for product lines," SIGSOFT Softw. Eng. Notes, vol. 26, no. 3, pp. 133-141, May 2001.

[23] J. Savolainen and M. Mannion, "From product line requirements to product line architecture: optimizing industrial product lines for new competitive advantage," in Proceedings of the 13th International Software Product Line Conference, ser. SPLC '09. Pittsburgh, PA, USA: Carnegie Mellon University, 2009, p. 315.

[24] SciTools Understand V2.6. http://www.scitools.com/.

[25] Sincero, R. Tartler, D. Lohmann, and W. S. Preikschat, "Efficient extraction and analysis of preprocessor-based variability," SIGPLAN Not., vol. 46, no. 2, pp. 33-42, Oct. 2010.

[26] van Solingen, R., Basili, V.R., Caldiera, G., and Rombach, H.D, "Goal Question Metric (GQM) Approach,". In Marciniak, J.J. (Ed.): Encyclopedia of Software Engineering (2nd Ed.), John Wiley & Sons:578-583, 2002.

[27] SrcML.http://www.sdml.info/projects/srcml (March 2012)

[28] M. Svahnberg and J. Bosch, "Evolution in software product lines: Two cases," Journal of Software Maintenance, vol. 11, no. 6, pp. 391-422, Nov. 1999.

[29] Treeviz. http://www.randelshofer.ch/treeviz/. (May 2012)

[30] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, Experimentation in software engineering: an introduction. Norwell, MA, USA: Kluwer Academic Publishers, 2000.

[31] R. K. Yin, Case Study Research: Design and Methods, 4th ed. Los Angeles, CA: Sage Publications, Inc, Oct. 2008.

[32] B. Zhang and M. Becker, "Code-based variability model extraction for software product line improvement," in Proceedings of the 16th International Software Product Line Conference - Volume 2, ser. SPLC '12. New York, NY, USA: ACM, 2012, pp. 91-98.

[33] B. Zhang and M. Becker, "Mining complex feature correlations from software product line configurations," in Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems, ser. VaMoS '13. New York, NY, USA: ACM, 2013.