Extending the Reflexion Method for Consolidating Software Variants into Product Lines

Pierre Frenzel, Rainer Koschke,
University of Bremen, Germany
{saint,koschke}@informatik.uni-bremen.de

Andreas P.J. Breu, Karsten Angstmann
Robert Bosch GmbH, Germany
{Andreas.Breu,Karsten.Angstmann}@de.bosch.com

Abstract

Software variants emerge from ad-hoc copying in-the-large with adaptations to a specific context. As the number of variants increases, maintaining such software variants becomes more and more difficult and expensive. In contrast to such ad-hoc reuse, software product lines offer organized ways of reuse, taking advantage of similarities of different products. To re-gain control, software variants may be consolidated as organized software product lines.

In this paper, we describe a method and supporting tools to compare software variants at the architectural level extending the reflexion method to software variants. Murphy's reflexion method allows one to reconstruct the module view, a static architectural view describing the static components, their interfaces and dependencies and their grouping as layers and subsystems. The method consists of the specification of the module view and the mapping of implementation components onto the module view. An automatic analysis determines differences between the module view and its implementation.

We extend the reflexion method from single systems to software variants. Because software variants share a very large amount of code, we use clone detection techniques to identify corresponding implementation components between variants. The correspondence is then used to transfer as much of the mapping for the analyzed variants to the next variant to be analyzed.

1 Introduction

A *software product line (SPL)* is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [8]. There are different approaches to SPLs to organize reuse for families of systems. Various implementation techniques, such as conditional compilation, parameterization, generics, and code generation, have been proposed to implement variability in SPLs in a more maintainable way.

In reality, SPLs often emerge from experiences with successively addressed markets with similar, yet not identical needs. It is difficult to foresee these needs and, hence, to design an SPL upfront. To address these different needs, software developers often create new products by cloning-at-the-large. The complete source code is copied and then adjusted to the different needs, leading to software variants. The resulting variants are maintained separately. As the number of variants increases, the effort for maintaining them increases, too. Changes in the shared code must be repeated in all variants. This clone-and-run approach represents ad-hoc reuse with insufficient configuration management. In order to take advantage of more advanced SPL implementation techniques, we need to compare these variants to determine their commonalities and variabilities.

Contributions. This paper introduces a new method to reconstruct the static architecture of variants. We adapt the reflexion method to this context. Taking advantage of the commonalities of variants, we apply the reflexion method incrementally to a series of variants, where we use clone detection and function similarity to carry over the existing mapping from code entities to architecture entities for the variants already analyzed. The benefit is a simpler and faster reconstruction of the variants' architectures, which helps in consolidating variants into more organized SPLs. We describe a case study in which the method was evalu-

ated in an industrial context.

Overview. The remainder of this paper is organized as follows. Section 2 describes related research. Section 3 introduces the new method. In Section 4, the industrial case study is described to evaluate the new method. Section 5 concludes.

2 Related Research

This section describes related research in the areas of architecture recovery in the context of software product lines and the reflexion method, the two areas that are combined in this paper. Architecture reconstruction in general is described in two recent surveys [15, 22]. Another key technique we are using is clone detection. Because we only apply clone detection, it is not the focus of the paper, and hence clone detection research is not discussed here. Instead we refer the interested reader to a recent comprehensive survey on clone detection research [16].

Architecture Reconstruction for Product Lines: Two major research projects are driving software product line research. One is the US-American *Product Line Practice Initiative* at the *Software Engineering Institute* (SEI) at the Carnegie Mellon University and the other one the European *Café* project.

The SEI has compiled practice areas for software product line engineering [8]. One of these practice areas is *reengineering of legacy systems* using the methods MAP [24] and OAR [5]. Both methods describe the process for the selection of existing components for integration in the product line. Yet, they do no provide technical details on program analysis and tool support.

The European *Café* project (and its predecessor *Esaps*) is the largest European project on SPLs [25]. In *Café* reengineering-based approaches are further developed and used, too [19]. These approaches are used to validate existing architectures against the product line architecture. They do not address how to recover the product line architecture from existing systems.

The *Fraunhofer institute for experimental software engineering (IESE)* developed the product line framework PuLSE^{TM} [3]. PuLSE offers a framework for a product development cycle including the development of the necessary reuse infrastructure. An integral part of PuLSE is RE-PLACE [4], a methodology to integrate existing systems into an SPL. This approach uses wrapping to mitigate interface discrepancies between the SPL and the legacy components without touching the internals of the legacy components. RE-PLACE is a pure process description and does not delve into concrete program analysis techniques to support the process.

Kolb et al. described a case study where the framework of PuLSE is used to systematically refactor an existing software component for reuse in an SPL [14, 13]. They propose various metrics to measure quality before and after the refactoring. The refactorings simplify the use of preprocessor statements implementing variability and remove redundancy detected by a textual clone detector. Their focus is on refactoring; they do not describe how to reconstruct the architecture for variants.

Riva proposed a method for architecture reconstruction and applied it to products in an SPL to recover various architectural views [23]. These views helped the analysts to create an SPL architecture. Afterward, using relational algebra, conformance of product architecture implementations is checked continuously. The SPL was recovered from modules that were already organized as a product line, whereas our goal is to consolidate variants, which represent unorganized, highly redundant product lines.

Faust and Verhoef [10] observed a frequent development pattern in the information systems world they named *software mitosis*: In global organizations, where software is in use at many sites, local changes are often made, which eventually leads to software variants. We have observed the same pattern in the embedded world, where variants emerge from the variability of the underlying hardware. Faust and Verhoef propose the grow-and-prune method, in which software variants are a conscious and strategic means to explore new development paths. Upon success and using various metrics as indicators for the degree of mitosis, variants are eventually merged again. Clone detection is a key technique in this approach to help prune redundancy in variants. They describe cost and benefits for a large information system following this approach. They do not outline the technical details on how to reconstruct the architecture of the variants.

Reflexion Method: The reflexion method developed by Gail Murphy [21, 20] helps to reconstruct a static architectural view, known as *module view* [12] or *development view* [18]. The module view describes the static decomposition of the system in terms of static modules, their interfaces and dependencies and their grouping as layers and subsystems. The module view that was initially designed—like any other architectural view—does often not reflect the real implementation due to changes made in the source without updating the documented module view. Using the reflexion method, we can reconstruct the mapping from source code entities onto the module view. The basic idea of the reflexion model is to create a hypothesized module view from existing documentation or interviews with architects. Source entities (global variables, routines,

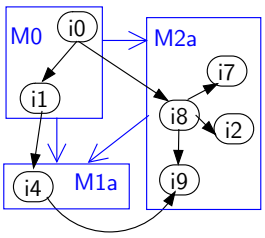

Figure 1. Reflexion example, the hypothesized module view consists of $M0$, $M1a$, and $M2a$ and the dependencies in between; all other elements are concrete; the mapping of concrete entities onto hypothesized modules is expressed through nesting.

types, classes, interfaces, packages, files, subdirectories, etc.) are extracted from a system along with their respective dependencies forming the concrete module view. These elements are mapped to the hypothesized view. A tool then computes resemblances and differences between the two views, according to the following rules (let H_1 and H_2 be two entities in the hypothesized module view and c_1 and c_2 be two entities in the concrete module view; let c_1 be mapped onto H_1 and c_2 onto H_2, respectively):

- if there is a specified dependency from H_1 to H_2 and a concrete dependency from e_1 to e_2, then there is a convergence between H_1 and H_2; for instance, the dependency between $M0$ and $M1a$ and between $i1$ and $i4$ in Figure 1 lead to a convergence
- if there is a specified dependency from H_1 to H_2 but no concrete dependency from any e_1 to any e_2, then there is an absence between H_1 and H_2; for instance, the dependency between $M2a$ and $M1a$ in Figure 1 cannot be confirmed by any concrete dependency and hence leads to an absence
- if there is a concrete dependency from e_1 to e_2 but no specified dependency from H_1 to H_2, then there is a divergence between H_1 and H_2; for instance, the dependency between $i4$ and $i9$ in Figure 1 is not covered by any hypothesized dependency and hence leads to a divergence

The resulting convergences, divergences, and absences yield the so called *reflexion view* or *reflexion model*. Iteratively, the hypothesized and concrete views and/or the mapping are refined based on the findings.

The technique was successfully used in several case studies for single systems [20]. Koschke and Simon extended the original reflexion model so that hypothesized modules can be hierarchical. They applied the extended method to two different compilers [17].

The most challenging parts of the reflexion method is to determine the hypothesized architecture and the mapping of concrete source entities onto the hypothesized entities. The original reflexion method does not provide any support for that. Sometimes naming conventions may be leveraged for the mapping. Yet, they often do not exist or are used inconsistently. Christl, Koschke, and Storey [6, 7] have integrated clustering techniques to propose mappings of entities whose neighbors have already been mapped, which is applicable if a partial mapping exists.

3 Reflexion Model for Software Product Lines

In SPLs we have three levels of models: the implementation, the architecture for single products, and the SPL architecture. In this section, we describe how to extend the reflexion method to compare product implementations to the product architecture and to compare product architectures to the SPL architecture.

So far the reflexion method is designed for single-system analysis. In a product line context, we need to extend the expressiveness in the hypothesized module view to accommodate the need to specify commonalities and variabilities in different product architectures and then consequently the rules to determine the reflexion view.

Because our goal is to consolidate software variants into product lines, we can assume that implementations and architectures of the variants are sufficiently similar. Thus, we want to leverage these similarities both at the implementation and architectural level, when we analyze different variants incrementally using the reflexion method. That is, if we compare the product implementation to a product architecture, we want to take advantage of the fact that we have a similar implementation, architecture, and mapping in between for previously analyzed variants. Consequently, we can ease the mapping process here by reusing the existing mapping for one variant when we analyze the next variant to a large extent.

We outline first the overall method and then we describe technical details in this section.

1. Select one first product and reconstruct the module view using the reflexion method; the decision on which product to analyze first can be made on the basis of different aspects such as familiarity, degree of authenticity of available documentation, system size, degree of supported common features, maximal similarity to all other products in terms of the implementation (e.g., amount of shared code), and version history.

2. Continue with variant i until all variants are analyzed:
   (a) find implementation commonalities and variabilities between i and all other variants $j < i$ already analyzed
   (b) carry over mapping for variants $j < i$ to variant i using the results of step 2a
   (c) validate these mapping suggestions
   (d) complete mapping and module view for variant i
   (e) map module view for variant i onto SPL module view carrying over the mappings for module view of variants $j < i$
   (f) validate these mapping suggestions
   (g) complete mapping and accommodate variabilities of variant i in SPL module view

Steps 2a-2d address the implementation level whereas steps 2e-2g unify the product architectures in the SPL architecture.

We will now describe the individual steps in more detail.

3.1 Mapping Process at Implementation Level

To reconstruct the module view for the first version, we can use the original reflexion method. In all other iterations, we want to reuse the existing mappings as far as possible. For this to work, we need to know the correspondences between the implementations of the variant to be analyzed and all variants analyzed so far.

We assume that two variants share a large amount of code. That is, for many functions in one variant, there are corresponding functions in other variants. These functions may be identical or slightly changed. In the latter case, they represent variability at the implementation level.

Furthermore, we assume that the architecture of different variants are similar. That is, corresponding functions of variants are mapped onto the same or similar hypothesized modules.

If both assumptions hold, we can first identify all corresponding functions in the already analyzed variants for a given function f in the variant to be analyzed. Then we can determine their mapping targets in the module view to make suggestions where to map f. For each validated suggestion, we can add the target module and its dependencies to the module view of this variant.

Correspondence between functions in two variants can be determined using the Levenshtein distance for the tokens of a function pair. The Levenshtein distance is the minimum number of operations needed to transform one string into the other, where an operation is an insertion, deletion, or substitution of a single character. In other words, it is the number of edits to transform one sequence into the other. Other authors used Levenshtein distance for clone analysis [2] and origin analysis [26].

The Levenshtein distance is an absolute value. To define a common threshold for acceptable similarity, we prefer a relative value between 0 and 1. To obtain such a similarity, we can divide the Levenshtein distance by the worst case number of edits. The worst case are two sequences totally different. In order to transform the larger one to the smaller one, we need to remove as many tokens as the larger one is longer and then to substitute every other remaining token. Consequently, the length of the larger sequence is the number of edits in such a case. The similarity metric is, hence, as follows (let *len* denote the length of a sequence and let *LD* denote the Levenshtein distance):

$$\mathrm{sim}(s1, s2) = 1 - \frac{\mathrm{LD}(s1, s2)}{\max(\mathrm{len}(s1), \mathrm{len}(s2))}$$

Because of the quadratic time complexity for computing the Levenshtein distance, we use the linear-time token-based clone detection by Baker [1] to identify pairs of functions (f_1, f_2) where f_1 and f_2 stem from different variants and share a code clone. Only for these, we compute the similarity metric $\mathrm{sim}(f_1, f_2)$. If this value is 1, the functions are identical. Because we need to cope with variability at the function level, we do not insist on identical functions. Instead, we identify for a given function f of a variant i all functions f_n in variants $j < i$ for which similarity is above a user-defined threshold: $\mathrm{sim}(f, f_n) \geq \theta$.

The corresponding functions are presented to the user ranked by *sim*. If there is only one corresponding f_n and $\mathrm{sim}(f, f_n) = 1.0$ or, respectively, if there are many identical f_n but all are in different variants, the user has the option to accept automatically.

If not only functions are to be mapped but also global variables and types, we can match corresponding entities of this type by name and their types or data structure declarations, respectively. Moreover, common corresponding accessing functions are another indicator.

We can raise the level of abstraction from individual functions to groupings such as files and directories in C or classes and packages in C++ and Java. For these, we can determine their similarity by accumulating *sim* for all pairs of functions (f_1, f_2) where f_1 is defined in a grouping of one variant and f_2 is defined in a grouping

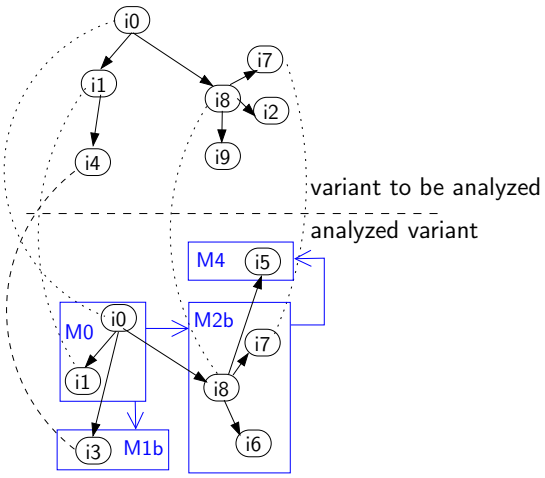

Figure 2. Comparing two variants.

of a different variant and $sim(f_1, f_2)$ is maximal for all such pairs.

Once the analyst has validated the information and selected pairs of corresponding functions or more coarse-grained groupings (f_1, f_2), we can carry over the mapping. If f_2 was mapped onto a hypothesized module M, we suggest to add M to the module view of the currently analyzed variant and to map f_1 onto M as well.

Figure 2 is used to illustrate this step. The dotted lines indicate the corresponding implementation entities between two variants. Because there is only one candidate for each correspondence, the circumstances suggest to create a module $M0$ to the module view of the variant to be analyzed and to map $i0$ and $i1$ onto it. The entities $i7$ and $i8$ may be mapped onto a new corresponding module $M2b$; yet, no entity corresponds to $i6$ mapped to $M2b$ in the other variant. This circumstance suggests to create a variant of $M2b$ for the variant to be analyzed. Entity $i4$ is similar but not identical to $i3$ expressed by the dashed line, which suggests to create a variant of $M1b$ onto which $i4$ is mapped. This mapping conforms to the dependencies in the module view of the analyzed variant.

We cannot make any immediate suggestions for entities $i2$ and $i9$ because they do not have corresponding entities. Yet, integrating our earlier ideas in the combination with clustering [6, 7], they could take on the role of $i6$ or $i5$. At last, the analyst completes the mapping and module view. The resulting module view is shown in Figure 4 as variant V_1.

3.2 Extensions to Hypothesized Architecture

To be able to express commonalities and variabilities in the SPL module view, we follow the approach by

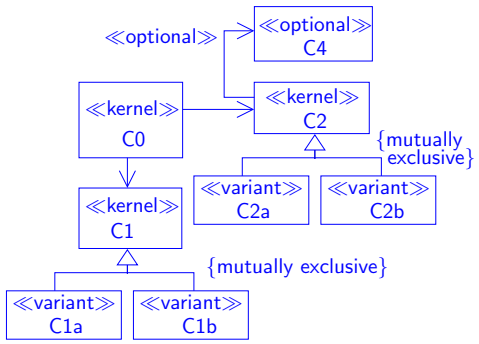

Figure 3. Example SPL architecture.

Gomaa [11]. Gomaa proposed various way of modeling SPL-relevant aspects in UML. Among these, static modeling is most relevant to our context.

The known elements of class diagrams—such as classes, interfaces, aggregation, inheritance, and other associations—can be used to model the module view for single systems and the SPL. In order to express variability and commonality in the SPL module view, he proposes to use the following UML stereotypes (see also the example in Figure 3):

- ≪kernel≫ marks all modules required by all products
- ≪optional≫ marks all modules that are present in only some products
- ≪variant≫ marks all modules that are similar, but not identical to other modules in other products

Variants can be specified for both ≪kernel≫ and ≪optional≫ modules. Hence, Gooma's categorization is actually orthogonal. To specify variability of a module, we can add a module to the module view tagged either ≪kernel≫ or ≪optional≫ and then derive variants from them using UML inheritance. To further express the variability, Gooma tags the inheritance by the constraint {`mutually exclusive`}.

Gooma uses the UML inheritance relationship to express variability. It would have been better to introduce a new type of relationship instead, for instance, using a new stereotype, because we must now distinguish between inheriting variants and inheriting optional and kernel modules. Although not explicitly stated by Gooma, we can assume the following additional constraints.

Inheritance for variants: For integrity reasons, it is not possible to derive a variant optional module from a kernel module and vice versa. Hence, if a module is optional, all derived variants are optional, too. The same holds for kernel modules. Variants can only exist

if they are (transitively) derived from an optional or kernel module. Every variant must have at least one sibling in the inheritance tree.

Inheritance for optional/kernel modules: A kernel module may inherit from a kernel module. The semantics is different for the type of inheritance used to model variation though. In such a case, both modules must be present and they must be derived from each other. Similarly, an optional module may inherit from an optional module in which case both modules may or may not exist in a product. If the derived module exists and the inheritance is not tagged optional, the module from which this module derives must exist as well. From a modeling point of view, it does not make sense to allow to derive an optional module from a kernel module and vice versa.

Gomaa proposed these stereotypes only for modules originally. It is obvious that every dependency to an optional module must also be optional, but there may also be dependencies to kernel modules that are optional. For this reason, we will use the stereotypes ≪optional≫ and ≪kernel≫ for dependencies, too. There is no need for ≪variant≫ dependency as this can be modeled using variant association classes in UML.

To improve readability of the SPL module view, we can use ≪kernel≫ as the default if a module or dependency does not have an explicit stereotype.

3.3 Reflexion View for Product Lines

At the SPL level, we want to compare product architectures to the SPL architecture by computing the reflexion view. In the previous section, we have extended the expressiveness for the hypothesized module view to model particularities of SPLs. Because of this extension, we need to extend the rules for computing absences, convergences, and divergences accordingly.

The above stereotypes translate directly to constraints on the mapping that relates product architecture (PA) and software product line architecture (SPLA). If a module M is tagged ≪kernel≫ in the SPLA, there must be a module in the PA that is mapped onto M or on one of its variants. If a module M is tagged ≪optional≫ in the SPLA, there may be a module in the PA that is mapped onto M or on one of its variants. If there is no such module in the PA, M and its variants are ignored for the reflexion analysis. If there is such a module in the PA, all (inherited and own) dependencies to and from the variant of M onto which the PA module is mapped must be present except for those that are explicitly tagged optional.

Following these constraints, we can derive a module view for the SPLA subject to checking, which consists of all (variant) kernel modules and those (variant) optional modules that are target of the mapping. We add all their (inherited) incoming and outgoing dependencies whose both ends exist in this derived module view.

This derived module view is used for the reflexion analysis and plays the role of the hypothesized view. Optional dependencies in the derived module view require to supplement the rule for absences:

> if there is a specified optional dependency from H_1 to H_2 but no concrete dependency from any e_1 to any e_2, then no absence between H_1 and H_2 is reported

Figure 4 is used to illustrate this step. For variant V_1, the derived module view for the hypothesized view consists of $C0$, $C1a$, and $C2a$ and the dependencies $C0 \rightarrow C1a$ and $C0 \rightarrow C2a$. Comparing this view to the module views of variant V_1, there are only convergences. For variant V_2, the relevant hypothesized view consists of $C0$, $C1b$, $C2b$, and $C4$ and the dependencies $C0 \rightarrow C1b$, $C0 \rightarrow C2b$, and $C2b \rightarrow C4$. In this case, too, there are only convergences.

4 Case Study

In this section, we report on an industrial case study in which we evaluated our method. We will first describe the family of variants, then specify measurements relevant to assess the method quantitatively, and finally present the results and potential threats to validity.

4.1 System Studied

We analyzed industrial software of a family of embedded systems from the automotive domain written in C. Company-sensitive details are omitted as much as possible. The software is embedded in a control device and deployed by different car manufacturers. Each car manufacturer uses it for different car models. Consequently, the devices differ at two levels: firstly, between car manufacturers and secondly, between different types of cars of a manufacturer.

The variation of the control devices comes from using different hardware and offering different features ranging from low-end to high-end cars. The diversity of the hardware and feature set is mirrored in the software. An older generation of this SPL was implemented by a mixture of preprocessor directives and branches in the version control system. There are hundreds of such variants. Because of the difficulties to maintain the old SPL, the latest SPL is implemented using better variability mechanisms. The old SPL is in

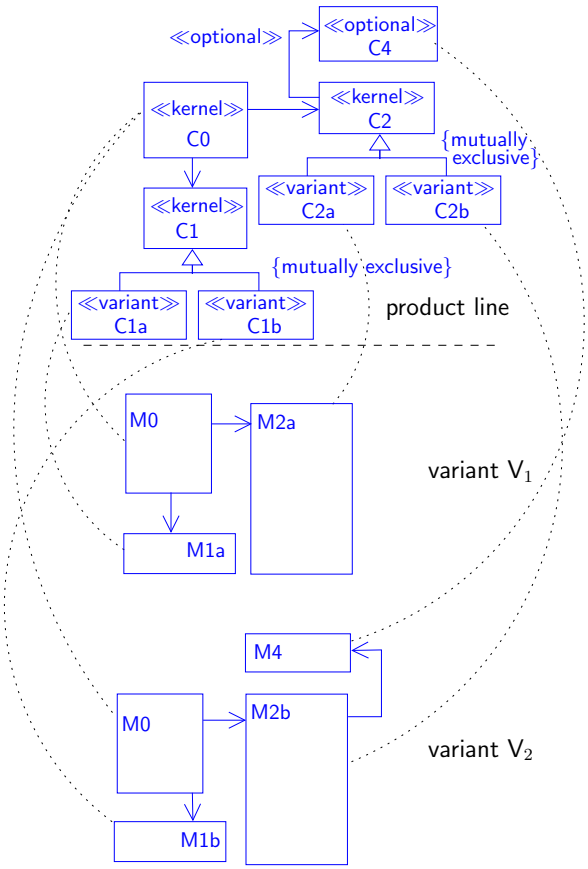

Figure 4. Comparing PAs and SPLA.

the service phase now but still in use. We use the old SPL as a test bed to evaluate our ideas.

To evaluate our proposed method, we selected two variants of this SPL. The goal of this study was to investigate feasibility. That is why we selected only two variants. To investigate the worst case, we selected one variant for a low-end car and one for a high-end car of two different car manufacturers. Throughout this section, we will refer to the low-end variant as L and to the high-end variant as H.

Despite the differences of the supported features, the architectures and even the physical code structure in terms of files and directories are similar. The directory structure has two directories in which all modules occur that implement customer-specific features. We call these the *feature modules* in the following. All other directories contain infrastructure code that is intended to be shared among variants. We call these modules the *kernel modules*. The kernel modules are intended to be prevailingly equal across variants although we may expect some differences across car manufacturers.

Table 1 lists some characteristics of these two variants. *SLOC* is the number of physical sources lines

variant	SLOC	files.	func	k-func	f-func
H	106,383	470	1,538	584	954
L	64,091	448	1,017	520	497

Table 1. System characteristics

measured by `sloccount`. Column *files* is the number of C files including header files and so-called project files containing variable declarations. Column *func.* is the number of functions defined in the variants. The number of functions of kernel modules is listed in column *k-func* and those of feature modules in *f-func*.

The architectures were reconstructed by the authors of this paper; two of them are reverse engineering experts and the other two domain experts. In some rare cases, the domain experts disagreed in the beginning and disputed to come to a common conclusion.

We analyzed first variant L and then H because H offers a superset of the features of L and has more code.

4.2 Measurements

The applicability of our suggested method depends upon the similarity between variants in terms of both implementation and architecture. We measure practicability of the method in our industrial case study by the degree of transferability of the mapping between two variants and the similarity of their module view.

To assess transferability of the mapping, we measure the degree of correspondences by considering the obviousness of the mapping. Each function f is associated with a set of corresponding identical functions $F_I = \{f' | \text{sim}(f, f') = 1\}$ and corresponding similar functions $F_S = \{f' | \theta \leq \text{sim}(f, f') < 1\}$ of the other variant. The following classes are relevant for the transferability of the mapping:

- I1 $|F_I| = 1$: there is one identical candidate to determine the mapping for f
- I2 $|F_I| > 1$: there are identical candidates to determine the mapping for f; this rare case may happen if the variant has function clones
- S1 $|F_I| = 0 \wedge |F_S| = 1$: there is one varied candidate to determine the mapping for f; the difference of f and this candidate is an indicator for a potential variability
- S2 $|F_I| = 0 \wedge |F_S| > 1$: there are several varied candidates to determine the mapping for f, hence, it is not obvious where to map f; an analyst has to inspect several candidates; presenting the candidates in descending order of similarity may help; if a candidate can be found, a variability point is detected

U otherwise: a corresponding function cannot be found

We can gather the frequency distribution of the above classes for all functions of the variant to be analyzed. Because this distribution depends upon the threshold θ, we will experiment with different values for θ in the range of $0.6 \ldots 0.95$.

To assess the similarity of the module views among variants, we can count the shared and different modules and dependencies. Yet, the comparison is not that simple if we have hierarchical modules. It is more likely that the module views are alike at top level but are different at more detailed levels. In such cases, we could report the sharing of modules and dependencies at varying nesting levels as follows, where $N_{n,i}$ is the set of modules and $E_{n,i}$ the set of dependencies for variant $i \in \{1, 2\}$ at nesting level n:

$$\text{Ncomm}_n = \frac{|N_{n,1} \cap N_{n,2}|}{|N_{n,1} \cup N_{n,2}|} \quad \text{Ecomm}_n = \frac{|E_{n,1} \cap E_{n,2}|}{|E_{n,1} \cup E_{n,2}|}$$

Modules are shared between variants if they are either both identical or variants of each other.

Finally, in order to compare the resulting reflexion views for the variants, we count the number of shared absences, convergences, and divergences.

4.3 Results

Table 2 shows the frequency distribution comparing the two variants depending on varying values of threshold θ. There is a rather large number of instances in U but we need to keep in mind that we started the analysis with L and then tried to match the functions in H with those in L. Because H has 1,538 functions whereas L has only 1,017, we can expect that about 500 new functions in H will not have a corresponding function in L. As a consequence, we can relate the 207 uniquely identically matching functions in $I1$ to the remaining ca. 1,000 functions for H for which a correspondence may exist. That is, about 20 % of the mapping may be transferred at high likelihood.

Surprisingly many functions have multiple identical matches as shown in column $|I2|$, which means that there are completely cloned functions in L.

As expected, the cardinalities of $S1$ and $S2$ decrease with increasing similarity threshold. The fact that $|S1| > |S2|$ suggests that the similarity metric may be an indicator for correspondence. To validate this indication, we validated the functions in $S1$ to check whether the functions are truly variants. Column $|S1'|$ lists the number of those functions in $S1$ which have only minor differences in terms of identifiers, literals, or

θ	$\|I1\|$	$\|I2\|$	$\|S1\|$	$\|S1'\|$	$\|S1''\|$	$\|S2\|$	$\|U\|$
0.6	207	27	175	21	121	85	1044
0.7	207	27	158	21	98	47	1099
0.8	207	27	106	20	66	26	1172
0.9	207	27	61	15	28	10	1233
0.95	207	27	37	9	13	0	1267

Table 2. Results for whole variants

θ	$\|I1\|$	$\|I2\|$	$\|S1\|$	$\|S2\|$	$\|U\|$
0.6	168	3	121	41	222
0.7	168	3	98	31	255
0.8	168	3	69	21	294
0.9	168	3	41	10	333
0.95	168	3	27	0	357

Table 3. Results for whole kernel modules

syntax. These functions could also be counted in $I1$. Furthermore, column $|S1''|$ lists the number of those functions in $S1$ that are true variants. The variation comes from additional code. Some of the code is nested in conditional preprocessor statements. We note that most of the candidates in $S1$ could be used to suggest a mapping.

Furthermore, it is interesting to see that $|S1'| < |I1|$ holds for all thresholds we considered, which means that commonality is larger than variability.

To validate the metric from a different angle, we analyzed kernel and feature modules separately. That is, we compared all kernel functions of H to all kernel functions of L and all feature functions of H to all feature functions of L. We expect that there is a higher degree of correspondence between kernel modules than between feature modules.

Tables 3 and 4 show the results of the respective separate analysis. The numbers clearly confirm our hypothesis. A large percentage of the existing mapping for one variant can likely be transferred. Yet, about a third of the functions does not have any correspondence, which indicates a high degree of variance even in the kernel modules.

Another interesting observation can be made if we compare the results of the two separate anlyses to the analysis of the whole system. The cardinalities of $I1$ for the two separate analyses (168+11=179) do not add up to the cardinality of $I1$ for the whole analysis (207). Our search for the missing 32 functions showed that there are identical cloned functions between kernel and feature modules.

Beyond commonalities at implementation and architecture level between variants, commonalities in the resulting reflexion views of both variants are also interesting. If the reflexion view of one variant has a di-

θ	$\lvert I1\rvert$	$\lvert I2\rvert$	$\lvert S1\rvert$	$\lvert S2\rvert$	$\lvert U\rvert$
0.6	11	8	23	15	521
0.7	11	8	23	5	531
0.8	11	8	19	0	540
0.9	11	8	13	0	546
0.95	11	8	8	0	551

Table 4. Results for whole feature modules

	only in L	only in H	in L and H
#modules	1	5	30
#depend.	10	78	149

Table 5. Similarity at the architectural level

vergence or absence, it may be acceptable for the other variant to have the same divergence or absence, respectively. For this reason, we compared the reflexion views of variant L and H. We found no absences for both variants, which is because we reconstructed the module view largely bottom-up. But we found several divergences, where the implementation should be improved. Of these divergences, only one was found shared in both variants, whereas there are 11 divergences only in L and 35 only in H. These are largely between modules that are specific to the respective variants. We observe in this case that there is too little similarity with respect to absences and divergences in the reflexion view to give hints on the mapping. Yet, 107 convergences are shared, which may be further help for transferring the mapping as it suggests that the module views are similar. Moreover, the fact that there are only 11 convergences only in L but 100 only in H indicates that the module view of L is largely subsumed by the module view of H.

Table 5 describes the similarity of the hypothesized module views of the variants after the reconstruction in terms of the number of modules and dependencies present in either only L, only H or both. Both variants have five hierarchical modules. Yet, they have the same elements except for one module. Consequently, it is not necessary to make a distinction between nesting levels in our case study. That is, $Ncomm$=83 % and $Ecomm$=63 % showing a high degree of similarity between the architectures. In particular, the additional modules in H add many new dependencies not present in L.

4.4 Threats to Validity

There are threats to validity of our study that should be taken into account when interpreting our results. Firstly, we have analyzed only two variants of a family of variants. The family has over one hundred variants. Yet, at least we have selected two variants that are at the two extremes of the features set for this family of devices as a worst case scenario.

We analyzed a family in one particular domain, namely, embedded systems. In the information systems domain, implementations and architectures of variants might differ more.

Then, we analyzed code of only one particular organization. Other organizations might have different development characteristics leading to different results. In particular, we analyzed industrial code. It is an open question how open-source code compares to industrial code in general.

5 Conclusions

In this paper, we proposed a new method to compare variants in terms of their static architecture and implementation. The method uses an extension to the reflexion method. To incrementally transfer the mapping from one variant to the other, we use a similarity function based on Levenshtein distance to determine the similarity between two functions sharing cloned code. We evaluated the method in an industrial case study. The results demonstrate that a large percentage of the existing mapping for one variant can be transferred, suggesting that an incremental analysis of a series of variants is possible.

Our technique is applicable even if functions are renamed or moved between files. If we wanted to take into account similar names and locations in files and directories, we could integrate these aspects as additional indicators in our similarity metric. We will explore the effect of this integration in future research. Our technique is generic enough to be applied to other domains than embedded systems as long as the variants show enough similarity in terms of implementation and architecture. Whether other domains show the same degree of variant similarity is subject to our future research, in which we replicate our experiment for other domains and programming languages. As another avenue for future research, we plan to integrate our feature location technique [9] to compare systems also in terms of their behavior and not just static structure.

Our method could also be used for analyzing the evolution of a system. To do so, multiple versions of the same system could be viewed as variants. Here, the analysis is even simplified because we have a natural order for the incremental analysis following the time line.

References

[1] B. S. Baker. On Finding Duplication and Near-Duplication in Large Software Systems. In L. Wills, P. Newcomb, and E. Chikofsky, editors, *Second Working Conference on Reverse Engineering*, pages 86–95, Los Alamitos, California, July 1995. IEEE Computer Society Press.

[2] E. Balazinska, M.and Merlo, B. Dagenais, M.and Lague, and K. Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. In *Working Conference on Reverse Engineering*, pages 98–107. IEEE Computer Society Press, 2000.

[3] J. Bayer, O. Flege, P. Knauber, R. Laqua, D. Muthig, K. Schmid, T. Widen, and J.-M. DeBaud. Pulse: a methodology to develop software product lines. In *Proceedings of the 1999 symposium on Software reusability*, pages 122–131. ACM Press, 1999.

[4] J. Bayer, J.-F. Girard, M. Würthner, J.-M. DeBaud, and M. Apel. Transitioning legacy assets to a product line architecture. In *Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 446–463. Springer-Verlag, 1999.

[5] J. Bergey, L. O'Brien, and D. Smith. Using options analysis for reengineering (OAR) for mining components for a product line. In *Proceedings of Second Software Product Line Conference*, volume 2379, pages 316–327. Springer Lecture Notes in Computer Science, Aug. 2002.

[6] A. Christl, R. Koschke, and M.-A. Storey. Equipping the reflexion method with automated clustering. In *Working Conference on Reverse Engineering*, pages 89–98. IEEE Computer Society Press, Nov. 2005.

[7] A. Christl, R. Koschke, and M.-A. Storey. Automated clustering to support the reflexion method. *Journal Information and Software Technology*, 49:255–274, Mar. 2007.

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

[9] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Computer Society Transactions on Software Engineering*, 29:210–224, 2003.

[10] D. Faust and C. Verhoef. Software product line migration and deployment. *Journal of Software Practice and Experiences*, 33(10):933955, Aug. 2003.

[11] H. Gomaa. *Designing Software Product Lines with UML – From Use Cases to Pattern-Based Software Architectures*. Addison-Wesley, 2005.

[12] C. Hofmeister, R. Nord, and D. Soni. *Applied Software Architecture*. Object Technology Series. Addison Wesley, 2000.

[13] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi. A case study in refactoring a legacy component for reuse in a product line. In *International Conference on Software Maintenance*, pages 369–378. IEEE Computer Society Press, March–April 2005.

[14] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi. Refactoring a legacy component for reuse in a software product line: A case study. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(2):109–132, March–April 2006.

[15] R. Koschke. Rekonstruktion von Software-Architekturen: Blickwinkel, Sichten, Ansichten und Aussichten. *Informatik – Forschung und Entwicklung, Springer Verlag.*, 19(3), Apr. 2005.

[16] R. Koschke. Survey of research on software clones. In R. Koschke, E. Merlo, and A. Walenstein, editors, *Duplication, Redundancy, and Similarity in Software*, number 06301 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2007. <http://drops.dagstuhl.de/opus/volltexte/2007/962> [date of citation: 2007-01-01].

[17] R. Koschke and D. Simon. Hierarchical reflexion models. In *Working Conference on Reverse Engineering*, pages 36–45. IEEE Computer Society Press, Nov. 2003.

[18] P. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, Nov. 1995.

[19] A. Maccari and C. Riva. Architectural evolution of legacy product families. In *Proceedings of the Fourth International Workshop on Product Family Engineering (PFE-4)*, pages 63–68, 2001.

[20] G. C. Murphy and D. Notkin. Reengineering with reflexion models: A case study. *IEEE Computer*, 30(8):29–36, Aug. 1997. Reprinted in Nikkei Computer, 19, January 1998, p. 161-169.

[21] G. C. Murphy, D. Notkin, and K. J. Sullivan. Software reflexion models: Bridging the gap between design and implementation. *IEEE Computer Society Transactions on Software Engineering*, 27(4):364–380, Apr. 2001.

[22] D. Pollet, S. Ducasse, L. Poyet, I. Alloui, and S. C. H. Verjus. Towards a process-oriented software architecture reconstruction taxonomy. In *European Conference on Software Maintenance and Reengineering*, pages 137–148. IEEE Computer Society Press, 2007.

[23] C. Riva. *View-based Software Architecture Reconstruction*. Ph. d. dissertation, Vienna University of Technology, Vienna, Austria, Oct. 2004.

[24] C. Stoermer and L. O'Brien. MAP—mining architectures for product line evaluations. In *IEEE/IFIP Working Conference on Software Architecture*, pages 35–44. IEEE Computer Society Press, Aug. 2001.

[25] F. van der Linden. Software product families in europe. the Esaps and Cafe project. *IEEE Software*, 19(4):41–49, July/August 2002.

[26] Z. Xing and E. Stroulia. Understanding phases and styles of object-oriented systems' evolution. In *International Conference on Software Maintenance*, pages 242–251. IEEE Computer Society Press, 2004.