On the Diversity of Capturing Variability at the Implementation Level

Xhevahire Tërnava, Philippe Collet

On the Diversity of Capturing Variability at the Implementation Level

Xhevahire Tërnava and Philippe Collet
Université Côte d'Azur, CNRS, I3S, Sophia Antipolis, France
{ternava,collet}@i3s.unice.fr

ABSTRACT

In many Software product lines (SPLs), if domain variability can be properly specified in terms of features in a feature model (FM), their implementation in core-code assets is hard to capture and maintain, as there are different techniques to implement the variability. Even with an organization in variation points and variants, most of these techniques do not shape the code in terms of features, and inconsistencies appear when the variability evolves at one level with no co-evolution at the other. To help SPL architects, one possible solution is to be able to reconstruct the FM by capturing the variability in core-code assets, but different implementation techniques expose diverse characteristics, hampering the process. We study in this paper the diverse dimensions of the existing variability implementation techniques, and how they can be captured in an abstract way. We then categorize them regarding these dimensions in a single catalog, extending previous classifications of such techniques. We also briefly show how the characteristics of the techniques could help to better capture the implemented variability, opening some potential in reverse engineering processes.

ACM Reference format:
Xhevahire Tërnava and Philippe Collet. 2017. On the Diversity of Capturing Variability at the Implementation Level. In *Proceedings of SPLC '17, Sevilla, Spain, September 25-29, 2017,* 8 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In a Software product line (SPL) approach, the specified domain variability is commonly expressed by a variability model, *e.g.*, a feature model (FM) using the concept of features [17]. These features are realized in different core assets and in this work we consider their realization at the implementation level (*i.e.*, in core-code assets). As variability evolves over time, new features are introduced or the existing variability is extended. When the implemented and the specified variabilities in an FM do not co-evolve, their mapping may gradually deteriorate and inconsistencies appear. Thus, a reverse engineering approach is usually needed to reconstruct an approximation of the FM, or part of it, from the core-code assets, so to help in resolving inconsistencies.

Reconstructing the FM from the implemented variability in core-code assets is not trivial, as the code may not be shaped in terms of features when several traditional techniques are used (*e.g.*, inheritance, design patterns). Within the reverse engineering approaches, some aim to find the feature locations in different software variants, within a similar domain, and to migrate them to an SPL. According to a recent survey [3], source code is then the most used input artifact in such migration. On the other side, there are approaches that show how to reconstruct an FM from the described variability in a propositional formula [11]. In both cases, the resulting variability model is abstracted from the implementation techniques, *e.g.*, finding the feature locations through analysing the abstract syntax tree (AST) of code in several product variants [31, 34]. There are also approaches that describe how to *capture* the implemented variability and reconstruct the FM when a single technique is used to implement the variability, *e.g.*, using preprocessors in C as implementation technique [19]. But, the role and characteristics of variability implementation techniques in the reverse engineering processes are not considered. To understand how an implementation technique may support a reverse engineering process, we analyse in this paper what characterizes the capturing of variability in core-code assets, namely, how to identify and abstract the variability information that is realized in core-code assets. We also use an excerpt of the Graph SPL example [20] to illustrate the tackled issues. An extract of the FM for this SPL is shown on Figure 1. A possible implementation is shown in Listing 1 using two techniques, the strategy pattern and parameters in Scala [1]. We then define the issues to be handled as follow:

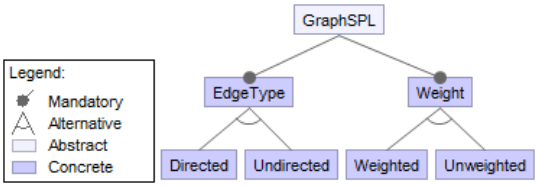

Figure 1: An excerpt of FM for Graph SPL

I1. Considering the variability implementation technique during the capturing of variability is important for several reasons, *e.g.*, except the feature names or feature locations, it is important to capture their relation logic, dependencies, and binding time to reconstruct the FM. For example, we should be able to reconstruct even a rough FM as the one in Figure 1 from the implemented variability in Listing 1.

I2. When variability is captured in core-code assets (*e.g.*, in Listing 1), abstracting it in terms of features or variation points with variants (defined in Section 2.1) is not similar.

I3. There is a large set of approaches that show how to evaluate and choose a technique to address some domain variability in core-code assets [2, 13, 22, 25, 29], but we are not aware of approaches that address explicitly the importance of techniques in the reverse engineering process.

```
1 object Conf {
```

[1] Our focus is on implementation techniques and not implementation languages.

SPLC '17, Sevilla, Spain
2017. 978-x-xxxx-xxxx-x/YY/MM…$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

```
  final val WEIGHTED: Boolean = true
}
abstract class Graph { /* Common part */ }
class ConcreteGraph extends Graph {
  def adddirectededge(s: Vertex, d: Vertex, w: Int) = {
    val edge = new Edge(s, d)
    if (Conf.WEIGHTED) {
      edge.weight = w
    }
    edges = edge :: edges
    addtoadjacencymatrix(edge)
  }
  def addundirectededge(s: Vertex, d: Vertex, w: Int) = {
    val edge1 = new Edge(s, d)
    val edge2 = new Edge(d, s)
    if (Conf.WEIGHTED) {
      edge1.weight = w
      edge2.weight = w
    }
    edges = edge1 :: edges
    edges = edge2 :: edges
    addtoadjacencymatrix(edge1)
    addtoadjacencymatrix(edge2)
  }
  def addedge(callback: (Vertex, Vertex, Int) => Unit,
      x: Vertex, y: Vertex, w: Int = 1) = callback(x, y, w)
```

Listing 1: Graph SPL implementation using two different techniques

To address these issues, we have to analyse how the variability can be implemented, *i.e.*, the diversity of variability implementation techniques, and the characteristic properties of variability abstractions, *i.e.*, features and variation points with variants, that are important to be captured in core-code assets (Section 3). We then analysed the diversity of 21 techniques, which are evaluated regarding the capturing properties of variability abstractions and are shown in form of a catalog (Section 4). We finally show how the characteristics of the techniques could help to better capture the implemented variability, opening some potential in reverse engineering processes (Section 5).

2 VARIABILITY REALIZATION

2.1 Variable Parts in Core-Code Assets

In realistic SPL settings, the implementation of variability in core-code assets comply, mostly, to a commonality and variability approach, regardless of the programming paradigm (*e.g.*, object-oriented, or functional) [9]. Specifically, a domain is decomposed into sub-domains, then within each sub-domain the commonality is factorized from the variability that is used to differentiate the software products within the domain.

The core-code assets consist of three parts: the core, commonalities, and variabilities. The core part is what remains of the system in the absence of any particular feature [32], *i.e.*, the assets that are included in any software product of the SPL. A commonality is the common part for the related variant parts within a sub-domain. After the commonality is factorized from the variability and implemented it becomes part of the core, *i.e.*, it is buried in the core [9], except when it represents some optional variability [30]. The variant parts are used to distinguish the software products in the domain. A sub-domain can have more than one common and variant part. The core with the commonalities and variabilities of all sub-domains constitute the wholeness of core-code assets in an SPL.

The commonalities and variabilities in core-code assets are commonly abstracted in terms of variation points (*vp*-s) and variants, as solution oriented abstractions. Unlike features in problem space, *vp*-s and variants are related to concrete elements in core-code assets. Originally, a variation point identifies one or more locations at which the variation will occur [16]. Respectively, variation points are known as a manifestation of variability in architecture and design [12], while the way that a variation point is going to vary is expressed by its variants. The *variable part* is like an organizing container [4]. It contains the location of variability in core-code assets (*i.e.*, the variation point), the variants, and the used technique to implement them.

Besides, there are approaches that use the concept of feature also at the implementation level, *e.g.*, feature modules [2], or the *vp*-s and variants at the specification level, *e.g.*, *vp*-s are known as the varying place in the FM [10]. In this work, unless explicitly specified, we consider that *vp*-s and variants are solution oriented abstractions, while features are problem oriented abstractions.

2.2 Variability Implementation Techniques

The techniques used for realizing the variability in an SPL are called variability realization techniques [8, 29], variability mechanisms [4, 16, 22], or variability implementation techniques [12]. *Variability realization technique* is a general term used for techniques acting at the architecture, design, or code level, while the term *variability implementation technique* will be only used for techniques at the code level, being the focus of our work. All variability implementation techniques came from several programming paradigms and are supported by different constructs in different programming languages, which in turn offer different properties.

An SPL is structured around a set of features of several kinds, which represent different functional or non-functional software products' requirements. These features have to be properly implemented and their reflection to *vp*-s and variants is manifold in architecture, design, and especially in implementation. In principle, one *vp* is associated with only one technique while the same technique can be used to implement several *vp*-s within a domain. Examples of variability implementation techniques are inheritance, preprocessors, feature modules or some design patterns.

2.3 Existing Guidance

An important issue for variability implementation is the ability to evaluate and choose a technique which will fulfill best some given variability requirements. Several evaluation schemas of various techniques are currently available with such aim. They mainly came from academia in form of *frameworks*, *taxonomies*, and *catalogs*.

Svahnberg et al. [29] group and organize the techniques by two dimensions of similarity: according to the size of software entities with variability or variable (components, frameworks, and lines of code), and their latest binding time. Examples of concrete techniques categorized by this taxonomy have been given, but for the majority of the other techniques it is up to the user to evaluate and categorize them.

Patzke and Muthig [25] gather several techniques and evaluate them by three types of variability (optional, alternative, and multi-coexisting). They also give a model that captures how and which technique is appropriate to be introduced [23] depending on the maturity level of the SPL [1, 6]. Concrete examples on each technique and other details have been complemented in an enriched set

of techniques and evaluation criteria [24]. Quite similarly, Muthig and Patzke [22] evaluate a slightly different set of techniques and criteria. Gacek et al. [13] discuss also a rich set of techniques and criteria. They are organized in a catalog and evaluated by the proposed criteria. A second catalog gives details on the language support for each technique. However, their evaluations are diverse, by different criteria, and for different subsets of techniques.

Coplien addresses many issues for realizing the commonalities and variabilities [9], although with no direct emphasis on SPL engineering. He introduces several techniques from different programming paradigms for variability implementation, and enlightens how these techniques factorize the commonality and accommodate the variability on it.

Apel et al. [2] recently cover the majority of the identified variability implementation techniques and study a set of criteria for each of them. Although a deeper discussion about techniques is given, they are not organized in a catalog nor compared, while only a subset of previously used evaluation criteria is considered.

The work of Fritsch et al. [12] is of a different nature. They propose an evaluation framework for techniques. Several evaluation criteria are determined, but it is up to the developers to evaluate the techniques by these criteria and to create their own catalog.

As the SPL engineering field is making progress, various techniques are regularly identified. Considering all the work mentioned above, we observe that they cover different subsets of techniques and use different evaluation methodologies and criteria. They study them from the perspective of how well they fit to implement some variability, but not how well they support the capturing of variability during a reverse engineering process.

3 DIMENSIONS OF DIVERSITY

A lot of knowledge is gathered about features of software products in an SPL during domain analysis. Just a part of that knowledge is modeled in the FM and the rest remains implicit or is kept in informal ways. Specifically, an FM represents only the parent-child hierarchy of features (*specialization/generalization*, *consist-of*), *e.g.*, `Directed` specialize/generalize `EdgeType`, *(ii)* the logical relation between features (*mandatory*, *optional*, *or*, *alternative*), *e.g.*, `Directed` with `Undirected` are alternative, and *(iii)* possibly some cross-tree constraints (*requires*, *mutual exclusion*) that are expressed in propositional logic (*cf.* Figure 1).

However, *vp*-s and variants in core-code assets are characterized by a richer set of characteristic properties, which are additional to the documented knowledge of features in the FM. These *vp*-s and variants with their properties are important to be captured during a reverse engineering process, *e.g.*, the logic relation between variants, their binding time are important to reconstruct the FM and to resolve variability. For example, in Listing 1 two *vp*-s and four variants can be captured. The `callback` function in line 26-27 is the *vp* `vp_edgetype` and the parameter `WEIGHTED` in line 2 is captured as another *vp* `vp_weight`. Then, the methods in lines 6-13 and 14-25 are captured as variants `v_directed` and `v_undirected`, respectively, for the first *vp*. Similarly, the `true` and `false` values in line 2 are captured as variants `v_weighted` and `v_unweighted`, respectively, for the second *vp*.

Table 1: Logic Relations of *vp*-s and of variants in a *vp*

Logic Relation	Description
Mandatory	*The vp or variant is part of each software product*
Optional	*The vp or variant can be part of the software product or not*
Multi Coexisting (Or)	*One or more than one of the variants in a vp can be part of the software product*
Alternative (Xor)	*Only one of the alternative variants in a vp can be part of the software product*

Table 2: Binding times of variation points

Binding	Values	Description
Static binding (S)	(S) compilation / link (S) build / assembly (S) programming (S/D) configuration (S/D) (re) deploy	*The variability is resolved early during the development cycle, i.e., the decision for a variant in a variation point is made early/statically.*
Dynamic binding (D)	(D) runtime (start-up) (D) pure runtime (operational mode)	*The variability is resolved later during the development cycle, i.e., the decision for a variant in a variation point is made as late as possible/dynamically.*

3.1 Characteristic Properties of Variable Parts

In the following, we gather the important characteristic properties of *vp*-s and variants that should be considered during the capturing of variability in core-code assets. These properties are diverse, as they depend on the used implementation techniques.

Logic Relation. Similar to the possible relations between features in an FM (*cf.* Figure 1), the logic relations of *vp*-s and of their variants are shown in Table 1. A single technique can offer at least one of these logical relations, *e.g.*, the inheritance can be used for implementing the alternative variants, overriding for implementing the multi-coexisting variants, or aggregation for optional variants. Concretly, in Listing 1, we capture the alternative logical relation between variants `v_directed` and `v_undirected`, which is realized by the strategy pattern.

Binding Time. Within a domain, *vp*-s may require being resolved at different development phases. Thus, each *vp* is associated with a binding time, *i.e.*, the time when the variability is decided or the *vp* is resolved with its variants, and it should be supported by the chosen variability implementation technique. A *vp* can be resolved early during the development cycle (*e.g.*, when the decision for a variant is made at compile time), or later during the development cycle (*e.g.*, dynamically, at runtime) [6]. The binding time here is the time when the variability should be resolved and should not be confused with the time when it will be introduced. As for the common kinds of binding times, a taxonomy is given by Capilla et. al and Bosch et. al [7, 8]. They are also shown in Table 2. For example, in Listing 1, we can capture that the `vp_edgetype` is bound to one of its variants, *e.g.*, `v_directed`, during runtime.

Defaults. Some variability may not be subject to frequent variations among the majority of software products in an SPL. In such cases, one of the variants on a *vp* can be set as its *default variant* [9]. For example, we capture `v_unweighted` as the default variant of

Table 3: Granularity of variants

Granularity	Values	Description
Coarse-grained	Component, framework with plug-ins as variants, file, package, class, interface, frame, feature module, etc..	*The specified variability has an effect in the coarsest elements of the implementation structure.*
Medium-grained	Method, field inside a class, aspect, delta module, frame, etc..	*The specified variability has an effect in the medium sized elements of the implementation structure.*
Fine-grained	Expression, statement, block of code within a method, frame, etc..	*The specified variability has an effect in the finest grained elements of the implementation structure.*

`vp_weight`. It is realized by setting the argument `w: Int = 1` (line 27).

Granularity. A *vp* or variant in the core-code assets can have different granularities depending on the size of variability and the used technique (*cf.* Tbl. 3). They can represent a coarse-grained element that is going to vary *e.g.*, a file, package, class, interface; a medium-grained element *e.g.*, a method, a field inside a class; or a fine-grained element *e.g.*, an expression, statement, or block of code. For example, we should be able to capture the `v_directed` on method level or the `vp_weight` as a parameter.

Evolution. Depending on whether the specified variability in the FM is meant to be evolved with new features, *vp*-s can be *open* or *closed*, *i.e.*, to be extended with new variants in the future or not, respectively. For example, we capture a *vp* as closed when it is implemented as an `enum` type in Java, and open when it is implemented simply as an abstract class.

Quality Criteria. Variability implementation techniques are supported by different constructs among potentially various programming languages, thus providing different qualities for the resulting implemented variability. A dominant quality criterion is the ability to shape the code (*i.e.*, its variability) in terms of features as cohesive reusable units so to handle more easily the variability among distinct abstraction levels. Several other quality criteria are introduced in the literature [2, 12, 13, 24]. What we consider the most important ones for evaluating techniques are shown in Table 4. For example, using the strategy pattern requires more preplanning effort than using parameters in Listing 1.

3.2 Classifications of Techniques

Due to the evolving diversity of variability implementation techniques, researchers have grouped them differently and mentioned dissimilar subsets of them. In general, implicitly or explicitly, all variability implementation techniques are found to be classified based on three orthogonal sub-dimensions: *(i)* traditional or emerging, *(ii)* language-based or tool-based, and *(iii)* annotative or compositional.

The first sub-dimension depends on the time when a technique has emerged and whether it is dedicated to the variability implementation in core-code assets.

Traditional techniques. They have emerged and evolved separately and quite before the emergence of the SPL paradigm. They encompass methods that are used for single system development but provide the necessary mechanisms to be good candidates for SPL engineering (*e.g.*, inheritance, overloading, generic types). Consequently, in all these techniques, the concept of feature does not have a first-class representation in implementation.

Emerging techniques. On the contrary they have emerged with the SPL engineering advances. Here, the concept of feature is a first-class citizen at the code level. In our study, all these techniques, *e.g.*, frames [5], feature modules [2], delta modules [27], come from academia.

Some of the techniques only depend on language mechanisms while others rely on extra tool support. Based on [2], we introduce the following sub-dimension of classification:

Language-based techniques. The variability is realized and resolved by different and dedicated language constructs or mechanisms. Examples of these techniques are inheritance, feature modules, aspects, and delta modules.

Tool-based techniques. In this case specialized tools are used to identify and resolve variability among the software assets. Although they are conceptually independent of any given language and orthogonal to its constructs, currently they are only supported by specific programming environments (*e.g.*, frames [5]).

The third sub-dimension is about the now common distinction on the way that variability is represented and resolved at code level.

Annotative techniques. The specified features are realized in core-code assets as a whole and the variants are annotated by a technique in order to include or exclude them during variability resolution. Different variability implementation techniques have different means for realizing annotations, *e.g.*, preprocessor directives in C, or simple tagging approach [15], but this falls into the tool-based approaches and not techniques. When an annotative technique is used to remove the unneeded variants from core-code assets, it supports *negative variability*.

Compositional techniques. Variants aim at having a cohesive or modular representation at code level, *e.g.*, in form of components, plug-ins, classes, packages, modules, aspects, deltas, subjects, etc. The intention is that a final product is derived by simply attaching or composing any of these modular units or variants in the base assets as part of core-code assets that are included in all software products. Therefore, these techniques are known to support *positive variability*, *e.g.*, the technique behind feature modules. In the last sub-dimension, negative or positive variability may sometimes be orthogonal to this classification (*e.g.*, deltas [27]).

Another very notable set of techniques is the one that follows a *generative approach* [10]. These techniques have the ability to generate software products from the generic assets. Respectively, they may use a description in a higher abstraction language level, such as a Domain Specific Language (DSL), for deriving software products [33]. We see these techniques as orthogonal to our issue of capturing variability.

Table 4: Quality criteria of variability implementation techniques

Quality	Description
Preplanning effort	*The required preplanning effort to introduce and use a variability implementation technique.*
Visibility of *vp*	*Variation points in code can be* ***explicit***, ***implicit***, e.g., *in clonning technique[24], or* ***ambiguous***, i.e., *in traditional techniques when the same mechanism is used for implementing the variability and the functionality of software.*
Information hiding	*It enables the modular reasoning of variability in implementation through separating variable modules in internal and external parts.*
Uniformity	*Some techniques which are used for variability in the implementation level can also be applied to realize the variability to the other not code artifacts among the other abstraction levels.*
Separation of Concerns (SoC)	*While the main concerns in an SPL development are features, SoC is related here to the ability to shape the code in terms of (separate) features*
Traceability	*The ability to trace features in their development lifecycle, specifically with the implemented artifacts.*
Scalability	*The ability of a technique to support the requirements during an extension/evolution of variability.*

4 CATALOG BUILDING METHOD

In this section we describe both the building method and resulting catalog of variability implementation techniques regarding the characteristic properties that can be captured.

4.1 Covered Techniques

To have an up-to-date catalog of implementation techniques, we collected the identified variability implementation techniques so far. They are diverse and spread among different research works and shared experiences, forming a significant set. It must be noted that we did not follow a systematic process in our literature review, taking the existing classifications and related references as starting points for finding relevant research publications. Then, we decided to consider only those techniques that are used in a closed-world SPL engineering process [18], *i.e.*, when a technique is used for implementing a set of features with a closed view within a domain. Different programming languages support subsets of them, and even for a single technique, different mechanisms or language constructs can be used. We excluded techniques like components and frameworks as they already use the considered closed-world techniques to achieve their variability.

4.2 Evaluation Process

We then rely on the previously introduced dimensions with their subdimensions as their evaluation criteria. Next we performed the evaluation of each technique by applying the following two methods in parallel.

First process. We used four small case studies from different domains to experiment with several techniques and *vp*-s characteristic properties. Those case studies are: *Arcade Game Maker SPL* [28], *Microwave Oven SPL* [14], *Expressions SPL* [21], and *Graph SPL* [2, 20] [2]. Their domains are quite well understood and used by the SPL community. For their implementation we used the Scala language, which supports both the object-oriented and functional paradigms and has rich constructs for modularization of features. This enables us to cover more technical choices with only a single implementation language.

Second process. We built an informed opinion for each technique and some of their criteria from the existing research works, some of them are presented in section 2.3. These works mainly evaluate a technique using Java and C++ language constructs or extensions of these languages, *e.g.*, AspectJ. We thus examined manually the realized evaluations and the obtained results by the others. Similar to us, most of them use three evaluation levels for a criteria, *e.g.*, good support, possible support, and no support. After the examination step, we compared the results from these works, especially when they were considering the same criteria for a same technique. Whenever two different works do not agree on a value for a specific criterion, we used one of our case studies to evaluate it, and we studied the usage of that technique. As a result, each technique was systematically evaluated by the mentioned criteria in the three classification dimensions.

4.3 Resulting Catalog

The evaluated techniques are shown organized in the proposed catalog in Table 5 [3]. Techniques are gathered in two main groups, those that provide *ad-hoc reuse* or *methodological reuse*. We did not want to exclude the most applied techniques, *e.g.*, *copy-and-paste* or *cloning*, from our classification. These techniques have well-known drawbacks and may not scale due to their *ad hoc* form of reuse. However, this is strongly related to the maturity levels of an SPL [1, 6] and practical for building quick solutions [23], which can be refactored later. The other techniques encourage the methodological reuse as the main ingredient for a sustainable SPL.

The catalog contains two *Legends*. The first legend is used to explain the evaluation results and the second one, *i.e.*, Legend B, is used to explain our evaluation method using previous works. Some evaluation values to a criteria are associated with a reference work, using numbers, to show when a result is influenced by an existing work and by which one. For example, the technique of *Frames* for the criterion of *Binding Time* (see the colored intersection in Table 5) is evaluated as offering static binding of variants. This result is supported by three references, *i.e.*, Gacek (2001) [13] and Patzke (2003, 2011) [24, 26], refereed by numbers 2, 5, and 7 respectively. In Table 5 only some of the main works that we used are shown, as they cover a considerable set of techniques. These references

[2] Their implementation are available on https://github.com/ternava/variability-cchecking

[3] See also in our repository https://github.com/ternava/expressions_spl/wiki.

can be used to show how the evaluated techniques and the used criteria are distributed among the previous research works.

5 CAPTURING VARIABILITY

In the following we illustrate and compare the capture of variability for the Graph SPL (*cf.* Figure 1) in two implementation cases, when variability is implemented by using the traditional techniques of strategy pattern with parameters (*cf.* Listing 1) and the emerging technique of feature modules (*cf.* Listing 2). To save space, in Listing 2 we have shown only the main organization of feature modules. Their complete implementation is available elsewhere [2].

```
1  layer BasicGraph;
2  class Graph {
3    Vector nodes = new Vector();
4    Vector edges = new Vector();
5    Edge add(Node n, Node m) {
6      Edge e = new Edge(n, m);
7      nodes.add(n);
8      nodes.add(m);
9      nodes.add(e);
10     return e;
11   } /* ... */
12 class Edge {    /* ... */ }
13 class Node {    /* ... */ }
14 }
15
16 layer Directed;
17  class Graph { /* ... */ } /* ... */
18
19 layer Undirected;
20 class Graph { /* ... */ } /* ... */
21
22 layer Weighted;
23 class Graph { /* ... */ }
24 class Edge { /* ... */ }
25 class Weight { /* ... */ }
```

Listing 2: The Graph SPL using feature modules

Capturing Vp-s and Variants. In Section 3, we have shown how we capture the variability in terms of *vp*-s and variants from Listing 1. The whole captured variability is given again in Table 6. The catalog of techniques can then help one to capture the right characteristic properties for each used technique. For example, the strategy pattern (*cf.* Table 5) can offer an alternative relation between variants, runtime binding, default variants, and the possibility to add new variants during the evolution. All these properties are consistent with the captured variability of `vp_edgetype`, and its variants, which are implemented by the strategy pattern (*cf.* Table 6).

Capturing Features. In Listing 2 the code is shaped in terms of features by using the technique of feature modules (called *layer*). In this implementation, each problem space feature (*cf.* Figure 1) is implemented by a feature module (*cf.* Listing 2), while their mapping is ensured by their common names. The module `BasicGraph` is the base module where can be added other feature modules as variants during the product derivation. The variability that can be captured in Listing 2 is the base module, which is mandatory, and 3 variants (`Directed`, `Undirected`, and `Weighted`) that are optional with the same binding time (*deployment*), granularity (*feature module*), and no evolution or default concepts. Moreover, the feature `Unweighted` is made default by being part of the base module. In addition there is no concept of *vp*-s.

To reflect on the differences between capturing *vp*-s with variants and features in core-code assets, we show that in Listing 1 we can capture, still, features instead of *vp*-s with variants. For

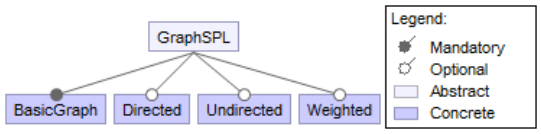

Figure 2: The reconstructed FM from features in Listing 2

example, feature `Directed` can be captured in lines 6 – 7, 11 – 13. Feature `Weighted` can be captured in lines 2, 6, 8 – 10, 14, 17 – 20. If we compare these lines of code with the lines of the captured *vp*-s and variants in Table 6, it is obvious that capturing *vp*-s with variants and features is different. Thus, by capturing features in core-code assets we capture only those lines of code that implement a problem oriented feature, while by capturing *vp*-s and variants we capture the implementation technique that gives a rich set of properties to them (*cf.* Table 6).

Reconstructing the Feature Model. As for reverse engineering, the implementation techniques can be assessed regarding their support to reconstruct an FM from the captured variability in core-code assets. This also points out the differences between the captured variability in terms of features and *vp*-s with variants.

For example, from the captured variability in Table 6 we can reconstruct an FM similar to the original FM of Figure 1 using the domain specific language (DSL) we previously proposed [30]. It notably provides mechanisms to capture and document the implemented variability in terms of *vp*-s and variants, thus helping to reconstruct a variability model. The only difference between these feature models is that feature `Unweighted` is realized as a default variant `v_unweighted`. The successful reverse engineering of the FM from this captured variability is made possible mainly because it is known the logical relations between *vp*-s and variants (*cf.* Table 1). The other captured properties can be used for other reasons, *e.g.*, the binding time during the product derivation.

Similarly, in Figure 2 is shown the reconstructed FM from the captured features and their properties in Listing 2. This reconstructed FM is hardly similar to the original FM in Figure 1 because the feature modules expose a single relation logic between them. These examples indicate that, depending on the implementation technique, we will be able to capture variability and reconstruct a good approximation of the FM, with the original one, or not.

6 DISCUSSION AND FUTURE WORK

In this paper, we analysed the variability implementation techniques and their diverse properties that can be captured during reverse engineering (*I3*). In particular, we showed that capturing the variability in core-code assets may depend on the implementation technique (*I1*). We studied the diversity of some well known techniques and provided a catalog that classifies them. As a result, each technique supports the capturing of variability in terms of features or *vp*-s with variants. In contrast to features, *vp*-s with variants can have different properties which depend on the used technique. By capturing their properties we could reconstruct a better approximation of the FM, from the core-code assets, compared with the original one. We also showed that both features and *vp*-s with variants can be used to capture variability in core-code assets but, although their meaning overlaps, they are not the same (*I2*).

Legend A:
●: good support / belong
◐: possible support (difficult)
○: no support (not often applicable) / does not belong
✱: high; ◖ : average; ◁ : low
E: explicit; **A**: ambiguous

	Feature types			Binding time			Open for evolution	Granularity			Preplanning effort	Visibility of *vp-s*	Information hiding	Uniformity	Sep. of concerns	Traceability	Scalability	Language paradigm	Annotative	Compositional	Language-based	Tool-based	Traditional	Emerging
	Optional	*Or*	*Alternative*	*Static* (S)	*Dynamic* (D)	**Defaults**		*Coarse*	*Medium*	*Fine*														
AD-HOC REUSE																								
Cloning / Patching	●	●	●	●	○	○	●	●	●	●	◁	A/E	○	●	○	○	○	Not specific	●	●	●	○	●	○
				7	7	7	7	7				7	7		7	7	7	7						
Conditional Execution (Parameters)	●	○	●	◐	●	○	○	○	◐	●	◁	A	◐	○	◐	◐	○	Not specific	●	○	●	○	●	○
	7	7	7	1,7	1,7	7	7	1,7	1,7	1,7	1	7	1	1	1,7	1,7	1	7	1	1	1	1	1	1
METHODOLOGICAL REUSE																								
Preprocessor directives	●	◐	●	●	○	◐	○	●	●	●	◁	A/E	○	●	○	◐	◐	Not specific	●	○	○	●	●	○
	2,3,4,7	2,4	1,2,3,4,7	1,2,7	1,2,7	6,7	7	1,3,4,6,7	1,3,4,6,7	1,3,4,6,7	1,7	7	1	1	2,7	1,2	2,7	7	1,2,4,6	1,4,6	1,3,4,6	1,3,4,6	1,3	1,3
Argument defaulting	●	○	○	●	○	●	○	○	●	○	◖	A	◐	○	○	◐	◐	Not specific	●	○	●	○	●	○
	4	4	4			6													3,4,6	3,4,6			3	3
Overriding	◐	●	○	○	●	●	●	○	●	○	◁	A	●	○	●	●	●	O. Oriented	○	●	●	○	●	○
Aggregation / Delegation	●	◐	◐	●	●	●	◐	●	●	○	◖	E	◐	○	●	◐	◐	O. Oriented	●	●	●	○	●	○
	2	2	2	2	2										2	2	2	2	2	2				
Inheritance	●	●	◐	●	●	●	●	●	●	○	◖	A	●	○	●	●	◐	O. Oriented	◐	●	●	○	●	○
	2	2	2	2,6	2,6			6	6	6			6	1	2	2	2	6	2,6	2,6	6	6		
Reflections	●	●	●	◐	●	●	◐	●	●	○	✱	E	●	○	●	●	●	O. Oriented	○	●	●	○	●	○
	2	2	2	2	2										2	2	2							
Aspects	◐	●	◐	●	◐	●	●	◐	●	○	◖	A	○	◐	●	●	●	Aspect Ori.	◐	●	●	○	○	●
	2,4	2,4	2,4	1,2,7	1,2,7	7	7	1,7	1,7	1,7	1,7	1,7	1	1	1,3,7	1	2,7	1,3	2	1,2,3	1,3,7	1,3,7	1,3	1,3
Polymorphism																								
Coercion (Casting)	◐	◐	◐	●	○	●	●	●	○	○	✱	A	●	○	●	●	●	O. Oriented	○	●	●	○	●	○
	4	4	4			4												3,4					3	3
Overloading	○	○	●	●	○	●	●	○	●	○	◖	A	●	○	◐	◐	◐	Procedural	●	●	●	○	●	○
	2,4	2,4	2,4	2,6	2,6			6	6	6					2	2	2	3,4	2,6	2,6			3	3
Subtype polymorphism	◐	◐	●	◐	●	●	●	●	●	○	✱	A	●	○	●	●	●	O. Oriented	○	●	●	○	●	○
	4,7	3,4	3,4,7		3,4,7	6	7	7	7	7		7			7			3,4		7	7	7	3	3
Parametric polymorphism (generics)	◐	●	●	●	○	○	○/●	●	●	○	✱	E	●	○	◐	●	◐	Generic prog.	●	●	●	○	●	○
	3,4,7	3,4	3,4,7	3,4,6,7	3,4,6,7	7		6	6,7	6,7		6,7						3,4	6	7	4,7	4,7	3	3
Design patterns																								
Strategy pattern	●	●	●	◐	●	●	●	●	●	○	✱	A	●	○	●	●	●	O. Oriented	○	●	●	○	●	○
	1	1	1	6	1,6	1			1		1		1	1	1	1			1	1	1	1	1	1
Decorator pattern	●	●	○	◐	●	●	●	●	●	○	✱	A	●	○	●	●	●	O. Oriented	○	●	●	○	●	○
	1	1			1						1		1	1	1	1								
Observer pattern	●	●	●	◐	●	●	●	●	●	○	✱	A	●	○	●	●	●	O. Oriented	●	●	●	○	●	○
					1						1		1	1	1	1			1	1	1	1	1	1
Template method pattern	●	◐	●	◐	●	●	●	●	●	○	✱	A	◐	○	●	●	●	O. Oriented	○	●	●	○	●	○
	1	1	1	6	1,6	1			6		1		1	1	1	1			1	1	1	1	1	1
Visitor pattern	◐	●	◐	◐	●	●	◐	●	●	○	✱	A	●	○	●	●	●	O.Oriented	○	●	●	○	●	○
Emerging techniques																								
Frames	●	●	●	●	○	●	○ /●	●	●	●	◖	E	○	◐	●	●	◐	Not specific	●	●	○	●	○	●
	2,5	2,5	2,5	2,5,7	2,5,7	5,7	7	5,7	5,7	5,7	2,7	5,7		1	2,7	2	2,7	5,7	2,5	2,5	5,7	5,7		
Feature Modules	◐	●	◐	●	◐	●	●	●	●	○	◖	A	○	●	●	●	●	Feature Ori.	○	●	●	○	○	●
	4	1,3,4	4	1	1			1	1	1	1		1	1	1,3	1		1	1	1,3	1,3	1,3	1,3	1,3
Delta Modules	◐	●	◐	●	○	●	●	●	●	○	◖	A	○	◐	●	●	●	Delta Ori.	●	●	●	○	○	●
				1				1	1	1								1	1	1	1	1	1	1

Legend B: **1** → Apel [2]; **2** → Gacek [13]; **3** → Muthig [22]; **4** → Patzke [25]; **5** → Patzke [26]; **6** → Coplien [9]; **7** → Patzke [24]

Table 5: Catalog of variability implementation techniques

Table 6: Capturing the implemented variability of Graph SPL in Listing 1

VP-s	Lines	Granularity	*Variants*	Lines	Granularity	Binding time	Logic relation	Default	Evolution
vp_edgetype	26 – 27	argument	v_directed	6 – 13	method	runtime	alternative	No	Open
			v_undirected	14 – 25	method			No	
vp_weight	2	parameter	v_weighted	2	value	programming	alternative	No	Close
			v_unweighted	2	value	time		Yes	

We used this analysis on the diversity of variability implementation techniques to build a framework for capturing and tracing the implemented variability [30]. The framework is implemented as an internal domain specific language (DSL) in Scala and uses reflection to capture variability in terms of *vp*-s and variants. It is mainly applicable for capturing the variability when it is implemented using traditional techniques, *e.g.*, inheritance, design patterns. The DSL, our four case studies, and some examples of using the DSL to capture their variability are available at https://github.com/ternava/variability-cchecking.

We are planning to demonstrate in more details the usage of the catalog. In particular, we are going to investigate how to capture the *vp*-s and variants and reconstructing the feature model in real case studies where several variability implementation techniques are used in combination, *e.g.*, inheritance, generic types, design patterns. We plan also to study the usage of the catalog to choose a technique for refactoring some variability in form of a *vp* and variants from a set of product variants during their migration as an SPL. We expect that the evaluation and choice of a variability implementation technique will also be made based on how well it is going to support a reverse engineering process, *i.e.*, on how easy it will be to reconstruct variability models from core-code assets.

REFERENCES

[1] Michał Antkiewicz, Wenbin Ji, Thorsten Berger, Krzysztof Czarnecki, Thomas Schmorleiz, Ralf Lämmel, ÈŸtefan Stănciulescu, Andrzej Wąsowski, and Ina Schaefer. 2014. Flexible product line engineering with a virtual platform. In *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, 532–535.

[2] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines*. Springer, Heidelberg Dordrecht London.

[3] Wesley Klewerton Guez Assunção and Silvia Regina Vergilio. 2014. Feature location for software product line migration: a mapping study. In *Proceedings of the 18th International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools-Volume 2*. ACM, 52–59.

[4] Felix Bachmann and Paul C Clements. 2005. *Variability in software product lines*. Technical Report CMU/SEI-2005-TR-012. DTIC Document.

[5] Paul G Bassett. 1996. *Framing software reuse: lessons from the real world*. Prentice-Hall, Inc., New Jersey.

[6] Jan Bosch. 2000. *Design and use of software architectures: adopting and evolving a product-line approach*. Pearson Education.

[7] Jan Bosch and Rafael Capilla. 2012. Dynamic Variability in Software-Intensive Embedded System Families. *Computer* 45, 10 (2012), 28–35.

[8] Rafael Capilla, Jan Bosch, and Kyo-Chul Kang. 2013. *Systems and Software Variability Management*. Springer, Berlin.

[9] James O Coplien. 1999. *Multi-paradigm Design for C+*. Addison-Wesley.

[10] Krzysztof Czarnecki and Ulrich W. Eisenecker. 2000. *Generative programming: methods, tools, and applications*. Addison-Wesley.

[11] Krzysztof Czarnecki and Andrzej Wasowski. 2007. Feature diagrams and logics: There and back again. In *Software Product Line Conference, 2007. SPLC 2007. 11th International*. IEEE, 23–34.

[12] Claudia Fritsch, Andreas Lehn, and Thomas Strohm. 2002. Evaluating variability implementation mechanisms. In *Proceedings of International Workshop on Product Line Engineering (PLEES)*. sn, 59–64.

[13] Critina Gacek and Michalis Anastasopoules. 2001. Implementing product line variabilities. In *ACM SIGSOFT Software Engineering Notes*, Vol. 26. ACM, 109–117.

[14] Hassan Gomaa. 2005. *Designing software product lines with UML*. IEEE.

[15] Patrick Heymans, Quentin Boucher, Andreas Classen, Arnaud Bourdoux, and Laurent Demonceau. 2012. A code tagging approach to software product line development. *International Journal on Software Tools for Technology Transfer* 14, 5 (2012), 553–566.

[16] Ivar Jacobson, Martin Griss, and Patrik Jonsson. 1997. *Software reuse: architecture, process and PUBLISHER for business success*. ACM Press/Addison-Wesley Pub. Co., New York.

[17] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. 1990. *Feature-oriented domain analysis (FODA) feasibility study*. Technical Report CMU/SEI-90-TR-21. Carnegie Mellon Univ.

[18] Christian Kästner, Sven Apel, and Klaus Ostermann. 2011. The road to feature modularity?. In *Proceedings of the 15th International Software Product Line Conference, Volume 2*. ACM, 5.

[19] Duc Minh Le, Hyesun Lee, Kyo Chul Kang, and Lee Keun. 2013. Validating consistency between a feature model and its implementation. In *International Conference on Software Reuse*. Springer, 1–16.

[20] Roberto E Lopez-Herrejon and Don Batory. 2001. A standard problem for evaluating product-line methodologies. In *Generative and Component-Based Software Engineering*. Springer, 10–24.

[21] Roberto E Lopez-Herrejon, Don Batory, and William Cook. 2005. Evaluating support for features in advanced modularization technologies. In *European Conference on Object-Oriented Programming*. Springer, 169–194.

[22] Dirk Muthig and Thomas Patzke. 2003. Generic implementation of product line components. *Objects, Components, Architectures, Services, and Applications for a Networked World* (2003), 313–329.

[23] Thomas Patzke. 2010. The Impact of Variability Mechanisms on Sustainable Product Line Code Evolution. In *Software Engineering*. 189–200.

[24] Thomas Patzke. 2011. *Sustainable Evolution of Product Line Infrastructure Code (Experimental Software Engineering)*. Ph.D. Dissertation. Fraunhofer IRB Verlag.

[25] Thomas Patzke and D. Muthig. 2002. *Product line implementation technologies. Programming language view*. Technical Report 057.02/E. Fraunhofer IESE.

[26] Thomas Patzke and Dirk Muthig. 2003. *Product line implementation with frame technology: A case study*. Technical Report 018.03/E. Fraunhofer IESE.

[27] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. 2010. Delta-oriented programming of software product lines. In *Software Product Lines: Going Beyond*. Springer, 77–91.

[28] Software Engineering Institute SEI. 2009. Arcade Game Maker, pedagogical product line. (2009). Retrieved June 15, 2017 from http://www.sei.cmu.edu/productlines/ppl/

[29] Mikael Svahnberg, Jilles Van Gurp, and Jan Bosch. 2005. A taxonomy of variability realization techniques. *Software: Practice and Experience* 35, 8 (2005), 705–754.

[30] Xhevahire Tërnava and Philippe Collet. 2017. Tracing Imperfectly Modular Variability in Software Product Line Implementation. In *International Conference on Software Reuse*. Springer, 112–120.

[31] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. 2014. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming* 79 (2014), 70–85.

[32] C Reid Turner, Alfonso Fuggetta, Luigi Lavazza, and Alexander L Wolf. 1999. A conceptual basis for feature engineering. *Journal of Systems and Software* 49, 1 (1999), 3–15.

[33] David M Weiss and others. 1999. *Software product-line engineering: a family-based software development process*. Addison-Wesley Professional; Har/Cdr edition, Reading, Massachusetts.

[34] Tewfik Ziadi, Christopher Henard, Mike Papadakis, Mikal Ziane, and Yves Le Traon. 2014. Towards a language-independent approach for reverse-engineering of software product lines. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. ACM, 1064–1071.