A framework for managing the imperfect modularity of variability implementations

Xhevahire Tërnava, Philippe Collet

A Framework for Managing the Imperfect Modularity of Variability Implementations

Xhevahire Tërnava[a,*], Philippe Collet[b]

[a] Université de Rennes 1, INRIA/IRISA, Rennes, France
[b] Université Côte d'Azur, CNRS, I3S, Sophia Antipolis, France

ARTICLE INFO

Keywords:
Software product line engineering
Variability management
Variability implementation
Imperfectly modular variability
Domain specific language

ABSTRACT

In many industrial settings, the common and varying features of related software-intensive systems, as their reusable units, are likely to be implemented by a combined set of traditional techniques. Features do not align perfectly well with the used language constructs, *e.g.*, classes, thus hindering the management of implemented variability. Herein, we provide a detailed framework to capture, model, and trace this imperfectly modular variability in terms of variation points with variants. We describe an implementation of this framework, as a domain-specific language, and report on its application on four subject systems and usage for variability management, showing its feasibility.

1. Introduction

Software product line (SPL) engineering is the common methodological process for developing together a set of related software-intensive systems. The process is intended to achieve mass customization with methodological reuse of the software systems seen as products, leading to lower cost and time-to-market and improved quality. At the domain level, the variability of considered systems is commonly described in terms of their common and varying features, as their reusable units, in a *feature model* (FM) [47, 48, 22]. Then, in a forward engineering approach, their features are realized in different software assets, including code assets. Consequently, the objective is that various systems can be systematically derived from a common and managed set of software assets.

Although variability is largely studied in the context of software product lines and product families [22], most modern software-intensive systems, being organized as a software product line or not, are variability-intensive [43, 34]. In systems that are not part of a product line, that is, variability-rich systems, the implemented variability in their code assets is neither explicit nor documented. This is because the variability of code assets is typically realized by a diverse and combined set of traditional techniques, such as inheritance, design patterns, overloading, or generic types. However, their used code units, such as classes, or methods in object-oriented programming, do not align well with the domain features or, simply put, neither features nor variation points with variants (their definition is given in Section 2) are by-product of traditional techniques [20]. This hinders the management of the variability of code assets, which seems to be extensive in real variability-rich systems [17, 89].

For example, let us consider the system of JavaGeom [57], an example of a real variability-rich system, which is not organized as a software product line but is variability-intensive, and was used in a previous study [89]. It is an open-source geometry library for Java that is easily understandable. Although not presented and organized as an SPL, it is architected around well-identified features. Compared to similar real variability-rich systems that are of over 100 K LoC and thousands of potential *vp*-s with variants [17, 89], JavaGeom has over 35 K LoC, being a medium-size variability-rich system. In Figure 1 is given an excerpt of its created feature model, which consists of five features. The abstract feature [90] `JavaGeomFM` does not require an implementation, it is used to represent conceptually the JavaGeom variability-rich system. As for the other four concrete features, `StraightCurve2D`, `Line2D`, `Segment2D`, and `Ray2D`, a form of their implementation is given in Listing 1, using Java language. At first sight, the mapping between features in Figure 1 and used code units, classes, in Listing 1 seems to be straightforward, but it is more complicated. First, the feature names and class names are mostly different, for instance, the feature name `Line2D` is different from its implementation class `StraightLine2D`. Besides, the classes extend or implement other classes and interfaces, showing more dependencies at the code level than that features have at the domain level. Then, the implemented variability by method overloading, `distance()` in lines 4-18, is not modeled in the FM. This single example, where only the

t.xheva@gmail.com (Xh. Tërnava); philippe.collet@univ-cotedazur.fr (P. Collet)

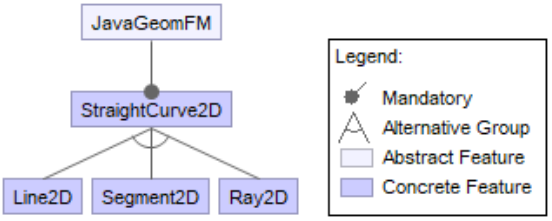


Figure 1: An excerpt of the feature model (FM) of JavaGeom variability-rich system (The FM is created using the FeatureIDE tool [62]).

```java
/* Class level variation point, vp_AbsLine2D */
public abstract class AbstractLine2D extends
    AbstractSmoothCurve2D implements
    SmoothOrientedCurve2D, LinearElement2D {

    /* Method level variation point,
       vp_distance */

    /* First variant of vp_distance */
    public double distance(Point2D p) {
        return distance(p.x(), p.y());
    }

    /* Second variant of vp_distance */
    public double distance(double x, double y)
       {
        Point2D proj = projectedPoint(x, y);
        if (contains(proj))
            return proj.distance(x, y);
        /* Omitted code */
        return dist;
    }
        /* Omitted code */
}
```

```java
/* First variant, v_Line2D, of vp_AbsLine2D */
public class StraightLine2D extends
    AbstractLine2D implements SmoothContour2D,
    Cloneable, CircleLine2D {
        /* Omitted code */
}
```

```java
/* Second variant, v_Segment2D, of vp_AbsLine2D
   */
public class LineSegment2D extends
    AbstractLine2D implements Cloneable,
    CirculinearElement2D {
        /* Omitted code */
}
```

```java
/* Third variant, v_Ray2D, of vp_AbsLine2D */
public class Ray2D extends AbstractLine2D
    implements Cloneable {
        /* Omitted code */
}
```

Listing 1: An excerpt of variability implementation in the JavaGeom variability-rich system considered as a software product line.

techniques of inheritance and overloading are used, shows some difficulties in mapping features at the code assets of a given variability-rich system. In real variability-rich systems, the problem is exacerbated as features are often not implemented by a single class or method. This is due partly to the many design choices that influence the usage of different implementation techniques to structure the variability implementations, and partly to the crosscutting nature of features with code units [51].

To overcome such variability management difficulties, existing approaches use annotations, often preprocessor directives [59], or have proposed to put into separate modules, for example, into aspects, feature modules, or delta modules [6], all lines of code that belong to each specific domain feature [7, 83]. In addition, different tools are proposed, for example, pure::variants [36], code tagging [42], or those that propose to use colors, at the representation layer of code, for distinguishing the lines of code that implement each feature [49]. Analysing which technique, tool, or combination of them [67] fits best to implement or manage some variability is hard to conclude as none of them is yet a preferred one [6], while the SPL application domains are themselves largely diverse. However, except for tool-based approaches, most of the proposed solutions affect the initial decomposition or design of code assets, such as their object-oriented design [7], and in case of migration from a variability-rich system to an SPL, a massive code refactoring is required. As a consequence, when an object-oriented design is refactored, for example, to a feature-oriented design [6], the logical relation among code units is usually lost at the code level. For instance, the alternative relation among the three child features in Figure 1 may lose, which relation is realized by subclasses through the technique of inheritance in Listing 1.

Facing these issues, our work takes the assumption that one can keep unchanged the initial design of code assets of a variability-rich system, that is, without the need to migrate it to an SPL, and manage its variability in terms of *variation points* with *variants*, as initial variability abstractions [45]. To the best of our knowledge, no variability management approach is currently addressing the early steps of capturing and modeling the variability of code assets of a variability-rich system [54, 5, 13, 16, 75, 80], especially not when a subset of variability implementation techniques are used in combination [45, 26].

Towards an approach that meets our assumption and goal on managing variability implementations in code assets of a variability-rich system, we proposed, in a previous work [88], a three step framework to manage such variability when a combined set of traditional techniques is used (Section 4). Such techniques expose a form of what we define as *imperfectly modular variability* (Section 3), as domain features do not perfectly align with the used code units. The framework is devised by analysing up to ten common variability implementation techniques used in variability-rich systems, which are revealed in a recent catalog of 21 traditional and emerging techniques [87] (*cf.* Section 2.2.3). This catalog is at first based on 7 previous frameworks, taxonomies, and catalogs [6, 33, 68, 72, 74, 73, 26]. For the purpose of our framework, we analysed the characteristics properties of these techniques, such as binding time.

1. The framework provides the means to make explicit the variability of code assets by capturing the used variability implementation techniques in terms of variation points (*vp*-s) with variants, as variability abstractions.
2. It then makes possible to use the captured *vp*-s with variants to model the variability of code assets in a fragmented way into so-called *technical variability models* (Section 4.2.2). Towards this, five types of *vp*-s are identified and suggested for use, whereas what is a fragment aims at being flexible. It can be a package, file, or class with variability that worth to be modeled separately.
3. The framework also introduces the possibility to establish n-to-m trace links between the modeled variability of code assets and their respective features in some feature model at the domain level.

Herein, we extend this work in several aspects:

- First, we further detail the concept of *imperfectly modular variability* and determine its induced challenges (Section 3), which motivate the whole presented work.
- We present the three steps of our framework with all necessary details, formal definitions, and related illustrations (Section 4), mainly by using a running variability-rich system introduced in Figure 1 and Listing 1.
- We present the implementation of the framework as a small textual *domain-specific language* (DSL) in Scala, publicly available. Its technical foundations are also given in Section 5.
- We report on a series of experiments on the application of this DSL to four subject systems (Section 6). This evaluation was conducted around four research questions to study the encountered implementation techniques, types of *vp*-s, the multiplicity of trace links, and to evaluate the overhead in modeling with the DSL.
- We also evaluate the capacity of usage of the proposed framework for variability management by adapting a previous work on early consistency checking [86] and showing how the tooled framework enables us to integrate such a checking method in one of the subject systems (Section 7).

Threats to the validity of the framework and its implementation are discussed in Section 8. We also put our contribution into perspective with some related work, including industrial approaches for managing variability at the implementation level (Section 9). Finally, Section 10 concludes this article by summarizing the contributions and discussing future work.

2. Background

Some of the key aspects of the variability management of a system, being it organized as an SPL or not, is the ability to model the realized variability of code assets and then possibly to trace it to its specified variability. In the following, we provide a background on variability modeling, realization, and traceability.

2.1. Variability modeling

Since its first presentation by Kang et al. [47], feature modeling has progressively been widely adopted as a means for scoping and modeling the variability of software systems within an SPL in terms of features [28]. A feature model

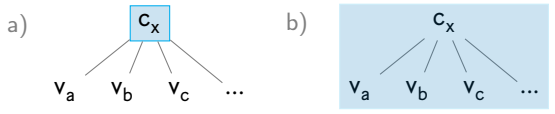


Figure 2: The variation point (*vp*) concept as: **a)** a *variable point* (*i.e.*, the c_x), **b)** a *variable part* (*i.e.*, the c_x with variants). The c_x is the common part for variants v_a, v_b, and v_c.

(FM) is a tree structure of features, consisting of *mandatory, optional, or,* and/or *alternative* logical relations between features with their cross-tree constraints, *implies* and/or *excludes*, that are expressed in propositional logic. In Figure 1 is given an example of an FM, for the JavaGeom variability-rich system. Its main feature, `JavaGeomFM`, represents conceptually the JavaGeom domain. It has a mandatory feature, `StraightCurve2D`, with three alternative features, `Line2D`, `Segment2D`, and `Ray2D`. While an FM can have cross-tree constraints between features, there is none in this example. Semantically, an FM represents all the valid software systems (*i.e.*, the feature configurations) within an SPL or within a variability-rich system.

2.2. Variability realization

We now provide some background on the variability of code assets, the variability modeling, and the used techniques for its realization in variability-rich systems.

2.2.1. Reusable code assets

In real variability-rich systems, despite the programming paradigm (*e.g.*, object-oriented, or functional), the implementation of reusable code assets complies, mostly, to a commonality and variability approach [26]. Specifically, a domain is decomposed into subdomains then, within each subdomain, the commonality is factorized from the variability. Thus, the 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 [91], that is, the assets that are included in any software system. A commonality is a common part of the related variant parts within a subdomain. After the commonality is factorized from the variability and implemented, it becomes part of the core (*i.e.*, it is buried in the core [26]), except when it represents some optional variability. On the other hand, the variant parts are used to distinguish the software systems in the domain. A subdomain can have more than one common and variant part. The core with the commonalities and variabilities of all subdomains constitute the wholeness of code assets in a variability-rich system.

2.2.2. Commonality and variability abstractions

While features are commonly used to model the domain variability of an SPL or variability-rich system (*i.e.*, as problem-oriented abstractions), the commonalities and variabilities in code assets are usually abstracted in terms of variation points (*vp*-s) with variants, respectively (*i.e.*, as solution-oriented abstractions for the realized variability of code assets) [27, 28, 76]. Moreover, unlike features that are mere names, *vp*-s with variants are related to concrete elements in code assets. Originally, "*a variation point identifies one or more locations at which the variation will occur*" [45], whereas the way that a variation point is going to vary is expressed by its variants.

In the literature, there are few different understandings of a *vp* [85, pg. 16]. Mostly, a *vp* is recognized *(i)* as an abstraction of a location or point that varies in reusable code assets (*cf.* Figure 2a) [45], *(ii)* as a variable part (*cf.* Figure 2b) [9], or *(iii)* as an abstraction of a variability implementation technique [22, pg. 48]. But, these all have a common meaning. Specifically, the *variable part* is like an organizing container [9]. It contains the *location* where some variability happens in reusable code assets (*i.e.*, the variation point), the variants, and the used *technique* to implement them. In this work we use the same meaning of *vp*-s with variants.

2.2.3. Variability implementation techniques

The used techniques for implementing variability in an SPL or variability-rich system are known as variability implementation techniques [32, 87]. They came from several programming paradigms and are supported by different constructs in different programming languages, which in turn offer different properties for features or *vp*-s with variants. Examples of such techniques are inheritance, preprocessor directives, feature modules, or some design patterns. For example, in Listing 1 is shown the usage of two variability implementation techniques, inheritance and overloading. They implement two variation points (see the comments in Listing 1), lines 1-20 and lines 4-18, respectively,

which have different properties. Both of them have different granularity, of *class* and *method* levels, and during the product derivation, they will be resolved at different times, *runtime* and *compile time*, respectively.

Because of the large diversity of techniques, an important issue for variability implementation is the ability to evaluate and choose a technique that will fulfill best some given variability requirements. Therefore, for various techniques, several evaluation schemas have emerged, mainly from academia, in the form of frameworks, taxonomies, and catalogs. They all evaluate a different subset of techniques regarding different properties, some of which overlap. For instance, Svahnberg et al. [84] propose to group and evaluate techniques by 2 main properties, according to the size of software entities with variability or variable (components, frameworks, and lines of code), and their latest binding time. Then, the others provide some more concrete and enriched sets of techniques and evaluation properties. For instance, Patzke and Muthig [72] evaluate 8 techniques with up to 6 properties, Patzke and Muthig [73] evaluate 1 technique based on 10 properties, Muthig and Patzke [68] evaluate 8 techniques with up to 8 properties, Gacek and Anastasopoules [33] evaluate 7 techniques with up to 7 properties, Patzke et al. [74] evaluate 7 techniques with up to 14 properties, or Apel et al. [6] evaluate 10 techniques with up to 15 properties.

A more systematic evaluation is provided by a recent catalog [87] that provides an updated set of 21 variability implementation techniques. They are categorized by their 24 different properties that they may support or belong to, such as the logical relation that a technique provides among variants of a *vp*, their granularity, binding time, traceability, language-based, tool-based, or in which programming paradigm it belongs to. As a main property, all 21 techniques are categorized into traditional or emerging techniques:

- **Traditional techniques** are those that 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. Although, using these techniques feature does not have a first-class representation in implementation. Such techniques are inheritance, overloading, or design patterns. For instance, the traditional technique of overloading provides an *alternative* logical relation between variants of a *vp*, but it cannot be used to provide an *or* or *optional* logical relation between them [33, 72, 87].
- **Emerging techniques** have appeared with the SPL engineering advances. Here, the concept of feature is a first-class citizen at the code level. Such techniques are frames [10], feature modules [6], or delta modules [78]. For instance, the emerging technique of feature modules provides good support for realizing the *or* logical relation between feature modules in code assets, but it hardly supports the realization of *alternative* or *optional* logical relation between them [6, 87].

Instead of eliminating this diversity, for example, by proposing to implement the whole variability only at the class level, or by mixing two languages, such as the case of C with `#ifdef`-s, or proposing to use another code unit, such as feature module, that crosscuts with the prior design of code, in this work we strive to model the implemented variability of variability-rich systems by capturing and modeling different properties of *vp*-s with variants that are implemented by traditional techniques. Specifically, considering the scope of our study, we retain the ten object-oriented traditional techniques and software design patterns provided in the catalog, which are commonly used to implement variability-rich systems. The other techniques, such as preprocessors or feature modules; aim at a methodological development of related software systems as an SPL, therefore they are outside the scope of this study. This enables to aim at variability management in variability-rich systems implemented with such traditional techniques. Apart from this, in Section 4.1.2 we detail four characteristic properties of *vp*-s with variants that are essential for managing the variability in code assets, such as logical relation, binding time, evolution, and granularity. It is important to emphasize that, in principle, each *vp* is associated with a single variability implementation technique, whereas the same technique can be used to implement several *vp*-s within a variability-rich system.

2.3. Variability traceability

The CoEST [1] defines a trace (noun 2) as "*a specified triple of elements comprising: a source artifact, a target artifact, and a trace link associating the two artifacts*", and the traceability as "*the potential for the traces to be established and used*" [23].

The concept of traceability is already used for different reasons in *single* software engineering, mostly to trace requirements. But, in SPL and variability-intensive systems engineering the relevant entities that are required to be traced and the semantics of trace links are different. Specifically, an explicit capturing of variability information and

[1] Center of Excellence for Software Traceability: `http://coest.org/`

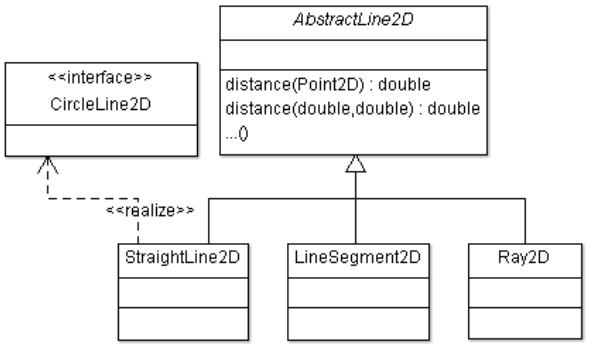


Figure 3: The detailed design of the given excerpt of *JavaGeom* variability-rich system in Listing 1.

StraightCurve2D ⟼ *AbstractLine2D*

Line2D ⟼ *StraightLine2D*

Segment2D ⟼ *LineSegment2D*

Ray2D ⟼ *Ray2D*

Note: *Along this work, the non-italic names stand for features, the italic names stand for code unit, whereas a prefix "vp_" and "v_" is used for variation points and variants. These prefixes are reintroduced in Section 4.1*

Figure 4: The mapping of domain features in Figure 1 to the code units in Figure 3. The ⟼ has the *"implemented by"* meaning.

their modeling of dependencies and relationships separate from other development artifacts is needed [16]. This is well known as variability traceability [4, 5]. It includes the ability to trace the specified variability in problem space to the realized variability in solution space, for example, the ability to trace features in Figure 1, as source artifacts, to their respective *vp*-s with variants in implementation in Listing 1, as target artifacts.

According to the generic traceability model [39], trace links are basically *established* to meet a specific *usage* purpose and need to be *maintained* during the whole engineering process [75]. Moreover, variability trace links can be established for different reasons and by different stakeholders [24, 5]. For example, they can be used for (semi)automating different processes during the application engineering [75], namely for resolving the variability during product derivation, evolving, checking consistency, addressing, or comprehending variability. In this work, we focus on the ability to establish these variability trace links and use them for checking the consistency of variability. For example, after tracing features in Figure 1 to their respective *vp*-s with variants in Listing 1, trace links can be used to check whether each specified feature in the FM is addressed at the code level, or whether the *alternative* relation between three features, `Line2D`, `Segment2D`, and `Ray2D`, is properly considered during their implementation in code.

3. Imperfectly modular variability

Variability management would be simplified in case that each domain feature was implemented in a modular way, by a single language code unit [6], as the mapping of domain features to their implementation would be straightforward. But, this is clearly not the case in real variability-rich systems [89], with the current language constructs and design methods, where variability is implemented by using a combined set of traditional techniques, such as inheritance, overloading, generic types, or design patterns. These techniques do not enable us to completely align the domain features to the places where they are implemented, leading to what we define as *imperfectly modular variability* at the implementation level. On the contrary, a perfect alignment of a domain feature to its implementation would make traces trivial to be established.

To illustrate what imperfectly modular variability means, let us consider the given features in Figure 1 for JavaGeom variability-rich system and the excerpts of their respective implementation as in Listing 1. Focusing on the implementation techniques in Listing 1, the design of which is shown in Figure 3, the abstract class, *`AbstractLine2D`*, is a *vp* and its three subclasses, *`StraightLine2D`*, *`LineSegment2D`*, and *`Ray2D`*, represent its variants that are created by specializing its implementation. In this way, features in the FM in Figure 1 seem to have a direct and modular mapping in implementation, as is shown in Figure 4, where ⟼ has the *implemented by* meaning.

Actually, this perfect form of modularity hardly exists. The variation point *`AbstractLine2D`* and its variants have a lot of internal variabilities. For instance, the method *`distance()`* in *`AbstractLine2D`* is another *vp* with two alternative variants implemented using the technique of overloading (*cf.* Listing 1 in lines 6-9 and 11-18). The variants of `distance()` can be considered as implementations in finer grained modules. They represent some technical and nested variability, which could be specified or not in the FM, but still need to be documented, traced, and managed (*e.g.*, in order to be resolved). Then, the feature `Line2D` uses several *vp*-s, such as *`AbstractLine2D`*, *`CircleLine2D`*,

the variant `StraightLine2D` (*cf.* Figure 1 and Figure 3), plus their finer grained *vp*-s mentioned just before.

Imperfect modularity comes from the fact that a feature is a domain concept and its refinement in code assets is a set of *vp*-s with variants, even if they are modular, meaning that it may not have a direct and single mapping. Also, while it would be preferable to use a single variability implementation technique to implement the variability of a system, none of the existing techniques is yet "*a preferred one*" [6], which could cover all kinds and properties of variability that may appear in different domains. In our illustrative example, the traditional techniques of inheritance and overloading are used together, which induces a *class* and *method* granularity of variability with *runtime* and *compile time* binding, respectively. In these techniques, the conceptual features do not have a first-class representation in implementation. Still, a degree of modularization of variability can be achieved when used with variability management or traceability in mind. Therefore, we propose the following definition.

Definition 1. *An imperfectly modular variability in implementation occurs when some variability is realized in a methodological way with several traditional variability implementation techniques used in combination. The code is not necessarily shaped in terms of domain features, but still, it is designed with the variability traceability in mind.*

Our motivation in this work is the ability to manage this imperfectly modular variability of variability-rich systems. Towards that, we determine the three following challenges.

$Challenge_1$: The diverse *vp*-s with variants of code assets of variability-rich systems, realized by different traditional techniques, need to be captured and modeled in some way, in order to trace them to domain features.

$Challenge_2$: The way variability is modeled at the implementation level and traced to the domain features should be well-suited for variability management, namely for checking their consistency.

$Challenge_3$: The feasibility of an approach addressing $Challenge_1$ and $Challenge_2$ should be demonstrated by a tool support.

The $Challenge_1$ is actually motivated by the $Challenge_2$. The importance of capturing and modeling the different types of *vp*-s and their implementation techniques becomes especially important during the usage of trace links. For instance, when trace links are exploited to resolve variability in a variability-rich system, knowing the binding time of variants in a *vp* is necessary. Then, the logical relation between variants in a *vp* is also required to check the consistency between the variability at the specification and implementation levels. In addition, knowing whether a *vp* is *open* for adding new variants or *unimplemented* is needed during variability evolution. All this indicates that considering the variability implementation techniques during the variability modeling of code assets, as challenged in $Challenge_1$, has a strong impact on the multiple usages of trace links, that is, in variability management ($Challenge_2$).

In this article, we propose a tooled framework to capture, model, and trace the imperfectly modular variability at the implementation level of a variability-rich system, in a forward engineering process, so to be able to exploit the captured trace links, *e.g.*, for consistency checking.

4. A three step framework

To address $Challenge_1$, we propose a three step framework for managing the imperfectly modular variability of code assets of a given variability-rich system. It supports the following tasks, which are also depicted in Figure 5:

1. Capturing the imperfectly modular variability of code assets in terms of *vp*-s with variants, as abstract concepts.
2. Modeling this variability in terms of *vp*-s with variants while keeping the consistency with their implementation in code assets.
3. Establishing the trace links between the specified and implemented variabilities.

The first two steps are mostly based on a conducted study of 21 variability implementation techniques regarding their 24 characteristic properties [87]. This study was briefly discussed in Section 2.2.3. We use this large set of summarized characteristic properties to understand what characterizes the variability in implementation and how it can be captured and modeled in terms of variation points with variants. Notably, we first analysed what characterizes only traditional techniques, being the focus of our study (*cf.* Section 2.2.3), which lead to the form of imperfect modularity of variability. Then, we noticed that 4 from the 21 characteristic properties are crucial for a variability management framework, namely for consistency checking, by which we would address $Challenge_2$. Those four properties are the

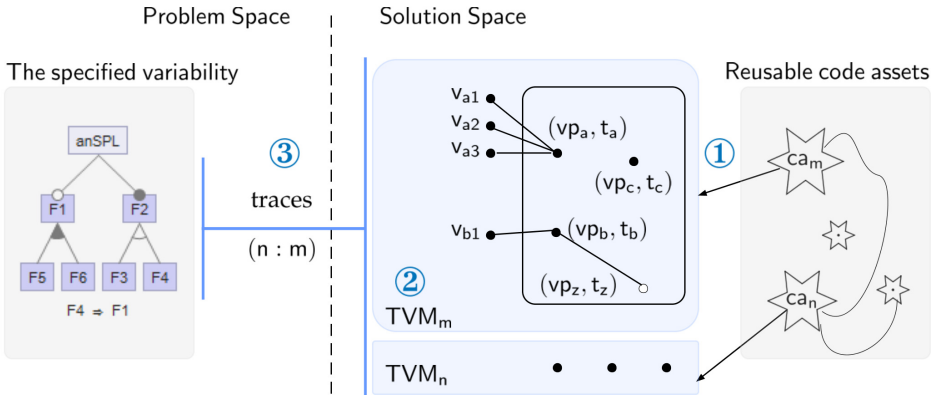


Figure 5: A three step framework for variability management of code assets of a given variability-rich system. TVM_m *stands for technical variability model of the reusable code asset* ca_m*, with vp-s* $\{vp_a, vp_b, ...\}$ *and their respective variants* $\{v_{a1}, v_{a2}, ...\}$ *that are realized by different techniques* $\{t_a, t_b, ...\}$*. Whereas,* $\{f_1, f_2, ...\}$ *are features in the FM.*

logical relation between variants of a *vp*, their *binding time*, *evolution*, and *granularity*. On the other hand, the third step regarding the trace of variabilities is based on some other existing frameworks on variability traceability [4, 5].

The second or third step begins only after the completion of its preceding step. In the following, we give a detailed and formal representation of each of these three steps, which are illustrated by examples.

4.1. Capturing the variability of code assets

As previously discussed, a variable part in code assets that has a form of imperfect modularity usually consists of a common part with its variants (*cf.* Figure 2b). It may happen that a whole code asset, such as a source file, a package, or a class, is a common part or varies (*i.e.*, is a variant). In effect, the variable part consists of an implementation technique that denotes a mechanism for factorizing the commonality among related variants, for creating the variants, a way for resolving the variants, and the variants themselves. Toward addressing the first step of the framework, we naturally abstract them by using the concepts of variation points (*vp*-s) with variants. In general, it is expected that some targeted code assets of a system can have several *vp*-s and their variants, which can have dependencies with the *vp*-s and variants in the same or in other assets. Therefore,

Definition 2. *Capturing the variability of code assets means to abstract the places where the variability happens, namely the common code assets, or their common elements, with their varying code assets or their varying elements, and to represent them by variation points (vp-s) with variants concepts, respectively.*

Capturing a *vp* with its variants is specific to the used technique to implement variability. Therefore, we propose the following formalization. Let vp_x be a specific variation point and v_{xy} one of its variants, where $x, y \in \mathbb{N}$. We assume that the vp_x is implemented by a single traditional technique t_x. The set $\mathbb{T}$ of possible techniques for vp_x with its variants is then made explicit in our framework:

$$\mathbb{T} = \{\text{inheritance, generic type, overriding, overloading, strategy pattern, template pattern, ...}\}$$

It includes any traditional technique that promotes a form of reusability for code assets in a given variability-rich system [2].

A *vp* with variants is not a by-product of such a traditional technique [20, 81]. Therefore, to capture them properly, we need also to tag them in code assets in some way. In this context,

Definition 3. *Tagging a variation point or variant means to map the variation point or variant concept to its concrete asset, or asset element, in code assets.*

[2] Although this set includes any of such techniques from object-oriented, functional programming, or design patterns, we exemplify and validate the framework with a limited set of them. In total, we use up to ten techniques, which are common and are used in our considered variability-rich systems (given in Section 6).

This mapping is needed to keep the association between the captured *vp* or variant concept and its respective code asset.

For example [a], from Listing 1, the superclass *AbstractLine2D* is a *vp* and we abstract/capture it as vp_AbsLine2D. Whereas, tagging has in purpose to ensure their association, namely (*AbstractLine2D*, vp_AbsLine2D). Similarly, its subclasses *StraightLine2D*, *LineSegment2D*, and *Ray2D* are the *vp*'s variants, which we abstract as v_Line2D, v_Segment2D, and v_Ray2D, respectively. Their respective associations are: (*StraightLine2D*, v_Line2D), (*LineSegment2D*, v_Segment2D), and (*Ray2D*, v_Ray2D).

[a] In the following, all illustrative examples will be given into similar blue boxes.

4.1.1. Tagging properties

We provide the following nomenclature for describing the tagging properties of *vp*-s with variants.

Let the set VP of variation points in code assets of a targeted system be:

$$VP = \{vp_a, vp_b, vp_c, ..., vp_n\}, \text{where } a, b, c, n \in \mathbb{N} \tag{1}$$

The set V of all realized variants for the $vp_x \in VP$ is then:

$$V = \{v_{x1}, v_{x2}, v_{x3}, ..., v_{xn}\}, \text{where } x, n \in \mathbb{N} \tag{2}$$

The set CA of all code assets of the system, with variability or being variable themselves, which have an association to some *vp*-s or their variants, is:

$$CA = \{ca_m, ca_n, ca_o, ..., ca_z\}, \text{where } m, n, o, z \in \mathbb{N}. \tag{3}$$

The set of elements in a code asset, for example, for $ca_x \in CA$, which have an association to some *vp*-s or their variants, is:

$$ca_x = \{ea_{x1}, ea_{x2}, ea_{x3}, ..., ea_{xn}\}, \text{where } x, n \in \mathbb{N}. \tag{4}$$

According to Definition 3, the abstractions of each *vp* and variant from sets (1) and (2) are associated with their concrete implementation in code assets, for example, with a varying file, class, or method, which we named as in sets (3) and (4). Analysing some of the traditional variability implementation techniques, we discerned five possible associations for *vp*-s and five for variants. They also represent all possible associations between a *vp* or a variant and their respective code assets. All these associations are described in the following and can be grouped into three kinds, named as *single location*, *implicit*, or *spread*.

Single location. This kind of association represents the case when a *vp* or variant is tagged to a single location in code assets. The four possible cases are:

- (vp_x, ca_x) or (vp_x, ea_{xn}), when the code asset ca_x or the *n* element of it is a common part of some variants. This is the case when vp_x is associated with only one code asset or with only one element of a code asset, respectively.

  For instance, the association (vp_AbsLine2D, *AbstractLine2D*) is between the *vp* and the class, as a code asset or an element of a file, that implements it.

- (v_{xy}, ca_x) or (v_{xy}, ea_{xn}), where y in v_{xy} represents a specific variant of V in (2) and n represents a specific varying code asset of CA in (3) or a varying element of ca_x in (4). This is the case when the variant is associated with only one varying asset or one varying element of it.

  (v_Line2D, *StraightLine2D*), (v_Segment2D, *LineSegment2D*), or (v_Ray2D, *Ray2D*), where each of these variants is implemented by a single class in code assets.

Implicit. This kind of association represents the case when a *vp* or variant is implicit and cannot easily be tagged in code assets. Therefore,

- $(\mathsf{vp}_x, \emptyset)$ or $(\mathsf{v}_{xy}, \emptyset)$, when the vp_x or v_{xy} is implicit, that is, hard to be tagged.

  For example, this is the case when an *if-else* statement is used [84].

We do not include in the framework two cases: *(i)* the case when a *vp* is introduced but has no variants [84], as we consider that it is part of the core assets [91], and *(ii)* the case when a variant is unimplemented, because, for example, an abstract class may have infinite subclasses as variants and abstracting them would be impossible.

Spread. This kind of association represents the case when the same *vp* or variant is tagged to several locations in code assets. Therefore,

- $(\mathsf{vp}_x, \{\mathsf{ca}_m, \mathsf{ca}_n, ..., \mathsf{ca}_z\})$ or $(\mathsf{vp}_x, \{\mathsf{ea}_{x1}, \mathsf{ea}_{x2}, ..., \mathsf{ea}_{xn}\})$, when the vp_x is found in more than one place in code assets. The vp_x can be associated also with any code asset from the set CA in (3) and/or with any of their elements from the set ca_x in (4).
- $(\mathsf{v}_{xy}, \{\mathsf{ca}_m, \mathsf{ca}_n, ..., \mathsf{ca}_z\})$ or $(\mathsf{v}_{xy}, \{\mathsf{ea}_{x1}, \mathsf{ea}_{x2}, ..., \mathsf{ea}_{xn}\})$, when the variant is found in more than one place in code assets. The v_{xy} can be associated also with any code asset from the set CA in (3) and/or with any of their elements from the set ca_x in (4).

  This is the case when a *vp* or variant is implemented by more than one core asset or their elements. For example, the (`vp_distance`, { *`distance(p)`*, *`distance(x, y)`* }) is a *vp* that encircle both overriden methods, therefore it needs to be tagged to both of them.

Moreover, these spread cases have a slightly different meaning from the crosscutting case of features, for example, when preprocessor directives are used. Commonly, preprocessors are used to annotate the lines of code that implement the functionality of a domain feature, which may crosscut the current design of code. Whereas, we consider that *vp*-s and variants align with the design of code.

In some existing approaches *vp*-s or variants are labeled depending on their resolution nature (*e.g.*, choice, substitution [40]), or depending on their location in assets (*e.g.*, kernel-param-vp, kernel-abstract-vp [37]). For now, we keep a more inclusive label, simply as a *vp* and variant concept, which can reflect in the future their resolution nature, location, or evolution.

4.1.2. Capturing characteristic properties

In addition to the tagging properties, depending on the used variability implementation technique, the nature of a code asset element that represents a *vp* or one of its variants varies. Their variety is presented by the characteristic properties of *vp*-s with variants [87]. Therefore, as part of our framework, we introduce here a formal definition of four properties that need to be captured, given that they are essential for variability management, such as for checking, resolving, and evolving the variability. They are the *logical relations* between variants in a *vp*, or between *vp*-s, their *binding time*, *evolution*, and *granularity*.

Logical relation. The logical relations between variants in a vp_x variation point that are commonly faced in practice are similar to the relations between features in an FM (*cf.* Section 2.1). In addition, *vp*-s with variants can have dependencies [21], which are also similar to the dependencies between features in an FM, such as *mutual exclusion*, and *requires*. All these possible relations are shown in Table 1.

A single technique, $\mathsf{t}_x \in \mathbb{T}$, can offer at least one of these logical relations between variants. For example, inheritance can be used for implementing the *alternative* variants, overriding for implementing the *or* variants, or aggregation for implementing the *optional* variant(s) of a *vp*.

Concretely, after the three variants `v_Line2D`, `v_Segment2D`, and `v_Ray2D` in Listing 1 are tagged, we capture their *alternative* logical relation, which is realized by inheritance.

Table 1
Logical relations of variation points and of variants in a variation point.

Logical Relation	Description
Mandatory	The variation point or variant is part of each software product
Optional	The variation point or variant can be part of the software product or not
Multi Coexisting (Or)	One or more than one of the variants in a variation point can be part of the software product
Alternative (Xor)	Only one of the alternative variants in a variation point can be part of the software product
Mutual exclusion	When, during configuration, the selection of variation point or variant requires the exclusion of another variation point or variant, and vice versa
Requires	When the selection of a variation point or variant requires the selection of another variation points or variant

Table 2
Binding times of variation points with variants.
Source: Adapted from [19, 22].

Binding time	Values	Description
Static binding (S)	(S) compilation / link (S) build / assembly (S) programming time (S/D) configuration (S/D) deploy and redeploy	*The variability is resolved early during the development cycle, that is, 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, that is, the decision for a variant in a variation point is made as late as possible/dynamically.*

Therefore, we define the set of possible logical relations ($\mathbb{LG}$) of *vp*-s and of variants in a *vp* as:

$$\mathbb{LG} = \{\text{mandatory, optional, alternative, or, mutual exclusion, requires}\}$$

Binding time. Within the code assets of a system, each *vp* is associated with a binding time, which is the time when the variability is decided or the *vp* is resolved with its variants during a product derivation. Then, different *vp*-s may require being resolved at different development phases. Generally, depending from the chosen variability implementation technique, a *vp* can be resolved early during the development cycle (*e.g.*, statically, when the decision for a variant is made at compile time), or later during the development cycle (*e.g.*, dynamically, at runtime) [18, 3, 41].

In some domains, a *vp* may require more than one binding time (*i.e.*, multiple binding times), for example when a feature requires a static and/or a dynamic binding [77]. In this case, it represents a real challenge for existing techniques, as they offer only a single binding time. To accomplish a flexible binding time, different techniques are usually required to be combined. However, these cases seem to be very specific and rare. On the other hand, there are distinguished *binding units*, which are important for identifying which *vp*-s should be bound together. Specifically, when several *vp*-s participate in implementing some major functionality in a system then they may require to be resolved together as a unit [22, Ch. 4],[56, 55]. Therefore in this work, we consider that a *vp* with its variants is realized by a single variability implementation technique that offers a single binding time and is bound independently.

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, which is differentiated by Svahnberg *et al.* [84]. As for the common kinds of binding times, a taxonomy is available [22, 19]. While some approaches have a broader view on binding times, for instance, advocating for different binding times at runtime [19, 41], we use the seven most common static and dynamic binding times in our model, as shown in Table 2.

Thus, the set of possible static and dynamic binding times ($\mathbb{BT}$) of a vp_x is:

$$\mathbb{BT} = \{\text{compilation, assembly, programming, configuration, deploy, startup, runtime, pure runtime, ...}\}$$

Table 3
Granularity of variation points and of their variants.

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

For example, in Listing 1 we capture that the `vp_AbsLine2D` is bound during the *runtime* to one of its three alternative variants, for instance, to `v_Segment2D`.

Evolution. Depending on whether the specified variability in the FM is meant to be evolved with new features, new *vp*-s may emerge or a specific *vp* may evolve with new variants in the future. Thus, depending on the used technique, the new or existing *vp*-s can be *open* or *closed* to evolution.

Therefore, the evolution set ($\mathbb{EV}$) is:

$$\mathbb{EV} = \{\mathsf{open}, \mathsf{closed}\}$$

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*.

In the JavaGeom system, the abstract class *`vp_AbstractLine2D`* in Listing 1 is a *vp* that we capture as an *open* `vp_AbsLine2D`, meaning that its variants can evolve in the future.

Granularity. A *vp* or variant in code assets can have a specific granularity depending on the size of variability and the used technique [6, p.59][50]. Specifically, a *vp* or variant, as abstractions of variability, may represent *(i)* a coarse-grained element that is going to vary, such as a file, a package, a class, or an interface, *(ii)* a medium-grained element, such as a method, or a field inside a class, or *(iii)* a fine-grained element, such as an expression, a statement, or a block of code. A summary of them is provided in Table 3 [3]. Thus, when traditional techniques are used, the following granularity set ($\mathbb{GR}$) contains the elements that actually show the granularity levels at which a variability implementation technique may be used to implement a *vp* or variant.

$$\mathbb{GR} = \{\mathsf{file}, \mathsf{package}, \mathsf{interface}, \mathsf{class}, \mathsf{method}, \mathsf{field}, \mathsf{statement}, \mathsf{generic\ type}, \mathsf{argument}, ...\}$$

For example, in Listing 1 we should be able to capture the variant `v_Line2D` in *class level*, which is realized by the class *`StraightLine2D`* in lines 21-24; or, the variant `v_distance` in *method level*, which is realized by `distance(...)` method in lines 6-9 or 11-18.

4.2. Modeling the variability of code assets

Given that each captured *vp* with its variants is realized by a single technique, they are then somehow related.

For example, in Listing 1 the three variants, v_Line2D, v_Segment2D, and v_Ray2D, are related by inheritance with the vp_AbsLine2D, which technique offers an *alternative* relationship between variants, a *runtime* binding of variants, and an *open* evolution of the *vp* itself.

[3] A similar table, which also includes the granularity of language constructs coming from the emerging techniques, such as feature modules as coarse-grained elements, is available elsewhere [87].

Therefore, we model their relation by associating each *vp* with its realized variants and the used implementation technique. The associated technique $t_x \in \mathbb{T}$ of the $vp_x \in VP$, which relation we write as (vp_x, t_x), describes three main properties of the vp_x: the logical relation for its variants (lg_x), the binding time (bt_x), and the evolution (ev_x):

$$t_x = \{lg_x, bt_x, ev_x, ...\}, \text{ where } lg_x \in \mathbb{LG}, bt_x \in \mathbb{BT}, \text{ and } ev_x \in \mathbb{EV}$$

The three dots in t_x indicate that the other properties can be added, such as the granularity level. Based on the above example, we can establish

$$\text{inheritance} = \{\text{alternative}, \text{runtime}, \text{open}, ...\}$$

In this way, we aim to address the second step of our framework (*cf.* Figure 5) for modeling the implemented variability while keeping the association among the *vp*-s with variants, their implementation techniques, and the elements of code asset that they abstract.

4.2.1. Types of variation points

For modeling the implemented variability, we observed that the following five types of *vp*-s can be distinguished. They are given in the set $\mathbb{X}$ below.

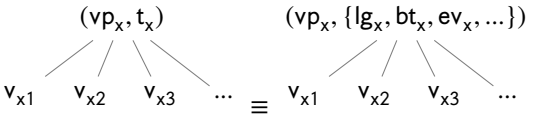
$$\mathbb{X} = \{\text{vp}, \text{vp\_unimplemented}, \text{vp\_technical}, \text{vp\_nested}, \text{vp\_optional}\}$$

This is not an exhaustive list of them, other types of *vp*-s may also be possible, but this set was proved to be sufficient during the framework application (*cf.* Section 6). A formalization for each of them is given in the following.

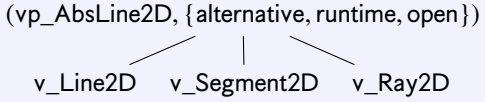
Ordinary. An ordinary variation point (*vp*) is an introduced and implemented *vp* by a specific technique, including all its variants, which are also implemented. Therefore, when the vp_x is an ordinary *vp* we model this variability in code assets as a set of its variants and the characteristic properties derived from the vp_x's implementation technique, t_x. This leads to the following definition (*cf.* Equations (1) and (2)):

$$vp_x = \{V, t_x\} = \{\{v_{x1}, v_{x2}, v_{x3}, ..., v_{xn}\}, t_x\}, \text{where } n \in \mathbb{N} \tag{5}$$

Based on this, we propose the following graphical representation of it [4]:



Such ordinary *vp* with its variants, from Listing 1, is:



Unimplemented. An unimplemented variation point (*vp*) is an introduced *vp* but without predefined variants, meaning that its variants are unknown during the domain engineering phase. When the vp_x is an unimplemented *vp*, we model it as:

$$vp_x = \{\{\varnothing\}, t_x\} \tag{6}$$

respectively,

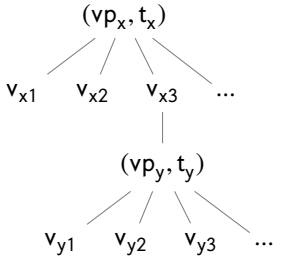
[4] To easily show the main distinction between the five types of *vp*-s, we use this representation (for simplicity, mostly only its left side). Whereas, for formalization reasons, we also provide their respective definition as for the ordinary *vp* in Equation (5).

It means that $\mathsf{vp_x}$ is introduced by the technique $\mathsf{t_x}$, whereas its variants will be implemented later.

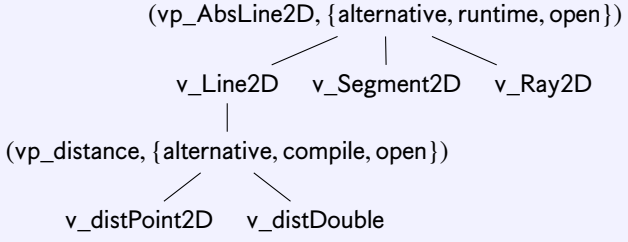
Technical. A technical variation point (*vp*) is introduced and implemented only for supporting internally the implementation of another *vp* which has a direct mapping to a domain feature. The mere difference with the other *vp*-s is that it does not have a direct mapping to a domain feature, thus the name technical. Moreover, a technical *vp* is in principle an ordinary *vp* and needs to be modeled at the implementation level as it needs, for example, to be resolved during product derivation. For the sake of generalization, we consider that a technical *vp* can also be an unimplemented *vp*. When a technical variation point $\mathsf{vp_y}$ is within a variant $\mathsf{v_{x3}}$ of an ordinary $\mathsf{vp_x}$, it is modeled as below.

$$\begin{aligned} \mathsf{vp_x} &= \{\{v_{x1}, v_{x2}, v_{x3}, ..., v_{xn}\}, t_x\} \\ \mathsf{vp_y} &= \{\{v_{y1}, v_{y2}, v_{y3}, ..., v_{yn}\}, t_y\} \\ &\text{where } \mathsf{n} \in \mathbb{N} \text{ and } \mathsf{vp_y} \subset v_{x3} \end{aligned} \tag{7}$$

respectively,



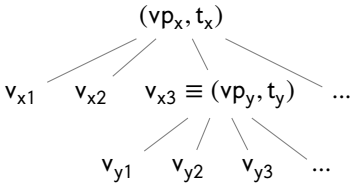
For example, let us consider that `vp_distance` is the abstraction of the method level variability in Listing 1. It can be considered as a technical *vp* of `vp_AbsLine2D`, within the variant `v_Line2D`, as it is not modeled in the FM. It has two alternative variants, `v_distPoint2D` (*cf.* lines 6-9) and `v_distDouble` (*cf.* lines 11-18), which are realized using the technique of overloading. This variability is then modeled as follows.



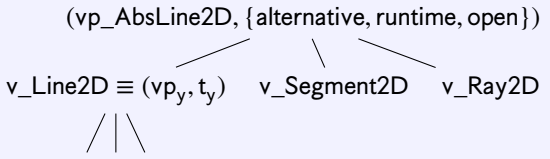
Nested. A nested variation point (*vp*) represents a variable part in a code asset which becomes the common part for some other variants. On the contrary to a technical *vp*, when the $\mathsf{vp_y}$ is a nested variation point of $\mathsf{vp_x}$ then a variable part (*e.g.*, the variant $\mathsf{v_{x3}}$) becomes the common part for the variability of $\mathsf{vp_y}$ (*e.g.*, for $\mathsf{v_{y1}}$, $\mathsf{v_{y2}}$, and $\mathsf{v_{y3}}$). This is modeled as follows.

$$\begin{aligned} \mathsf{vp_x} = \{\{v_{x1}, v_{x2}, v_{x3}, ..., v_{xn}\}, t_x\} = \{\{v_{x1}, v_{x2}, \mathsf{vp_y}, ..., v_{xn}\}, t_x\} = \\ = \{\{v_{x1}, v_{x2}, \{\{v_{y1}, v_{y2}, v_{y3}, ..., v_{yn}\}, t_y\}, ..., v_{xn}\}, t_x\} \\ \text{where } \mathsf{n} \in \mathbb{N} \text{ and } v_{x3} \equiv \mathsf{vp_y} \end{aligned} \tag{8}$$

respectively,



For example, `v_Line2D can be a vp for its (un)implemented variants.`

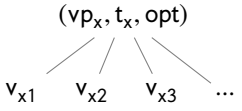


Optional. An optional variation point (*vp*) represents a *vp* in code assets that together with its all variants is optional. This means that, whenever an optional *vp* is included or excluded in a software product, so are its variants. Moreover, variants may have any of the logical relations among themselves.

In our framework, when vp_x is optional then we represent the (vp_x, t_x) tuple as a triple, that is, (vp_x, t_x, opt). We use the acronym opt here in order to distinguish the optionality of the *vp* itself, which is different from the optional logical relation among variants in a *vp*. An optional *vp* can be ordinary, unimplemented, and can have other nested or technical *vp*-s. This is modeled as:

$$(vp_x, \text{opt}) = \{\{v_{x1}, v_{x2}, v_{x3}, ..., v_{xn}\}, t_x\}, \text{where } n \in \mathbb{N} \tag{9}$$

respectively,



To make a *vp* as optional it may not require a technique, that is, simply the *vp* with its variants is not included in the final software product. Therefore, we should document/model it.

4.2.2. Fragmented variability modeling

In addition to the logical relation among variants in a *vp*, different *vp*-s and their variants may also be related, for instance, the nested, technical, and/or ordinary *vp*-s with their variants. This implies that the whole modeled variability at the implementation level will have a forest-like structure, where the *vp*-s and their variants that are related will belong to the same tree, whereas those that are not related will belong to different trees.

In our framework, instead of modeling the whole implemented variability at once and in one place, we propose to model it in a fragmented way. That is, each fragment may include at least one tree with a *vp* and its variants. While there is a problem of reconciliation between evolving code and variabilities with separate models, we choose this solution for three reasons: *(i)* to follow the expected forest-like structure of variability in code assets, *(ii)* to take into account the large amount of implemented variability in real systems, which can be even larger than the domain variability (capturing it in a single model may hamper scalability), and *(iii)* to be able to reason locally over a fragment, for example, to check its consistency. Moreover, instead of a traditional global variability model for a variability-rich system, a modular variability model opens the path toward the development of software ecosystems and product lines of product lines [52]. What we consider as a fragment aims at being flexible. Specifically, it is a code asset with variability, which can be a package, a file, or a class. In other words, a fragment can be any unit that has its inner variability and it is worth being modeled locally or separately.

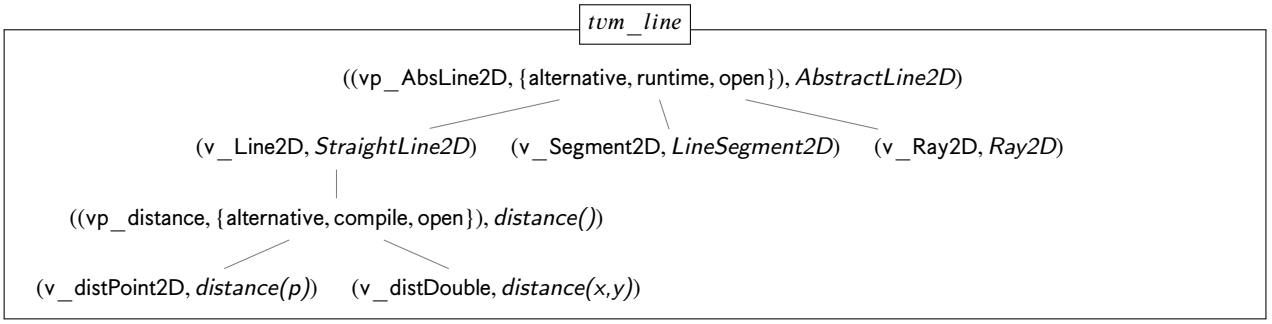


Figure 6: Documentation of the implemented variability in a TVM for the JavaGeom PL (*cf.* Listing 1). The italic names stand for tags of *vp*-s and variants to the variable asset elements.

For this reason, we propose to model the variability of each fragment in a separate model, named as *technical variability models* (TVM). It will contain the abstractions of *vp*-s with variants, tags to their respective code asset elements, and their characteristic properties within some given code assets. Then, a TVM is the power set of all *vp*-s in some given code assets. For example, let us suppose that the *vp*-s in (1) capture the variability of the code asset ca_x in (3) and (4), then the TVM_x is:

$$\mathsf{TVM}_\mathsf{x} = \mathsf{Pow}(\mathsf{VP}_\mathsf{x}) = \{\{\{\mathsf{v}_{\mathsf{a}1}, \mathsf{v}_{\mathsf{a}2}, ..., \mathsf{v}_{\mathsf{an}}\}, \mathsf{t}_\mathsf{a}\}, \{\{\mathsf{v}_{\mathsf{b}1}, \mathsf{v}_{\mathsf{b}2}, ..., \mathsf{v}_{\mathsf{bn}}\}, \mathsf{t}_\mathsf{b}\}, ..., \{\{\mathsf{v}_{\mathsf{z}1}, \mathsf{v}_{\mathsf{z}2}, ..., \mathsf{v}_{\mathsf{zn}}\}, \mathsf{t}_\mathsf{z}\}\} \tag{10}$$

where $\mathsf{a}, \mathsf{b}, \mathsf{n}, \mathsf{x}, \mathsf{z} \in \mathbb{N}$.

As a result, the whole existing variability of code assets of a system, that is, of CA in Equation (3), is modeled in technical variability models (TVMs). It is important to note that, for simplicity, the tags of variants to the variable asset elements in the TVM_x in Equation (10) are not shown, while they are illustrated in Figure 6. This is because, as specified in Section 4.1.1, any of the variants in TVM_x can have one or more tags to the reusable code assets in Equation (3) and/or Equation (4).

In a more illustrative way, in Figure 6 is shown the tvm_line that model the implemented variability in Listing 1. Specifically, it shows the two *vp*-s, the ordinary `vp_AbsLine2D` and the technical `vp_distance`, with their variants and tags to the variable asset elements for the excerpt of JavaGeom system. Similarly, the remained variability in JavaGeom can be modeled within the same or in other TVMs.

A different number of TVMs can be used in a given system, while all of them together model the variability of its all code assets. Conceptually, all TVMs constitute the *main technical variability model* (MTVM), which is specific to the given system. As part of our framework, the MTVM can be defined as the set of all TVMs, given in the following.

$$\mathsf{MTVM} = \{\mathsf{TVM}_\mathsf{m}, \mathsf{TVM}_\mathsf{n}, \mathsf{TVM}_\mathsf{o}, ..., \mathsf{TVM}_\mathsf{z}\} \tag{11}$$

Thus, the MTVM contains the modeled variability of all code assets of a given system. Moreover, unlike the organization of features in an FM as a tree structure, *vp*-s with variants in the MTVM reside in a forest-like structure.

Based on these two steps of the framework, we noticed that the original meaning of a *vp* given in Section 2.2.2 can be further refined:

Definition 4. *A variation point is a location in code assets of a system that represents a node for collecting, attaching to the core implementation, and configuring a set of variable units (variable elements of code assets) that are related in functionality. In particular, it is the place at which the variation occurs [45] and represents the used technique to realize the variability.*

4.3. Tracing the variability of reusable code assets

Variability traceability is usually organized around two types of trace links: *realization* and *use* [5, 75]. The first type of links is used in domain engineering for relating the specified variability at the domain level with the artifacts

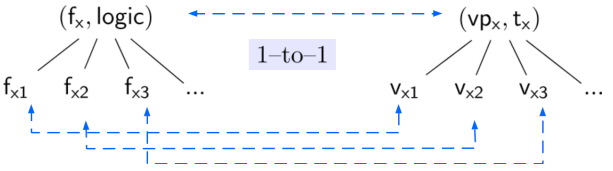


Figure 7: 1-to-1 mapping: ideal mapping.

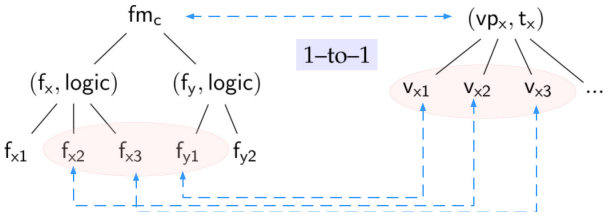


Figure 8: 1-to-1 mapping: a special case of this mapping.

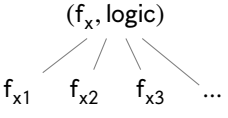
that realize it, such as in code level. Similar usage of this type of link can be found for example between application and platform features [58]. The *use* type of links relates an artifact during the application engineering process to its respective core assets that are developed during the domain engineering process [75]. It is used to show from which core assets an artifact is elicited during product derivation. Usually, the *realization* trace link is implied by "variability traceability" and in the same meaning we use it also here, for the third step of our framework.

4.3.1. Realization trace links

Let us suppose that $f_x \in FM$ is a domain feature of a given system, then

$$FM = \{f_1, f_2, f_3, ..., f_n\}, \text{ where } n \in \mathbb{N} \tag{12}$$

is the set of features in its feature model (FM). In a similar way with the illustration of a *vp* with variants during the framework's first two steps, a compound feature f_x with its varying features and their logical relation (*i.e.*, *logic*) can be given in an illustrative way as follows. The *logic* can have any of the values from Table 1 in Section 4.1.2.



For mapping features of an FM to *vp*-s and variants of TVMs we use a single bidirectional type of trace links `implementedBy` ($\longmapsto$) or `implements` ($\longmapsfrom$), which presents the *variability realization* trace link at the implementation level. Although the nature of trace links is bidirectional, a single bidirectional type is needed to distinguish which is the source artifact and which is the target artifact of the trace link (according to the trace link definition given in Section 2.3). Namely, `implementedBy` and `implements` make possible this differentiation. Otherwise, a link such as $f_x \leftrightarrow vp_x$ would become hard to read as the source and target artifacts are not distinguished.

In general, this mapping of variability from the domain level to the implementation level is expected to be n-to-m (*cf.* step ③ in Figure 5). For example, if a domain feature is implemented by three *vp*-s , then three trace links are needed to describe this mapping. Specifically, from the analysed diversity of *vp*-s with variants [87], traces of variability may consist of (1) 1-to-1 mapping, (2) 1-to-m mapping, and/or (3) n-to-1 mapping.

1-to-1 mapping. This is the ideal mapping where each specified feature is realized by a single code asset (element), which is represented by a *vp* or variant concept. Such mapping looks as in Figure 7.

In this case, we say that a compound feature f_x and its varying features f_{x1}, f_{x2}, and f_{x3} are implemented ideally by a single variation point vp_x with its variants v_{x1}, v_{x2}, and v_{x3}, or conversely. Thus, their mapping is ideal or 1-to-1. Therefore, for feature f_x we write:

$$\begin{aligned} (f_x &\longmapsto vp_x) \\ \texttt{implementedBy}\ (f_x, &vp_x) \\ \texttt{implements}\ (vp_x, &f_x) \end{aligned} \tag{13}$$

For example, from Figures 1 and 3 in *JavaGeom*, (`StraightCurve2D` $\longmapsto$ `vp_AbsLine2D`). That is, the feature `StraightCurve2D` is implemented by the single variation point `vp_AbsLine2D`, thus mapped to it.

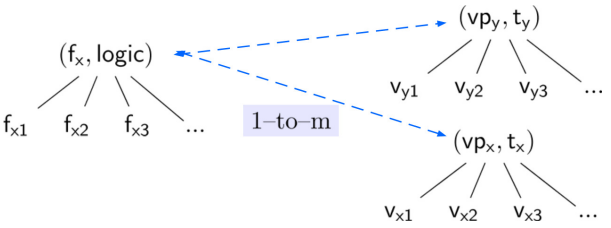


Figure 9: 1-to-m mapping: the mapping of a feature to several ordinary *vp*-s.

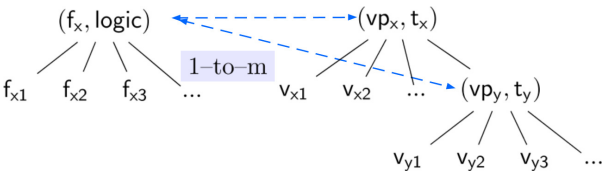


Figure 10: 1-to-m mapping: the mapping of a feature to several nested *vp*-s.

Similarly, based on the three other features illustrated in Figure 7, a varying feature f_{xn} is implemented ideally by a single variant v_{xn} of vp_x, that is,

$$\begin{aligned} &(f_{xn} \longmapsto v_{xn}) \\ \texttt{implementedBy}\ &(f_{xn}, v_{xn}) \\ \texttt{implements}\ &(v_{xn}, f_{xn}) \end{aligned} \tag{14}$$

For example, from Figures 1 and 3 in *JavaGeom*, (`LineSegment2D` $\longmapsto$ `v_Segment2D`). That is, the `LineSegment2D` feature is implemented by the single `v_Segment2D` variant, thus mapped to it.

In general, the vp_x and v_{xn} can be from different TVMs. Therefore, we may need to specify, for example, that $vp_x, v_{xn} \in TVM_x$. Moreover, the *vp*-s with variants in a TVM can be from different code assets.

In the ideal mapping we distinguish a special case when a *vp* with variants implement some features that do not correspond to the same hierarchy in the FM. In an illustrative way, this looks as in Figure 8. According to [22], this happens when the variation point has a broader scope, by relating the functionalities that belong in different subsystems. But, this can be resolved by refactoring the organization of features in the FM or by refactoring the implementation of *vp*-s with variants.

1-to-m mapping. In this mapping, each domain feature is realized by more than one *vp* or variant. In Figures 9 and 10 are illustrated the cases when a feature is mapped to two *vp*-s, being ordinary or nested. In this case, a feature f_x is implemented by several *vp*-s, which can be from the same subdomain of code assets or not, then

$$\begin{aligned} &(f_x \longmapsto \{vp_x, vp_y, vp_z, ...\}) \\ \texttt{implementedBy}\ &(f_x, \{vp_x, vp_y, vp_z, ...\}) \\ \texttt{implements}\ &(\{vp_x, vp_y, vp_z, ...\}, f_x) \end{aligned} \tag{15}$$

In our observations, this case happens depending on the type of *vp*-s (*cf.* Section 4.2.1). Concretely, when vp_x is an ordinary variation point and vp_y its technical variation point, then their mapping looks as in the following:

For example, from Figures 1 and 3, (`StraightCurve2D` $\longmapsto$ {`vp_AbsLine2D`, `vp_distance`}). That is, the `StraightCurve2D` feature is implemented by two variation points, `vp_AbsLine2D` and `vp_distance`, thus mapped to them.

n-to-1 mapping. In this multiple mapping, a vp_x with its variants implement partially several features, which can be from the same or different parts of the tree hierarchy in the FM. They are illustrated in Figure 11 and are traced as [5]

$$\begin{aligned} &(\{f_x, f_y, f_z, ...\} \longmapsto vp_x) \\ \texttt{implementedBy}\ &(\{f_x, f_y, f_z, ...\}, vp_x) \\ \texttt{implements}\ &(vp_x, \{f_x, f_y, f_z, ...\}) \end{aligned} \tag{16}$$

[5] The colored link is used to indicate their partial mapping.

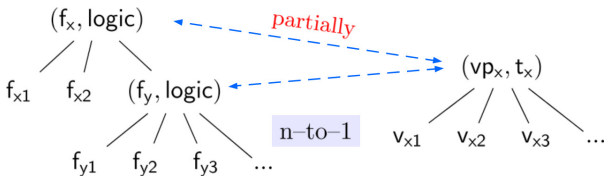


Figure 11: n-to-1 mapping: when several features are partially implemented by a single variation point.

This mapping is relevant to point out the feature interactions or feature crosscutting to *vp*-s and variants in code assets of a variability-rich system, when combinations of features lead to the resolution of a specific *vp*-s.

The overall mapping between features and *vp*-s with variants (see step ③ in Figure 5) is expected to be a partial mapping. This is because, some features in FM can be abstract features, by not requiring an implementation [90], or they can be deferred to be implemented later, for example, when a variation point is unimplemented.

4.3.2. Establishing trace links

An automated approach for establishing the trace links may be desirable. It depends on the multiplicity of mapping between features and *vp*-s with variants. Specifically, when the mapping is ideal then the trace links can be automated by using the same names for features and their respective *vp*-s or variants. For instance, in the JavaGeom system we mostly used the same names for *vp*-s, variants, and features, except that we added the prefix *v_* for a variant and similarly the prefix *vp_* for a variation point, making features like `Ray2D` and variants like `v_Ray2D`. Whereas, in the case of multiple mapping (*e.g.*, 1-to-m), the *vp*-s or variants that implement the same feature can be grouped under the same name with the feature's name that they implement. But, this requires extra work during the variability modeling in TVMs.

In our three step traceability approach (*cf.* Figure 5), establishing the trace links (for step ③) requires both variability models at the specification and implementation levels (derived in steps ① and ②), that is, the FM and TVMs, respectively. Then the established trace links can be kept separate and orthogonal to the FM and the TVMs. As a result, we consider that the proposed framework addresses the first challenge (*cf.* $Challenge_1$ in Section 3) for modeling and tracing the imperfectly modular variability of a targeted system's code assets.

5. Framework implementation

To address $Challenge_3$ (*cf.* Section 3), we developed a prototyped tool to support the three steps of our framework. Different mapping studies [29, 30, 14] and surveys on industrial practices [17, 42] show that, from the variability modeling tooled approaches, textual variability modeling languages have an increased attention. The main stated reasons are against the scalability issues and the difficulties in comprehending a large amount of variability when competing graphical tooled approaches are used. We thus propose the following textual variability modeling language in our context of code assets.

5.1. Variability modeling domain-specific language (VM-DSL)

We implemented the proposed language as a textual *domain-specific language* (DSL). A DSL is a small language that is narrowly focused on a particular problem domain [93, 94, 31]. Technically, we chose the Scala language to implement the framework as it is well-known for its first-class support for designing expressive DSLs [70, 35, Ch. 6]. In addition, the interoperability between Java and Scala enables one to use our DSL in almost all Java-based variability-rich systems, such as JavaGeom.

In Listings 2 and 3 is provided the syntax of our *variability modeling domain-specific language* (VM-DSL). Its design and syntax is guided by the following three main demands, which aim to support the three framework steps.

Demand 1. *One should be able to abstract the implemented variability of code assets of a given system in terms of vp-s with variants while keeping a tag to their respective code assets.*

To meet this demand, the first part of syntax in Listing 2 reveals the VM-DSL support for abstracting and tagging, that is, capturing, *vp*-s with their variants. Specifically, the ⟨*vp*⟩ indicates that each *vp* in code assets is captured by giving a name, a type, and a tag to code assets through setting the values for ⟨*vp-name*⟩, ⟨*vp-type*⟩, and ⟨*tag*⟩,

```
⟨vps-list⟩     ::= ⟨vp⟩ | ⟨vp⟩ ⟨EOL⟩ ⟨vps-list⟩

⟨variants-list⟩ ::= ⟨variant⟩ | ⟨variant⟩ ⟨EOL⟩ ⟨variants-list⟩

⟨vp⟩           ::= ⟨vp-name⟩ '=' ⟨vp-type⟩ '(' 'asset' '(' ⟨tag⟩ ')' ')'

⟨variant⟩      ::= ⟨variant-name⟩ '=' 'Variant' '(' 'asset' '(' ⟨tag⟩ ')' ')'

⟨vp-name⟩       | ⟨variant-name⟩ := string name

⟨vp-type⟩      ::= 'VP' | 'oVP' | 'tVP' | 'nVP'

⟨tag⟩          ::= ⟨class-name⟩ | ⟨method-name⟩ | ⟨field-name⟩

⟨fragment⟩     ::= 'fragment' '(' ⟨tvm-name⟩ ')' '{'
                   ⟨vp-name⟩ 'is' ⟨logical-relation⟩ 'with_variants' '('⟨{⟩⟨variants⟩⟨}⟩')'
                   'use' ⟨technique⟩ 'with_binding' ⟨binding-time⟩ 'and_evolution' ⟨evolution⟩ ⟨EOL⟩
                   '}'

⟨tvm-name⟩     ::= ⟨class-name⟩ | ⟨file-name⟩ | ⟨package-name⟩ | string name

⟨logical-relation⟩ ::= 'MND' | 'OPT' | 'ALT' | 'MUL'

⟨variants⟩     ::= ⟨variant-name⟩ | ⟨variant-name⟩ ',' ⟨variants⟩

⟨technique⟩    ::= 'Inheritance' | 'Overloading' | 'Strategy_Pattern' |'Template_Pattern'

⟨binding-time⟩ ::= 'Compile' | 'StartUp' | 'Runtime'

⟨evolution⟩    ::= 'Open' | 'Close'
```

Listing 2: Syntax for defining variation points and variants, and modeling the implemented variability in terms of them.

```
⟨trace-links⟩ ::= 'traces' '{' ⟨links⟩ '}'

⟨links⟩ ::= ⟨vp-name⟩ 'implements' ⟨feature-name⟩ | ⟨variant-name⟩ 'implements' ⟨feature-name⟩

⟨vp-name⟩ | ⟨variant-name⟩ | ⟨feature-name⟩ := string name
```

Listing 3: Syntax for defining trace links.

respectively. While the ⟨*vp-name*⟩ can have any string values, the ⟨*vp-type*⟩ can have one of the four predefined types of a *vp* given in our framework (*cf.* Section 4.2.1), namely `VP`, `oVP`, `tVP`, or `nVP`. It should be noted that an unimplemented *vp* is defined as an ordinary *vp* (*i.e.*, as `VP`) but without variants. Whereas, the ⟨*tag*⟩ is set to the path of the respective code assets, which can be a ⟨*class-name*⟩, a ⟨*method-name*⟩, or a ⟨*field-name*⟩. Similarly, the ⟨*variant*⟩ indicates that each variant in code assets is captured by giving a name, using a language construct, and giving a tag to code assets through setting the value for ⟨*variant-name*⟩, using `Variant` construct, and setting the value for ⟨*tag*⟩, respectively.

Technically, all captured *vp*-s and variants are kept into their list structures, ⟨*vps-list*⟩ and ⟨*variants-list*⟩, respectively. To set the ⟨*tag*⟩ value our VM-DSL relies on the reflection capabilities in Scala. This helps to keep the relationship and strong consistency between the defined concepts of *vp*-s and variants, that is, between a ⟨*vp-name*⟩ or a ⟨*variant-name*⟩, as well as their concrete implementation in code assets.

Demand 2. *One should be able to model the variability of a given system by capturing four characteristic properties of each vp with variants, namely their used implementation technique, logical relation, binding time, and evolution. Then, depending on the needs, a dedicated variability model can be set up for different parts of the system.*

To meet this demand, in the second part of Listing 2 is given the language syntax for modeling the variability within the `fragment` construct where the four characteristic properties of a *vp* and its variants are specified, namely the ⟨*technique*⟩, ⟨*logical-relation*⟩, ⟨*binding-time*⟩, and ⟨*evolution*⟩. The `fragment` construct can contain more than one *vp* with its variants and requires a ⟨*tvm-name*⟩. The values for the ⟨*technique*⟩, ⟨*logical-relation*⟩, ⟨*binding-time*⟩, and ⟨*evolution*⟩ are set to one of their 5, 4, 3, and 2 predefined values, respectively. Their predefined values are enumerated in Listing 2. Still, they can be extended in the future with other values, for example, to cover a larger set of techniques and binding times. All together, the syntax in Listing 2 is used to set up a single variability model, named as a technical variability model (TVM) in our framework. The filename of each variability model and the ⟨*tvm-name*⟩ of the `fragment` construct are designed to distinguish different variability models within a given system.

Technically, we defined the ⟨*technique*⟩, ⟨*logical-relation*⟩, ⟨*binding-time*⟩, and ⟨*evolution*⟩ values as extensions of their respective sealed classes [6] in Scala, thus well defining their possible values. In addition, the VM-DSL is made expressive by using implicit conversions in Scala. This Scala feature allows the DSL to convert one object to another and gives the user the impression that methods are dynamically added to an existing class. We used it especially to implement the `fragment` construct, in order to express in a more natural language the modeling of variability, as in Listing 2. As a result, the language supports fragmented modeling, meaning that several variability models (*a.k.a.* technical variability models -TVMs) can be defined for a given system.

Demand 3. *One should be able to trace the modeled variability of a given system's code assets to its domain features.*

In addition to language constructs for variability modeling, the VM-DSL provides support for establishing the trace links, as defined in the third step of the framework. The language syntax is given in Listing 3. It provides a `traces` construct with an `implements` construct for establishing the trace links between the specified features and the modeled variability in TVMs. It supports the creation of 1-to-m trace links, where a single feature, that is, ⟨*feature-name*⟩, can be mapped to several *vp*-s or variants, that is, ⟨*vp-name*⟩ or ⟨*variant-name*⟩. This type of trace link also includes the support for the 1-to-1 mapping. In our current prototyped implementation, the n-to-1 mapping is not supported, but this limitation has a low impact for two reasons: *(i)* variability management approaches mainly consider that *vp*-s with variants at the implementation level are a refinement of features from the domain level [44] and *(ii)* the amount of variability at the implementation level is expected to be larger than the variability at the domain level [75, Ch. 4]. This means that a feature, in general, is expected to have a mapping to several *vp*-s and/or variants, thus VM-DSL's support for 1-to-m mapping, while the inverse case is quite rare.

Technically, these traces are internally kept in a map structure as `Map[feature;vp]` and `Map[feature;variant]`, where the `feature` is the key whereas the `vp` or `variant` is its value. Moreover, a key in the map structure can have several values.

The provided syntax of our variability modeling DSL (VM-DSL) in Listings 2 and 3 can be used to implement the framework as another internal DSL in other general-purpose languages or as an external DSL. In addition to the variability modeling of code assets, VM-DSL supports the conversion of TVMs and trace links into propositional logic. This enables the implementation of many scenarios of variability management, such as consistency checking between variability models at the specification and implementation levels [13, 63, 86]. We will detail this usage of our framework in Section 7.

The prototype implementation of our DSL is publicly available at https://github.com/ternava/variability-cchecking/. Its description and usage guidance are also given at https://ternava.github.io/vm-dsl/, including detailed applications to four variability-rich systems.

5.2. VM-DSL usage example

Following the given syntax in Listings 2 and 3, Listings 4 and 5, respectively, show a usage example of the implemented VM-DSL. Specifically, Listing 4 shows the modeled variability of code assets of the JavaGeom system given in Listing 1. Then, Listing 5 shows how the trace links are established between the modeled variability of code assets in Listing 4 and domain features in Figure 1.

From Listing 4, the VM-DSL has to be imported at first into the current scope, such as package, file, or class, where variability needs to be separately documented into a TVM (line 4). In line 5, the usage of Scala reflection is made available. Then, an *ordinary vp* (line 7) with its three variants (lines 8-10) are defined/captured. They are used to model the variability within the `fragment` construct (lines 12-16). As mentioned in Demand 1, the relationship of

[6] A sealed class cannot have any new subclasses added except the ones that are defined in the same file [70]

```
1  /* tvm_line.java */
2  package tvm  // all TVMs are kept within a package
3
4  import dsl._
5  import scala.reflect.runtime.universe._
6
7  val vp_AbsLine2D   = VP(asset(typeOf[AbstractLine2D].typeSymbol))
8  val v_Line2D       = Variant(asset(typeOf[StraightLine2D].typeSymbol))
9  val v_Segment2D    = Variant(asset(typeOf[LineSegment2D].typeSymbol))
10 val v_Ray2D        = Variant(asset(typeOf[Ray2D].typeSymbol))
11
12 import fragment._
13 fragment("geom2D.line") {
14         vp_AbsLine2D is ALT with_variants (v_Line2D, v_Segment2D, v_Ray2D) use
15         INHERITANCE with_binding RUN_TIME and_evolution OPEN
16 }
```

Listing 4: The captured and modeled variability for Listing 1.

```
1  /* tracelinks.java */
2  import dsl.traces._
3
4  traces {
5    vp_AbsLine2D implements StraightCurve2D
6    v_Line2D     implements Line2D
7    v_Segment2D  implements Segment2D
8    v_Ray2D      implements Ray2D
9  }
```

Listing 5: Traced variability between Listing 1 and Figure 1 through the captured *vp*-s with variants in Listing 4.

captured *vp*-s and variants to their respective code assets is kept by using Scala reflection mechanisms. This makes possible to only capture *vp*-s and variants that have a concrete realization in code assets, so that inconsistent *vp*-s and variants without a right tagging to code assets are disclosed at compile time.

From Listing 5, each captured *vp* and variant in Listing 4 is traced to their respective features in Figure 1 that they implement, by using the `traces` construct. In Listings 4 and 5, the comments in lines 1 and 2 indicate that each TVM can be kept into a separate file, including the trace links, and in a dedicated package within a given system.

By providing the VM-DSL prototype as an implementation of the proposed framework, we consider that the $Challenge_3$ is partially addressed. To address the rest of it, we report in the following on the experimental evaluation of the proposed VM-DSL.

6. Applications

The provided framework is expected to be applicable to realistic variability-rich systems where traditional techniques are used. Therefore, to show its applicability, we decided to apply its tool support, the VM-DSL, in several relevant variability-rich systems, as subject systems, to model and trace their implemented variability.

6.1. Research questions

We organized the evaluation by first defining four main research questions, which purpose is to further address $Challenge_3$ on the applicability of our tooled framework (*cf.* Section 3). The four research questions are given below.

RQ_1: **What variability implementation techniques are encountered within a variability-rich system that uses traditional techniques?** This question is intended to answer the encountered combination of used variability implementation techniques in variability-rich systems. To this end, we analyse how often each technique is used and in what combination in four subject systems.

Table 4
The implementation details of four used subject systems.

Subject system	Language	LoC	Files	Features	Implemented Features
Graph [60]	Scala	323	3	19	16
Arcade Game Maker [2]	Scala	651	17	27	26
Microwave Oven [38]	Scala	914	19	26	23
JavaGeom [57]	Java	35,456	142	941	941

RQ_2: **What are the encountered types of *vp*-s within a variability-rich system?** The purpose of this question is to highlight the types (*ordinary*, *technical*, *optional*, *nested* [7]) of *vp*-s that are often found within a variability-rich system, the diversity of the characteristic properties of these *vp*-s, and the number of TVMs needed to model the variability of a system. To this end, we modeled and analysed the variability of code assets of four subject systems.

RQ_3: **What is the encountered multiplicity of variability trace links within a variability-rich system?** The purpose is to show what is the encountered multiplicity of trace links when traditional variability implementation techniques are used. Therefore, we analyse the encountered multiplicity of trace links between domain features and *vp*-s with variants at the code assets in one of the four subject systems.

RQ_4: **What is the overhead for modeling the variability by using the VM-DSL?** The purpose is to evaluate the design of the VM-DSL by investigating how many additional *lines of code* (LoC) are needed to model the variability within a variability-rich system and what is their report with the overall LoC of code assets. To this end, we define a measure that helps to evaluate this overhead, then we calculate it for four subject systems and make observations.

6.2. Analysed subject systems

We expected that the application of our tooled framework in different variability-rich systems of different domains leads to varying collected data, especially for questions RQ_1 to RQ_3. As capturing variability implementations is a time consuming process, we applied VM-DSL on several subject systems, but we could not build very large experiments. For instance, we could not choose big systems using preprocessor directives, such as the Linux kernel, as they did not use object-oriented traditional techniques as defined in the scope of our work, and using our DSL on top of preprocessors to model their variability sounded inappropriate. Therefore, we considered several different object-oriented pre-existing systems, such as Java-based examples in SPL2go [8] and mostly those used in the SPL engineering community. From them, we first selected three small subject systems (with less than 1 K LoC, *cf.* Table 4) that were easily available, have an extensive usage in the SPL engineering community with simple and well-defined domains, so that we can rely on well-established feature models to relate the variability implementations with. We complement these subjects with a fourth subject system being a real feature-rich API (with over 35 K LoC, *cf.* Table 4), which broadens the scope of our validation. The list of four subject systems, with their LoC, used programming language, and the number of domain features, is summarized in Table 4. Specifically,

- **Graph** is a family of graph applications, which is proposed as a case study for evaluating the product line methodologies [60].
- **Arcade Game Maker** is a family of three arcade games: brickles, pong, and bowling. It is introduced by SEI [9] as a case study for experimenting and learning the development of software product lines [2].
- **Microwave Oven** is a family of microwaves, which is proposed to show the usage of UML for developing a set of software-intensive systems from requirements to detailed design [38].
- The fourth subject system, **JavaGeom**, introduced in Section 1 is a real open-source feature-rich API for creating geometric shapes in Java [57].

Before modeling the implemented variability of these subject systems, through applying the VM-DSL, we had to enhance each of them to get complete ones. For the first three subject systems we had their feature models at hand

[7] As defined in Section 4.2.1
[8] SPL2go page with case studies: `http://spl2go.cs.ovgu.de/`.
[9] `https://www.sei.cmu.edu/`.

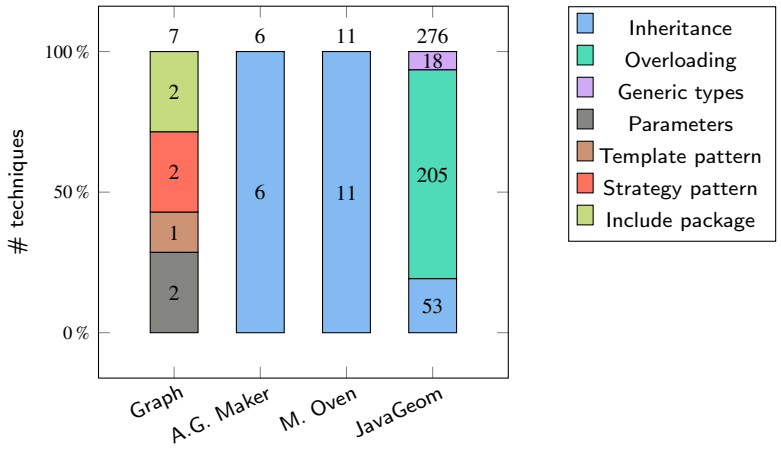


Figure 12: The used techniques, their number, and their combination, in each subject system.

and we just needed their implementations. While different implementations of these systems exist, the available ones were organized as full SPL, e.g., the Graph system is available in SPL2go, but is built using FeatureIDE. As in our scope are variability-rich systems that are not organized as an SPL, we implemented these subject systems. We use the Scala language as it supports the targeted variability implementation techniques and it was easier to apply the VM-DSL within the same development environment.

Obviously, implementing ourselves the variability-rich systems that are going to use the VM-DSL is creating a bias on our experiments, especially on RQ_1 and RQ_2. To reduce it, we tried as much as possible to apply good design practices in the implementation, not especially looking for adding several techniques, but rather picking up the right technique to get a proper design of the systems, as developers of a variability-rich system would have done. In particular, whenever it was possible, we used those techniques that were specified in the design of each given subject system. For instance, the Arcade Game Maker [2] has a detailed design that makes explicit the suggested techniques to implement each *vp* with variants. Similarly, a detailed design is available also for two other subjects, Graph [60] and Microwave Oven [38]. As for JavaGeom, we had to analyse its domain and build its feature model, as the Java source code was available but no feature model was existing. JavaGeom is made of 142 Java source files (excluding tests) and 35 456 lines of code. Considering that the implementation of its 3D geometric shapes is under implementation, we only took its 122 files of 2D geometric shapes that constitute 92% of the code. We then analysed manually the textual description of features in its documentation and their implementation in code. We extracted 941 concrete features (*cf.* Table 4), which are rather close to the implementation. A summary of the size and implementation of these four systems, based on their lines of code, files, and number of features, is given in Table 4. The domains of these subject systems can further be understood by their feature models, which are made available online [10].

After we prepared the feature model and code assets for each subject system, we used the VM-DSL to model the variability of their code assets. For the first three subject systems, we modeled their variability during their domain implementation or just after their whole implementation. For the JavaGeom subject, as we analysed its variability in advance and recorded in a spreadsheet all its *vp*-s with variants, we used the VM-DSL to model only some of its variability implementations [11]. Finally, we analysed the modeled variability of all subject systems regarding the four given research questions in Section 6.1. We discuss these results in the following subsections.

6.3. Encountered implementation techniques

In order to answer the first research question (RQ_1), we analysed the resulted variability models from the VM-DSL usage in each subject system and counted the occurrences of the variability implementation techniques. The results are shown in Figure 12. Specifically, in the Graph subject four techniques are used, namely strategy pattern, template pattern, parameters, and include package. Although the *include package* may not be considered as an implementation technique, we counted it here to show one way how an *optional vp*, included in our framework, can be realized. In

[10] https://github.com/ternava/Variability-CChecking/wiki

[11] Details are available here: https://github.com/ternava/Variability-CChecking/tree/master/src/SPL_Analysed_Examples

Table 5
Number of technical variability models (TVMs) and the captured *vp*-s with variants in each subject system.

Subject system	TVMs	*vp*-s	Logical relation	Binding time	Evolution	Granularity	Variants
Graph	3	7	1 mandatory 6 optional	7 runtime	7 open 1 method	6 class	12
Arcade Game Maker	3	6	6 mandatory	6 runtime	6 open	6 class	17
Microwave Oven	8	11	11 mandatory	11 runtime	11 open	11 class	20
JavaGeom	11	276	276 mandatory	53 runtime 223 compile time	53 open 223 close	53 class 205 method 18 type	665

such a case, an implemented *vp* within a package will be included in the final product only if the package that contains it is included too. In the Arcade Game Maker and Microwave Oven subjects, only inheritance is used as a single implementation technique. As mentioned in Section 6.2, we only used inheritance as it is specified in their detailed design as the most appropriate technique to implement *vp*-s with variants in their domains. Presumably, this can be explained with their domain of variability and a few number of features. In JavaGeom subject up to three techniques are used, namely inheritance, overloading, and generic types. In our analysis we omit the technique of overriding as it is used constantly to realize the specialization in the inheritance hierarchy.

The summarized results are given in Figure 12. They show that in maximum of up to four techniques are used in combination. Then, within a subject system a specific technique is used at least once, such as inheritance in three case studies, Arcade Game Maker, Microwave Oven, and JavaGeom. In Arcade Game Maker it is used 6 times or 100%, in Microwave Oven it is used 11 times or 100%, whereas in JavaGeom it is used 53 times or 19%. In total, seven techniques are used in the four subject systems. As a partial answer to the first research question RQ_1, this shows that a combined set of different techniques is easily used to implement the variability of a variability-rich system. These data especially enforce our overall motivation that, in many variability-rich systems where traditional techniques are used, it is common that several of them are used in combination. In addition to the reported experiment, a combined usage of traditional techniques is also observed by a recent tooled approach, *symfinder* [89]. It provides an automated approach for identifying *vp*-s with variants at class and method level for Java-based variability-rich systems. Its evaluation on eight very large systems, such as AWT library in the JDK 8, Apache Maven 3, and JFreeChart, shows that in each system a combined set of techniques are used. Therefore, the necessity to capture and model the diverse *vp*-s with variants of such code assets shows a strong need for the support we provide in our framework.

6.4. Modeled variability

In each subject system we also analysed the modeled variability in each file within a package. Whenever we found that a variability implementation technique is used at class or method level we abstracted it in terms of *vp*-s with its variants, including their logical relation, binding time, and evolution properties. The resulting data are given in Table 5.

6.4.1. Encountered characteristic properties of variation points

In Table 5 are shown the resulting number of *vp*-s and variants in each considered subject system. For instance, the Graph subject has 7 *vp*-s with 12 variants, the smallest number, whereas the JavaGeom subject has 276 *vp*-s with 665 variants, being the largest number. These data show to some extent the ability of our tooled framework to model a considerable number of *vp*-s with variants at class and method level for a variability-rich system.

This table also gives details of the characteristic properties for the modeled *vp*-s within each subject system. From the resulting 300 *vp*-s in four subject systems (*cf.* Table 5), 294 *vp*-s are modeled as mandatory, that is, 1 *vp* in Graph, 6 *vp*-s in Arcade Game Maker, 11 *vp*-s in Microwave Oven, and 276 *vp*-s in JavaGeom. Whereas, the remained 6 *vp*-s are optional, all being found in the Graph subject. Each of them is realized into separate files and can be incrementally added or not to the core of its code assets. This is not the case for the *vp*-s in Microwave Oven and Arcade Game Maker, as the implementation of *vp*-s crosscuts files. Regarding the JavaGeom, it may have optional *vp*-s too, but we classified all of them as mandatory *vp*-s as knowing whether a *vp*-s is optional required deeper domain knowledge. Then, the 294 mandatory *vp*-s (*cf.* values in the column *logical relation* in Table 5) are modeled as follows:

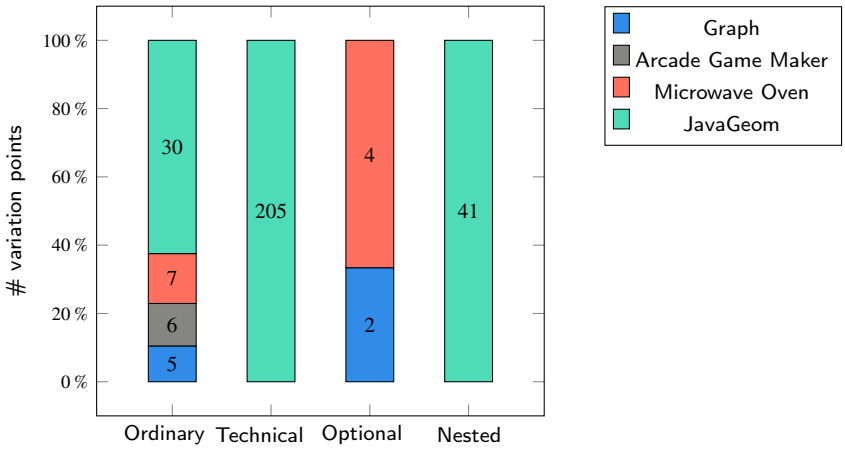


Figure 13: The used techniques, their number, and their combination, in four subject systems.

- 71 *vp*-s have *alternative* variants, with *runtime* binding, *open* evolution, and *class* granularity;
- 18 *vp*-s have *alternative* variants, with *compile time* binding, *closed* evolution, and *type* granularity; and
- 205 *vp*-s have *alternative* variants, with *compile time* binding, *closed* evolution, and *method* granularity.

We observed that in JavaGeom 41% of the medium-grained variabilities happen at the *constructor level* of a class, the rest is at the *method level*. Then, in all subject systems, *vp*-s have between 1 and 9 variants. A close observation showed that most of them have only 2 variants, for example, around 54% of *vp*-s in JavaGeom have only 2 variants.

These data show that *vp*-s of a given variability-rich system, or among them, have indeed diverse characteristic properties. Specifically, even with a small set of subject systems, we observed that each *vp* has its own logical relation between variants, their binding time, ability to be evolved in the future, and a granularity. Moreover, we also observed that these properties look in direct proportion with the used techniques, that is, the larger the set of used techniques the larger is expected to be the diversity of *vp*-s. This answers partially the second research question (RQ_2).

6.4.2. Encountered types of variation points

Considering that each technique is abstracted by a *vp*, the number of used techniques denotes the total number of *vp*-s in a subject system. Thus, from Figure 12, the modeled variability in Graph subject has 7 *vp*-s, in Arcade Game Maker has 6 *vp*-s, in Microwave Oven has 11 *vp*-s, and in JavaGeom has 276 *vp*-s. Then, each *vp* is modeled using one of the five types of *vp*-s, *ordinary*, *unimplemented*, *technical*, *nested*, or *optional*. Their used types are summarized in Figure 13, revealing several findings.

First, except in Arcade Game Maker, more than one type of *vp*-s exist in each subject system. For instance, in JavaGeom three types of *vp*-s are used to model its variabilities, namely *ordinary*, *technical*, and *nested vp*-s. In total, we encountered four types of *vp*-s, *ordinary*, *technical*, *optional*, and *nested* (*cf.* Figure 13). The *unimplemented vp*-s may exist but they were implicit. For example, it was hard to tell which class is introduced in code assets in order to be extended with new variants in the future. These results show that in total there are far more *technical* and *nested vp*-s (82%) than *ordinary* and *optional vp*-s (18%). This is best highlighted from the largest subject system of JavaGeom, where there are around 89% more *technical* and *nested vp*-s than *ordinary vp*-s. Furthermore, this can be important for the resolution of variability and indicates that there is a multiple mapping between domain features and *vp*-s in the implementation.

These observations also partially answer the second research question (RQ_2), showing that different types of *vp*-s are needed to model the variability of code assets in our subject systems. The presence of *technical* and *nested vp*-s indicate that *vp*-s are the refinement of domain features, a characteristic that matters for establishing the variability trace links.

6.4.3. Number of TVMs

In general, we set up more than one TVM to model the entire variability of each subject system. Table 5 shows the resulted number of TVMs with their *vp*-s and variants. For example, we modeled the variability of the Graph subject

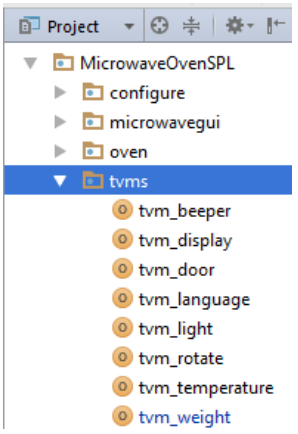


Figure 14: The eight TVMs into separate files for Microwave Oven.

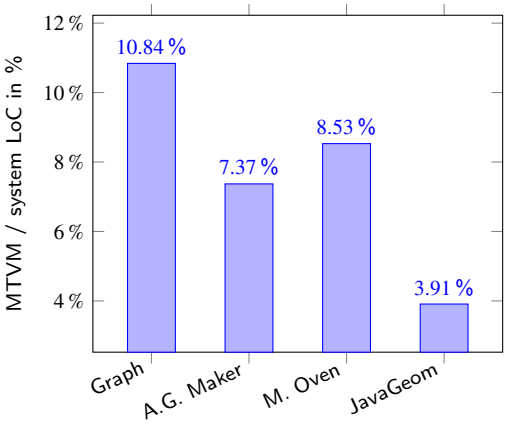


Figure 15: The MTVM's LoC to system's LoC ratio in four subject systems.

in 3 TVMs, which contain in total 7 *vp*-s with 12 variants, whereas JavaGeom subject took 11 TVMs containing 276 *vp*-s with 665 variants.

It is important to mention that all TVMs in JavaGeom are at the package level, as we intended to minimize the dependencies of *vp*-s across different TVMs, whereas in three other subject systems they are at the file level. Although the granularity of TVMs seems to be related to the size of the variability-rich system, a combination of TVMs at the package, file, or class level would have been also possible. For instance, Figure 14 shows the eight TVMs at the file level for the Microwave Oven subject system.

Technically, we stored all modeled variability of a variability-rich system directly into a map structure representing the power_set, that is, the *main* TVM (MTVM). This indicates that, in addition to all individual TVMs, the VM-DSL makes available all modeled variability of a variability-rich system into a single model. Still, from the beginning, someone may choose to model all variability of a given variability-rich system into a single model.

This answers the remained part of the second research question (RQ_2). Specifically, these results show that more than one TVM can be used, at package, file, or class level, to model the variability of a variability-rich system. This fragmented modeling has been shown to have notable advantages, for example, on variability consistency checking [86].

6.5. Encountered multiplicity of trace links

In order to answer the third research question (RQ_3), we analysed the encountered multiplicity of trace links between domain features and *vp*-s with variants of code assets of a given variability-rich system.

To trace variability, the variability models at both abstraction levels are required, that is, the FM and the TVMs. In our approach, both of them should be in some way within the same variability management tooling so that the links can be established. In our case, we developed an extension to the VM-DSL so that a domain FM can be entered in our tool. We then conducted this part of the experiment with the Microwave Oven subject system. We decided for it as, in comparison to JavaGeom, we had knowledge for each of its *vp*-s with variants. Then, in comparison to Graph and Arcade Game Maker, it has the largest number of *vp*-s with variants (*cf.* Table 5).

After we prepared features of Microwave Oven into a textual format [12], we used the part of VM-DSL given in Listings 3 and 5 for tracing them to its modeled *vp*-s with variants. The results [13] showed that 17 *vp*-s and variants of Microwave Oven (*cf.* Table 5) map to their respective 17 features (*cf.* Table 4), that is, they have 1–to–1 mapping. Whereas 2 *vp*-s with their 2 variants (*cf.* Table 5), each, are mapped to only 2 features (*cf.* Table 4), that is, they have 1–to–3 mapping. For instance, Microwave Oven has a compound feature `WeightSensor` with two *alternative* features `AnalogWeight` and `DigitalWeight`, and also a feature `Light`. The compound feature `WeightSensor` is implemented by a single variation point, `vp_WeightSensor`, and each of its alternative features is implemented by a

[12] Its features into textual format are available here: https://github.com/ternava/Variability-CChecking/blob/master/out/production/Variability-CChecking/SPLexamples/fm_prop_microwaveoven_spl.txt

[13] The resulted mapping is available here: https://github.com/ternava/Variability-CChecking/blob/master/src/SPL_Analysed_Examples/MicrowaveOvenSPL/FeatureVPtraces.scala.

single variant, `v_AnalogWeight` and `v_DigitalWeight`. Thus, each feature is traced to one *vp* or variant, therefore their mapping is 1–to–1. As for the feature `Light`, it is implemented by `vp_Light` and its two variants `v_LightOn` and `v_LightOf`. In this case, a single feature is traced to one *vp* and two variants, therefore their mapping is 1–to–3.

As we found the two forms of mapping in this single subject system, which forms are also supported by the VM-DSL, we consider that these results can bring an answer to the third research question (RQ_3). They indicate that, even within a small variability-rich system, with few features and LoC, when traditional variability implementation techniques are used, the mapping of domain features to *vp*-s with variants of code assets is imperfect, that is, the 1–to–m multiplicity of trace links may be encountered.

6.6. MTVM to system ratio

In order to evaluate the overhead for modeling the variability by using the VM-DSL, we rely on the fact that the usage of the VM-DSL contributes to the overall lines of code in a variability-rich system. Therefore, to answer the last research question (RQ_4), we studied the *MTVM to system ratio*, defined below.

Definition 5. *The MTVM to system ratio is the percentage of lines of code in all technical variability models (TVMs) toward the lines of code for realizing the functionality in code assets of a variability-rich system.*

From Listing 2 one can observe that for capturing a *vp* or variant only a single line of code is needed, that is, only $\langle vp \rangle$ or $\langle variant \rangle$. For modeling the variability another line of code is also needed, using $\langle fragment \rangle$, or two lines for a nested *vp*. It is important to clarify that, for example, lines 14-15 in Listing 4 represent a single line of code in VM-DSL and are calculated so, but we have separated them into 2 or 3 lines for the sake of readability. Then, each new TVM requires always five additional lines of code (*cf.* Listing 4, lines 2, 4, 5, 12, and 13).

For each subject system, the calculated MTVM to system ratio is given in Figure 15. It shows that on average 7.66% of LoC are used to model their variabilities into TVMs and the rest is used for realizing their respective system's functionalities. It can be observed that, for instance, Graph has the fewest lines of code (*cf.* Table 4, 323 LoC) but needs around 11% LoC for modeling its variability. Then, Microwave Oven that has more lines of code (914 LoC) than Arcade Game Maker (651 LoC) also uses more LoC for variability modeling (8.53% compared to 7.37%, respectively).

This indicates that the overhead for using the VM-DSL, calculated by the MTVM to system ratio, does not depend directly on the lines of functional code of a variability-rich system. It actually is proportional to the number of *vp*-s, variants, and TVMs, thus answering the last research question (RQ_4). As such, this ratio is a basic indicator for measuring the overhead for modeling the variability by the VM-DSL. In the future, it could be extended by taking into account other elements, for example, the required time to identify for modeling *vp*-s with variants of some given code assets and to produce their LoC.

6.7. VM-DSL limitations

As a means to further address $Challenge_3$, in this section we showed how our proposed VM-DSL addresses the four research questions (*cf.* RQ_1-RQ_4 in Section 6.1) by applying it in four subject systems. The answers to these questions show the degree at which the VM-DSL, as a proposed prototype implementation of our framework, can be used to capture, model, and trace the imperfectly modular variability of a given variability-rich system. Still, we observed some limitations.

A limitation of our VM-DSL, but not of the framework itself, is that we could not apply it for tracing the variability at the finest granularity level (*e.g.*, at the expression level), as using *reflection* in Scala for tagging the variability is not possible at that level. A solution would be to refactor this finest-grained variability, although using *reflection* is not mandatory. Variability could also be captured by using a form of annotations [42]. However, these annotations would still require a manual process, as well as the development of another tool for parsing the annotated code and *automatically* extracting the variability information. In addition, annotations would stay amalgamated with code. On the other hand, *reflection* helps us to maintain the strong consistency between the *vp*-s with variants and their implementation in code assets. They help to keep the variability information separated from the code assets such that the TVMs can be part of the code asset; not amalgamated with its code, or separated into their own files.

In addition, we acknowledge the weak consistency (*i.e.*, corresponding relationship) between the implementation technique itself and the modeled logical relation, binding time, and the evolution properties of a *vp* with its variants (*cf.* Listing 4, lines 14-15). For instance, the technique of inheritance is identified and modeled by the person using the VM-DSL, instead of being automatically detected by a tool, such as *symfinder* [89]. Inconsistencies might happen in this part of the modeled variability. However such inconsistencies would happen also in the case when annotations are used, as they are also given manually.

7. Framework usage

To complement the previous validation and directly tackle the $Challenge_2$ (*cf.* Section 3), we demonstrate here the usage of our framework for variability management. We aim at showing that our proposed approach for modeling some implemented variability and establishing trace links can be exploited to check the consistency of available models of variability. To do so, we rely on one of our previous work [86], which was only focusing on consistency checking, and put it into perspective to show how its requirements are met by our tooled framework.

7.1. Requirements for consistency checking

Within an SPL, it is a well-known desirable property to check whether the specified variability (*e.g.*, the feature model as in Figure 1) and implemented variability (*e.g.*, the variability model of code assets as in Listing 1) are consistent [63]. To drive our analysis, we first define a consistency rule that represents what must be satisfied by the variability models at specification and implementation levels.

Definition 6. *(Consistency rule) Within an SPL or variability-rich system, where the specified domain variability and the implemented variability convey the same functionality, they also should represent the same set of software products.*

Taking into account the mappings between a feature model and the modeled implemented variabilities, we identify and aim to address the following three consistency requirements.

$Reqmt_1$: One should be able to check the consistency of variabilities when the mapping between features and *vp*-s with variants is 1-to-n, including the 1-to-1 mapping.

$Reqmt_2$: In case that some variability is implemented by using an improper technique, an inconsistency may appear, for example, when an *alternative* logical relation between features is implemented by a *vp* that has an *Or* logical relation between its variants. Thus, one should be able to check the consistency of features logical relations in an FM with regard to the used variability implementation techniques.

$Reqmt_3$: To make consistency checking practical, early detection of inconsistencies is also needed, to reduce costs of correction of erroneous implementations, but also as some variability may be deferred to be implemented later.

We expect that these requirements will be supported by our framework, and its VM-DSL implementation, by relying on trace links, different *vp*-s types, and TVMs to build a consistency checking method.

7.2. Design and implementation of the consistency checking method

For checking the consistency of the entire variability of a variability-rich system, that is, between all its features and *vp*-s with variants under their 1-to-m mapping, it is required that *(i)* the whole specified variability in an FM is implemented and modeled in terms of *vp*-s with variants in an MTVM, and *(ii)* all their trace links are established. This restricts checking to a complete system, which itself is likely to be represented by large variability models [17], harder to check, but also harder to trace and fix inconsistencies after all of them are shown at once [92]. In addition, it is common that some variability implementation is deferred, for example, during the application engineering phase. Therefore, some partial checking is then highly desirable. Besides, even for illustrative variability-rich systems with a small set of features, the propositional formula of variability models becomes quite large. Moreover, in realistic variability-rich systems, checking for inconsistencies only within a single FM still expose scalability issues [15]. To overcome these problems, we propose a *consistency checking method* for detecting the variability inconsistencies earlier during the development process.

The proposed consistency checking method takes as input a feature model (FM), representing the domain variability, the trace links (TLs), and a TVM or a subset of them, created with our tooled framework. The method enables early detection of variability inconsistencies by relying on three main checking steps, namely *slicing*, *substitution*, and *assertion*. Each of these steps is based on propositional logic[14].

- **Slicing.** During this step, a single TVM or a subset of them from the MTVM of a variability-rich system is selected and used to slice the FM. The purpose of the slicing operation [1] is to select for checking only some of the desired variability, thus making possible an early consistency checking that only focuses on the already implemented features.

[14] In the following, the considered feature models are only propositional ones.

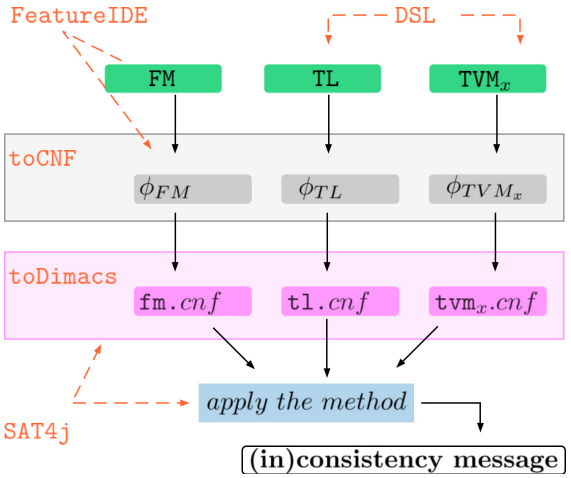


Figure 16: Prototype toolchain for consistency checking. Inputs: feature model (FM), trace links (TL), and technical variability model (TVM$_x$). Their respective propositional formula: ϕ_{FM}, ϕ_{TL}, and ϕ_{TVM_x}. Their respective DIMACS format: $fm.cnf$, $tl.cnf$, and $tvm_x.cnf$.

- **Substitution.** After the slicing step, the consistency rule states that domain features convey the same functionality with their traced *vp*-s with variants at the implementation level. Technically, this implies that abstractions of features should be substitutable with the abstractions of their traced *vp*-s or variants. The purpose is to check whether features are mapped correctly to their corresponding *vp*-s and variants.
- **Assertion.** Finally, if features can be substituted with their traced *vp*-s or variants, then an inconsistency is detected by asserting if the number of feature configurations is equal with the number of *vp*-s with variants configurations.

The consistency checking method has been implemented using the Scala language[15]. As our VM-DSL is also realized in Scala (*cf.* Section 5), we could use the TVMs and trace links directly. The resulting toolchain is depicted in Figure 16 with the three inputs, FM, TLs, and TVM (or a subset of them). While the FM is converted to a propositional formula using FeatureIDE [62], we implemented the conversion for the TLs and the TVMs. Next, their propositional logic formulas are converted to a *conjunctive normal form* (CNF), namely ϕ_{FM}, ϕ_{TL}, and ϕ_{TVM_x}, respectively. Then, their CNF formulas are encoded into DIMACS CNF format, namely $fm.cnf$, $tl.cnf$, and $tvm_x.cnf$, respectively, so that SAT solving techniques can be used for consistency checking, using the Sat4j library. More details on the method and on the performance of its implementation are provided in a previous work [86].

The implemented method has also been applied to the Microwave Oven subject system. We report only on this subject system as, in contrast to Graph and Arcade Game Maker subjects, we encountered multiple trace links between its domain features and captured *vp*-s with variants (*cf.* Section 6.5), which is one of the requirements for our consistency checking method ($Reqmt_1$). Then, in comparison with the JavaGeom, we used its externally created feature model [38], while for JavaGeom we had to create it directly from the code assets. Thus, to avoid any further bias and data manipulation, we decided to evaluate our method with the Microwave Oven system. This variability-rich system has a feature model with 23 implemented features (*cf.* Table 4), 8 TVMs with 11 *vp*-s and 20 variants in total (*cf.* Table 5). It also contains the 1–to–m trace links established during Section 6.5. By selecting a single or a subset of these TVMs at a time, the consistency checking is made possible early in the development process, that is, as soon as the specified variability is implemented. But, we also expected that the consistency checking time for the resulting sliced formulas should be smaller. As for comparison, we recorded the execution time for some TVMs in this subject system. The measurements are performed on a PC with 2.50 GHz and 4 GB RAM, on Windows 10. For instance, we selected for checking one TVM with 7 *vp*-s and variants. The execution time was almost instantaneous. Specifically, the execution time for the sliced FM was *2 ms*, for the TVM itself was *7 ms*, and the time for checking their consistency was *5 ms*. Then, for comparison, we measured the execution time for checking the validity of only the whole FM of Microwave Oven system, which has 26 features and 720 configurations. It took around *0.13 seconds*. As we

[15]Its implementation is publicly available at https://github.com/ternava/variability-cchecking.

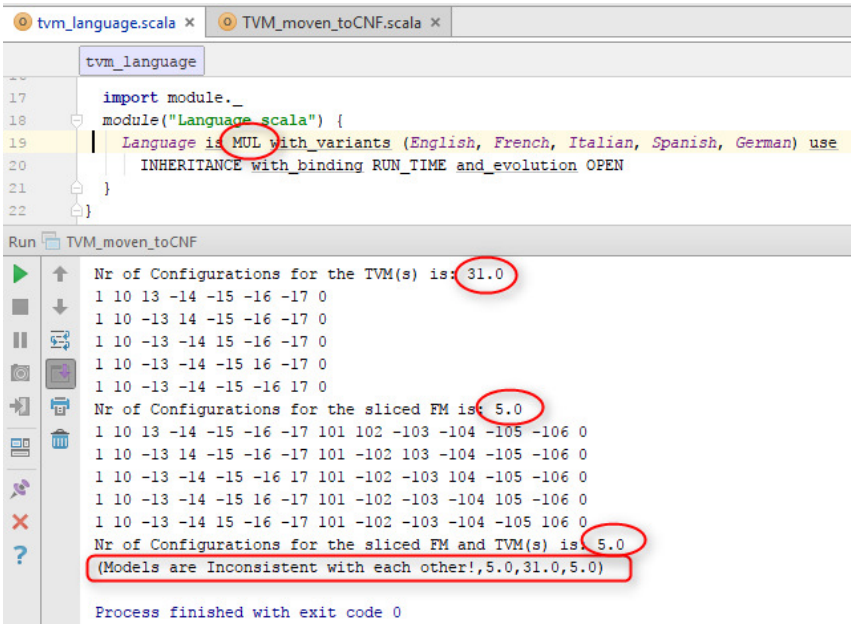


Figure 17: Detected inconsistency between the FM and a TVM in the Microwave Oven variability-rich system.

expected, the execution times for checking the consistency of partial variability compared to the validity of only the FM is smaller.

Figure 17 shows the output of an inconsistency detection. According to our *consistency rule*, the implemented method will report any (in)consistency between FM and TVMs as a difference between their numbers of configurations. In this example, the number of configurations for the FM is 5 and for the TVMs is 31, which indicates that the FM and the selected TVMs variabilities are inconsistent. The reported numerical format of an inconsistency helps to show only that an inconsistency exists between the FM and the checked TVMs of a system. In this prototyped usage of our framework, the focus of our method is only to detect inconsistencies. Helping developers in addressing this inconsistency is a known non-trivial problem [63], a desirable enhancement would thus be to point to the cause of the detected inconsistency and to recommend a solution for it. For instance, it could be done by explicitly showing which features are inconsistent to which *vp*-s with variants, or whether the inconsistency has happened because the trace links may not be well-established. Nothing in the framework hampers the realization of this extension.

7.3. Evaluation

We first evaluate the consistency checking method according to the three given requirements, $Reqmt_1$ to $Reqmt_3$. Implementing the method, we successfully could check the variability when the mapping between features and *vp*-s with variants is 1-to-1 and 1-to-m, thus fulfilling the first requirement ($Reqmt_1$). The second requirement, $Reqmt_2$, implies checking whether the properties of features, such as their logical relation, correspond to the logical relations of *vp*-s with variants. Within the method, this checking process happens during the substitution and assertion steps. Then, for all modeled TVMs in the subject system, we were able to select one TVM at a time or a subset of them, using the slicing step. This makes possible to check earlier the implemented variability, meeting requirement $Reqmt_3$. We successfully checked the consistency of variabilities under four logical relations of features, respectively *vp*-s with variants, namely *mandatory*, *optional*, *or*, and *alternative* ones. In this case we partially fulfilled the second requirement, as we have not checked the consistency regarding other properties, for instance, binding time. The main reason was that these other properties were missing in the FM.

Moreover, the proposed method denotes the usage of our designed TVMs and trace links given in Section 4.2 and Section 4.3, respectively. Its evaluation with the Microwave Oven shows that, it handles the case of checking consistency when *(i)* variability is implemented by different variability implementation techniques and *(ii)* *vp*-s with variants can have an 1-to-1 and 1-to-m mapping to the specified features in an FM.

Overall, this consistency checking method and its prototype implementation demonstrate a typical usage of our

framework for variability management, thus addressing the second challenge (*cf.* $Challenge_2$ in Section 3). Further, the fulfilled requirements ($Reqmt_1$, $Reqmt_2$, and $Reqmt_3$) and the successful application to one of our subject systems give us confidence in the good applicability of our framework and its implementation, the VM-DSL, as a concrete solution for modeling and tracing the variability of code assets.

8. Threats to validity

In addition to the VM-DSL limitations given in Section 6.7, we discuss here several validity threats in our validation.

8.1. Construct validity

A first threat to validity is related to *the experiment bias* regarding the selected subject systems. We used three academic variability-rich systems as subject systems that we implemented ourselves. As we discussed in Section 6.2, this threatens the validity of the obtained results on research questions that aim at characterizing the encountered implementation techniques. To reduce this bias, we tried to implement the variability of these systems by following good practices in object-oriented design and, whenever was possible, by using the specified techniques in their original design, namely in [60, 2, 38]. But, there could be other forms of implementation that would handle the considered variability while being not detected by our framework. Still, the proposed framework supports most of the traditional techniques already surveyed for variability implementations [33, 32, 72, 84, 87], which are themselves very common, such as inheritance and overloading.

8.2. Internal validity

There are several approaches that reveal that domain features crosscut in code assets of variability-rich systems, for instance, in the case of the Arcade Game Maker subject system [25]. In our framework, we consider that a feature may have a trace to a *vp* or variant, which have a direct tag to code assets. It then may seem that features crosscutting in code assets are omitted, forming a threat on *ignoring* some facts. Actually, we acknowledged it at the very beginning and proposed to trace features to *vp*-s and variants, which represent the actual implemented variability of code assets of a system, instead of tracing features directly to code assets. Then, we consider that features to *vp*-s and variants can have n–to–1 multiplicity of trace links, that is, they may crosscut but to the *vp*-s and variants.

For evaluating the consistency checking method, we used a subject system that has only ordinary and optional *vp*-s. It can be viewed as a threat related to the *incompleteness* of the considered types of *vp*-s for evaluating the method, such as nested and technical *vp*-s. Considering them and improving the method accordingly is inline with our future work.

Another threat is related to the difficulty we faced during the application of the VM-DSL to subject systems, especially on recognizing whether all used techniques are variability related. For instance, it was sometimes hard to distinguish whether an inheritance hierarchy implements some (product line) variability (*i.e.*, which has in purpose to differentiate products in an SPL or variability-rich system) or some internal functionality that should not be modeled or traced. Because of this *deficiency* of information during our experiments, we decided to classify all used techniques as variability related techniques. However, in real systems this may not be always true [66]. For instance, there are approaches that consider that all C preprocessor directives in a system are used for implementing some PL variability, whereas some others disagree [95]. Specifically, Zhang et al. [95] state that "*from our experience most #ifdef blocks (e.g., 87.6% in the Danfoss SPL) are actually not variability related, but for other purposes such as include guards or macro substitution*". Although this is not a concern that is particular to our framework or its realization, it shows a limitation to the generalization of the approach. Recent works [89, 66] on automatic detection of some object-oriented techniques rely on the density of variability implementations within a set of classes to determine areas of interest in terms of variability, but this technique is not precise enough and still needs some human expertise to map features to variability implementations [66].

8.3. Conclusion validity

A threat to conclusion validity is the *lack of expert evaluation* of our tooled framework by software architects in their variability-rich systems. Such an evaluation would help us to have a stronger interpretation of results and to further identify the possible improvements of our tooled framework, for instance, by reporting the time spent and difficulties to model the variability of a whole real variability-rich system by our VM-DSL. Considering its importance, evaluating our tooled framework in real systems and by their experts is also part of our future work.

8.4. External validity

While we believe the different experiments show the feasibility of the propositions for modeling and tracing the considered techniques, the number and diversity of variability-rich systems do not enable us to conclude on a large applicability of the proposed framework and implementation.

On the covered techniques in the subject systems, their number in the three academic variability-rich systems were four, but in the fourth real variability-rich system, JavaGeom, the total number of used techniques became seven. It cannot be determined whether more real systems would enable the validation over a larger set of techniques. Nevertheless, as we discussed for the construct validity threat, the covered techniques by the framework are the most common ones. We also believe that such validation can successfully be extended to other variability-rich systems that use other techniques, for example, design patterns, as some recent works show that they are present in many object-oriented variability-rich systems [89].

As for the framework usage, we only analysed one scenario with consistency checking, but we believe it is a very relevant one to also show the feasibility of the usage. Obviously more extensions would bring a better validation on this aspect.

9. Related work

In the context of product lines and product families, several approaches related to our framework exist on variability management, including variability modeling and variability traceability of code assets. On the other side, there are alternative approaches to our framework. Further, several textual variability modeling languages related to our VM-DSL have been proposed. In the following we discuss the related and alternative approaches to our framework, as well as related works to our DSL implementation.

9.1. Variability management approaches

Variability traceability. Several approaches on variability management provide detailed relationship or mapping models between the variability of core assets at different abstraction levels. Most of these models came in form of a general mapping model [13, 12], metadata model [71], basic traceability model [65], a representation independent variability meta-model [80, 79], a conceptual model for traceability [16], or a framework for variability modeling and management in all abstraction levels [82, 22, Ch.9]. A common aspect of these approaches is a formal specification of the relations or associations among the specified variability (*i.e.*, features in an FM), the *vp*-s with variants, as variability abstractions in different core assets, and core assets themselves. However, in all these approaches the first steps on capturing and modeling the variability are not considered. In this work, we provide a detailed framework, which has in purpose to handle the management of variability at the implementation level. Specifically, we distinguish the variability model at specification and implementation levels in a similar way to Becker [13] and Metzger *et al.* [63], and we keep the variability information separated from the code assets as by Berg *et al.* [16] and Pohl *et al.* [75]. However, we go further by detailing the earlier steps of traceability, such as the capture and modeling of variability in terms of *vp*-s and variants, which is the main difference with, or a kind of complement of, other existing approaches.

Besides these approaches, there are other proposals that specifically address the traceability of variability. Mostly, they address the general issues of variability traceability management in SPL engineering, such as establishing, using, and/or managing the trace links, without much focus on how the variability in different abstraction levels, or types of core assets, is realized or identified, and modeled. For instance, AMPLE [5] is a general framework based on a reference meta-model for tracing the variability of different types of artifacts by several types of trace links. It discusses the main traceability dimensions in SPL engineering, specifically when model-driven engineering and aspect-orientation are applied. Quite similarly, XTraQue [46] and the approach by Bayer and Widen [11] target tracing the variability in different types of artifacts. In XTraQue the main idea is to translate the product line documents, as the UML documents, into XML documents and to trace their variability by using several types of trace links.

Although they consider that the variability in core assets is already made explicit in some way and need just to be traced, we consider that these works are complementary with our contribution. Specifically, while they deal more with the abstract level of variability, we provide details on how to capture and model the variability of code assets.

Variability modeling. As for the variability modeling, some other works exist regarding the types of *vp*-s and variants, for instance, by Jacobson *et al.* [45], Schmid *et al.* [80], Sinnema *et al.* [82, 81], and Capilla *et al.* [22]. We used their notion of *vp*-s and *variants* and several of their findings regarding the diversity of *vp*-s. Yet, we consider that

the types of *vp*-s come from the used variability implementation techniques themselves. In addition, for modeling the variability in a fragmented way we get inspired by the variability-aware module system [52] where each module defines its own local variability model. However, in this approach, a `.c` file is considered as a module with inner variability, which has only *compile time* binding. In the contrary, what is considered as a fragment in our framework aims at being flexible, such as a package, file, or class. Then, possibly, fragments are kept separated from the code assets. Besides, we consider that variability is implemented by traditional techniques, not including the preprocessor directives.

When preprocessors in C are used [59], the used directives are commonly recognized as *vp*-s and variants which are more explicit compared to the case when traditional techniques are used. If we consider preprocessor directives, they also can offer all kinds of logical relations between variants in a *vp*. Although compared to the case when several traditional techniques are used, they offer a single binding time for *vp*-s. Still, as preprocessors are a form of annotations with variability information between variants, it could be interesting to apply our framework for modeling and tracing the implemented variability when only this technique is used.

Variability management tools. Furthermore, there are prototypes and commercial tools for variability management, such as FeatureIDE [62], pure::variants [36], or domain-specific languages [17]. While FeatureIDE is a tool support for the orthogonal approaches to variability traceability (*cf.* Section 9.2), pure::variants provides a set of integrated tools to support each phase of the SPL development process, including the implementation phase. Their variability management approach is not much related to the used programming paradigm. Thus, it can be used to manage the variability implemented by preprocessors in C/C++, or implemented by other functional or object-oriented techniques. In the last case, its captured way of variability relies on annotations, a form of macro-like calls. In contrast, instead of annotating the domain features to code assets of a variability-rich system we use a textual DSL to model *vp*-s with variants of code assets, that is, the structure of code, through the capture of the used implementation technique inside a package, file, or class. Then, while the main purpose in pure::variants is variability resolution, we provide a slightly more general approach by not becoming specific on how the *vp*-s and trace links are going to be used. For instance, one could decide to use them for consistency checking (*cf.* Section 7) or variability resolution.

9.2. Orthogonal approaches to variability traceability

When traditional techniques are used for implementing the variability in an SPL or variability-rich system, the code is not shaped in terms of features [45, 20, 84, 22]. In these techniques, the main concerns being separated are objects and/or functions, depending on the used programming paradigm. Thus, a feature can be implemented as a file, class, method, plugin, package, a combination of them or by any part of them. Therefore, as is shown in our framework, the trace relation is n–to–m between the specified features to the *vp*-s with variants at the implementation level. But, there are approaches that shape the code assets in terms of features [6, 64, 78, 8], or the specified features hold a direct representation in code [42, 53], that is, their mapping is 1–to–1. We consider that these are alternative approaches to the variability traceability itself, and to our framework, as the traceability of features is straightforward in this case.

9.3. Textual variability modeling approaches

In addition to the graphical-based variability modeling approaches, there are several textual variability modeling languages. Specifically, a recent study provides a summary and a comparison, regarding their provided support, of up to fourteen textual variability modeling languages [14]. In almost all of them, the main language concept is feature, whereas the main concepts in our VM-DSL are *vp*-s and variants. Besides, from their mentioned aspects that are relevant when designing new languages, we provide support for *binding time*, but not only – through capturing characteristic properties of *vp*-s with variants, *analyzability* – through supporting the conversion of TVMs and trace links to propositional logic, *expressiveness* – through using the futures of Scala for realizing expressive domain-specific languages, and *extensibility* – through providing the language syntax. Moreover, we provide a textual variability modeling language that aims to capture and model the implemented variability of variability-rich systems, which may not be necessarily organized as a software product line.

10. Conclusion and future work

Summary. Variability management is a complicated activity during the engineering phases of software product lines or variability-rich systems. In many settings, variability is specified in terms of features, and they are implemented by using a combined set of traditional techniques, such as inheritance, overloading, or design patterns. In such a case, they

do not have an explicit mapping at the implementation level, which leads to a form of imperfectly modular variability and strongly hinders the variability management at the code level.

In this work, we provided a three step framework for managing this imperfectly modular variability of code assets of a variability-rich system. It supports *(i)* the capture of the implemented variability in terms of variation points (*vp*-s) with variants, *(ii)* the fragmented modeling of these *vp*-s and variants, with tags to their respective code assets, into technical variability models (TVMs), and *(iii)* establishing the n-to-m trace links between domain features and their respective *vp*-s or variants. In particular, during the first step different types of *vp*-s with variants are captured, including their diverse characteristic properties, namely their logical relation, binding time, evolution property, and granularity. These types and properties are used to model the variability of code assets into TVMs, which form a forest-like structure of variability at the implementation level. This enables a form of fragmented variability modeling which, together with the established trace links, can be used for different purposes, such as for resolving the variability, evolving, checking its consistency, or simply comprehending the addressed variability in code assets.

We implemented the framework as an internal textual *domain-specific language* (DSL) in Scala, and applied it to four different subject systems being variability-rich systems. The results show that in realistic systems, a combined set of traditional techniques can indeed be used to implement the variability. For instance, in the four used subject systems we captured and modeled the implemented variability with up to seven common variability implementation techniques, namely inheritance, overloading, generic types, parameters, template pattern, strategy pattern, and include package. Different types of *vp*-s with their diverse characteristic properties and variants were also encountered. Furthermore, a different number of TVMs were successfully used to model the variability. In addition, the 1-to-m variability trace links were evident even in an academic variability-rich system. While being on a small set of subject systems, these results show the feasibility of our framework and its potential application in realistic variability-rich systems with a small textual DSL.

Additionally, we have proposed a method for checking the consistency between the specified and implemented variabilities. The method was also implemented and applied to one of the subject systems. Its application shows that the captured variability of code assets and their established traced links to the domain features can be successfully exploited to check their consistency. This shows the potential usage of our contribution as a framework for variability management, which was our broader objective.

Future Work. In future work, we first plan to complement the framework to support the modeling and reasoning over dependencies between *vp*-s with variants, which dependencies are different from the cross-tree constraints between features in feature models at the specification level. In addition, as we model variability in several TVMs in the framework, we will also have to support *vp*-s with variants in one TVM having dependencies to *vp*-s with variants in other TVMs. In the literature, there are already some frameworks that handle dependencies of *vp*-s [21, 80], which can serve as a starting point for this extension. Further, we will evaluate the scalability of our tooled framework on a larger set of variability-rich systems, and with an application by other users. In particular, by applying it to a broader range of real variability-rich systems, we expect to validate our framework and the DSL on a larger set of variability implementation techniques, going beyond the seven used techniques in this contribution.

Ongoing work that tackles the automatic capturing and modeling of *vp*-s in similar kinds of code structure [89, 66] or a recommendation approach based on our framework for modeling *vp*-s with variants, comparable to the code recommendation approaches for reuse [61], could also be considered to be coupled to the proposed contribution, so to reduce the burden of manual variability capturing and modeling and only on relevant or intricate variations points. In addition, we are working on the interoperability of our tool with other variability management tools, such as pure::variants, by using the *variability exchange language* (VEL) [69], as a coming standard for exchanging variability information between variability management tools. By this, we will aim at demonstrating the usage of our framework for resolving variability during product derivation.

With these extensions and more evaluations, we hope that new insights can emerge, for example, to highlight patterns of variability in the code structure and thus to better help practitioners in managing variability and code reuse.

References

[1] Acher, M., Collet, P., Lahire, P., France, R.B., 2011. Slicing feature models, in: 2011 26th IEEE/ACM International Conference on Automated Software Engineering, IEEE. pp. 424–427. URL: https://doi.org/10.1109/ASE.2011.6100089, doi:10.1109/ASE.2011.6100089.

[2] AGM, 2009. Arcade game maker pedagogical product line. https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=485941. [Accessed 2020-07-26].

[3] Alves, V., Schneider, D., Becker, M., Bencomo, N., Grace, P., 2009. Comparitive study of variability management in software product lines and runtime adaptable systems .

[4] Anquetil, N., Grammel, B., Galvão, I., Noppen, J., Khan, S.S., Arboleda, H., Rashid, A., Garcia, A., 2008. Traceability for model driven, software product line engineering, in: ECMDA Traceability Workshop Proceedings, SINTEF. pp. 77–86.

[5] Anquetil, N., Kulesza, U., Mitschke, R., Moreira, A., Royer, J.C., Rummler, A., Sousa, A., 2010. A model-driven traceability framework for software product lines. Software & Systems Modeling 9, 427–451. URL: https://doi.org/10.1007/s10270-009-0120-9, doi:10.1007/s10270-009-0120-9.

[6] Apel, S., Batory, D., Kästner, C., Saake, G., 2016. Feature-Oriented Software Product Lines. Springer.

[7] Apel, S., Beyer, D., 2011. Feature cohesion in software product lines: An exploratory study, in: Proceedings of the 33rd International Conference on Software Engineering, ACM. pp. 421–430. URL: https://doi.org/10.1145/1985793.1985851, doi:10.1145/1985793.1985851.

[8] Apel, S., Leich, T., Saake, G., 2008. Aspectual feature modules. IEEE Transactions on Software Engineering 34, 162–180. URL: https://doi.org/10.1109/TSE.2007.70770, doi:10.1109/TSE.2007.70770.

[9] Bachmann, F., Clements, P.C., 2005. Variability in Software Product Lines. Technical Report CMU/SEI-2005-TR-012. Carnegie Mellon University Pittsburgh PA Software Engineering Institute.

[10] Bassett, P.G., 1996. Framing Software Reuse: Lessons From the Real World. Prentice-Hall, Inc.

[11] Bayer, J., Widen, T., 2001. Introducing Traceability to Product Lines. Master's thesis. URL: https://doi.org/10.1007/3-540-47833-7_38, doi:10.1007/3-540-47833-7_38.

[12] Becker, M., 2003a. Mapping variabilities onto product family assets, in: Proceedings of the International Colloquium of the Sonderforschungsbereich, p. 12.

[13] Becker, M., 2003b. Towards a general model of variability in product families, in: Workshop on Software Variability Management, pp. 19–27.

[14] Beek, M.H.t., Schmid, K., Eichelberger, H., 2019. Textual variability modeling languages: An overview and considerations, in: Proceedings of the 23rd International Systems and Software Product Line Conference-Volume B, ACM. pp. 151–157. URL: https://doi.org/10.1145/3307630.3342398, doi:10.1145/3307630.3342398.

[15] Benavides, D., Segura, S., Ruiz-Cortés, A., 2010. Automated analysis of feature models 20 years later: A literature review. Information Systems 35, 615–636. URL: https://doi.org/10.1016/j.is.2010.01.001, doi:10.1016/j.is.2010.01.001.

[16] Berg, K., Bishop, J., Muthig, D., 2005. Tracing software product line variability: From problem to solution space, in: SAICSIT, Citeseer. pp. 182–191. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.107.9241&rep=rep1&type=pdf, doi:10.1.1.107.9241.

[17] Berger, T., Rublack, R., Nair, D., Atlee, J.M., Becker, M., Czarnecki, K., Wąsowski, A., 2013. A survey of variability modeling in industrial practice, in: Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems, ACM. pp. 1–8. URL: https://doi.org/10.1145/2430502.2430513, doi:10.1145/2430502.2430513.

[18] Bosch, J., 2000. Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach. Pearson Education.

[19] Bosch, J., Capilla, R., 2012. Dynamic variability in software-intensive embedded system families. Computer , 28–35URL: http://dx.doi.org/10.1109/MC.2012.287, doi:10.1109/MC.2012.287.

[20] Bosch, J., Florijn, G., Greefhorst, D., Kuusela, J., Obbink, J.H., Pohl, K., 2001. Variability issues in software product lines, in: International Workshop on Software Product-Family Engineering, Springer. pp. 13–21. URL: https://doi.org/10.1007/3-540-47833-7_3, doi:10.1007/3-540-47833-7_3.

[21] Buhne, S., Halmans, G., Pohl, K., 2003. Modeling dependencies between variation points in use case diagrams, in: REFSQ, Citeseer. pp. 59–69.

[22] Capilla, R., Bosch, J., Kang, K.C., et al., 2013. Systems and software variability management. Concepts Tools and Experiences .

[23] Cleland-Huang, J., Gotel, O., Zisman, A., et al., 2012. Software and Systems Traceability. volume 2; 3. Springer. URL: https://doi.org/10.1007/978-1-4471-2239-5, doi:10.1007/978-1-4471-2239-5.

[24] Cleland-Huang, J., Gotel, O.C., Huffman Hayes, J., Mäder, P., Zisman, A., 2014. Software traceability: Trends and future directions, in: Future of Software Engineering Proceedings. ACM. FOSE '14, pp. 55–69. URL: https://doi.org/10.1145/2593882.2593891, doi:10.1145/2593882.2593891.

[25] Conejero, J.M., Hernández, J., 2008. Analysis of crosscutting features in software product lines, in: Proceedings of the 13th International Workshop on Early Aspects, pp. 3–10. URL: https://doi.org/10.1145/1370828.1370831, doi:10.1145/1370828.1370831.

[26] Coplien, J.O., 1999. Multi-Paradigm Design for C++. Addison-Wesley Longman Publishing Co., Inc.

[27] Czarnecki, K., 2004. Overview of generative software development, in: International Workshop on Unconventional Programming Paradigms, Springer. pp. 326–341. URL: https://doi.org/10.1007/11527800_25, doi:10.1007/11527800_25.

[28] Czarnecki, K., Grünbacher, P., Rabiser, R., Schmid, K., Wąsowski, A., 2012. Cool features and tough decisions: A comparison of variability modeling approaches, in: Proceedings of the sixth international workshop on variability modeling of software-intensive systems, ACM. pp. 173–182. URL: https://doi.org/10.1145/2110147.2110167, doi:10.1145/2110147.2110167.

[29] Eichelberger, H., Schmid, K., 2013. A systematic analysis of textual variability modeling languages, in: Proceedings of the 17th International Software Product Line Conference, ACM. pp. 12–21. URL: https://doi.org/10.1145/2491627.2491652, doi:10.1145/2491627.2491652.

[30] Eichelberger, H., Schmid, K., 2015. Mapping the design-space of textual variability modeling languages: A refined analysis. International Journal on Software Tools for Technology Transfer 17, 559–584. URL: https://doi.org/10.1007/s10009-014-0362-x, doi:10.1007/s10009-014-0362-x.

[31] Fowler, M., Parsons, R., 2010. Domain-Specific Language. Pearson Education.

[32] Fritsch, C., Lehn, A., Strohm, T., Bosch, R., 2002. Evaluating variability implementation mechanisms, in: Proceedings of International Workshop on Product Line Engineering, sn. pp. 59–64.

[33] Gacek, C., Anastasopoules, M., 2001. Implementing product line variabilities, in: Proceedings of the 2001 Symposium on Software Reusability: Putting Software Reuse in Context, ACM. pp. 109–117. URL: https://doi.org/10.1145/375212.375269, doi:10.1145/375212.375269.

[34] Galster, M., Weyns, D., Tofan, D., Michalik, B., Avgeriou, P., 2013. Variability in software systems — a systematic literature review. IEEE Transactions on Software Engineering 40, 282–306. URL: https://doi.org/10.1109/TSE.2013.56, doi:10.1109/TSE.2013.56.

[35] Ghosh, D., 2010. DSLs in Action. Manning Publications Co.

[36] pure-systems GmbH, 2020. pure::variants. URL: https://www.pure-systems.com/products/pure-variants-9.html.

[37] Gomaa, H., 2004. Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures. Addison-Wesley Professional, USA.

[38] Gomaa, H., 2005. Designing Software Product Lines with UML. IEEE.

[39] Gotel, O., Cleland-Huang, J., Hayes, J.H., Zisman, A., Egyed, A., Grünbacher, P., Dekhtyar, A., Antoniol, G., Maletic, J., 2012. The grand challenge of traceability (v1. 0), in: Software and Systems Traceability. Springer, pp. 343–409. URL: https://doi.org/10.1007/978-1-4471-2239-5_16, doi:10.1007/978-1-4471-2239-5_16.

[40] Haugen, O., Wąsowski, A., Czarnecki, K., 2013. CVL: common variability language, in: Proceedings of the 17th International Software Product Line Conference, ACM, New York, NY, USA. p. 277. URL: https://doi.org/10.1145/2491627.2493899, doi:10.1145/2491627.2493899.

[41] Helleboogh, A., Weyns, D., Schmid, K., Holvoet, T., Schelfthout, K., Van Betsbrugge, W., 2009. Adding variants on-the-fly: Modeling meta-variability in dynamic software product lines, in: Proceedings of the Third International Workshop on Dynamic Software Product Lines, Carnegie Mellon University; Pittsburgh, PA, USA. pp. 18–27. URL: https://lirias.kuleuven.be/1655532?limo=0.

[42] Heymans, P., Boucher, Q., Classen, A., Bourdoux, A., Demonceau, L., 2012. A code tagging approach to software product line development. International Journal on Software Tools for Technology Transfer 14, 553–566. URL: https://doi.org/10.1007/s10009-012-0242-1, doi:10.1007/s10009-012-0242-1.

[43] Hilliard, R., 2010. On representing variation, in: Proceedings of the Fourth European Conference on Software Architecture: Companion Volume, ACM. p. 312–315. URL: https://doi.org/10.1145/1842752.1842810, doi:10.1145/1842752.1842810.

[44] Hunt, J.M., McGregor, J.D., 2006. 9 implementing a variation point: A pattern language. 10th Software Product Line Conference. Workshop on Variability Management-Working with Variability Mechanisms , 83–96.

[45] Jacobson, I., Griss, M., Jonsson, P., 1997. Software Reuse: Architecture, Process and Organization for Business Success. Addison-Wesley Professional.

[46] Jirapanthong, W., Zisman, A., 2009. XTraQue: traceability for product line systems. Software & Systems Modeling 8, 117–144. URL: https://doi.org/10.1007/s10270-007-0066-8, doi:10.1007/s10270-007-0066-8.

[47] Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S., 1990. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report. Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst.

[48] Kang, K.C., Kim, S., Lee, J., Kim, K., Shin, E., Huh, M., 1998. FORM: a feature-oriented reuse method with domain-specific reference architectures. Annals of Software Engineering 5, 143. URL: https://doi.org/10.1023/A:1018980625587, doi:10.1023/A:1018980625587.

[49] Kästner, C., 2010. Virtual Separation of Concerns: Toward Preprocessors 2.0. PhD dissertation. Otto-von-Guericke-Universität Magdeburg.

[50] Kästner, C., Apel, S., Kuhlemann, M., 2008a. Granularity in software product lines, in: 2008 ACM/IEEE 30th International Conference on Software Engineering, IEEE. pp. 311–320. URL: https://doi.org/10.1145/1368088.1368131, doi:10.1145/1368088.1368131.

[51] Kästner, C., Apel, S., Ostermann, K., 2011. The road to feature modularity?, in: Proceedings of the 15th International Software Product Line Conference, Volume 2, ACM. pp. 1–8. URL: https://doi.org/10.1145/2019136.2019142, doi:https://doi.org/10.1145/2019136.2019142.

[52] Kästner, C., Ostermann, K., Erdweg, S., 2012. A variability-aware module system, in: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, ACM. pp. 773–792. URL: https://doi.org/10.1145/2384616.2384673, doi:10.1145/2384616.2384673.

[53] Kästner, C., Trujillo, S., Apel, S., 2008b. Visualizing software product line variabilities in source code, in: Proceedings of the 12th International Software Product Line Conference: 2nd International Workshop on Visualisation in Software Product Line Engineering, pp. 303–312.

[54] Kim, J., Kang, S., Lee, J., 2014. A comparison of software product line traceability approaches from end-to-end traceability perspectives. International Journal of Software Engineering and Knowledge Engineering 24, 677–714. URL: https://doi.org/10.1142/S0218194014500260, doi:10.1142/S0218194014500260.

[55] Lee, J., Kang, K.C., 2003. Feature binding analysis for product line component development, in: International Workshop on Software Product-Family Engineering, Springer. pp. 250–260. URL: https://doi.org/10.1007/978-3-540-24667-1_18, doi:10.1007/978-3-540-24667-1_18.

[56] Lee, J., Muthig, D., 2008. Feature-oriented analysis and specification of dynamic product reconfiguration, in: International Conference on Software Reuse, Springer. pp. 154–165. URL: https://doi.org/10.1007/978-3-540-68073-4_14, doi:10.1007/978-3-540-68073-4_14.

[57] Legland, D., 2019. JavaGeom - geometry library for java. URL: https://github.com/dlegland/javaGeom/tree/master/src. [Online; accessed 25-April-2020].

[58] Lettner, M., Rodas, J., Galindo, J.A., Benavides, D., 2019. Automated analysis of two-layered feature models with feature attributes. Journal of Computer Languages 51, 154–172. URL: https://doi.org/10.1016/j.cola.2019.01.005, doi:10.1016/j.cola.2019.01.005.

[59] Liebig, J., Apel, S., Lengauer, C., Kästner, C., Schulze, M., 2010. An analysis of the variability in forty preprocessor-based software product lines, in: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1, ACM. pp. 105–114. URL:

https://doi.org/10.1145/1806799.1806819, doi:10.1145/1806799.1806819.

[60] Lopez-Herrejon, R.E., Batory, D., 2001. A standard problem for evaluating product-line methodologies, in: International Symposium on Generative and Component-Based Software Engineering, Springer. pp. 10–24. URL: https://doi.org/10.1007/3-540-44800-4_2, doi:10.1007/3-540-44800-4_2.

[61] Luan, S., Yang, D., Barnaby, C., Sen, K., Chandra, S., 2019. Aroma: Code recommendation via structural code search. Proceedings of the ACM on Programming Languages 3, 1–28. URL: https://doi.org/10.1145/3361527, doi:10.1145/3361527.

[62] Meinicke, J., Thüm, T., Schröter, R., Benduhn, F., Leich, T., Saake, G., 2017. Mastering Software Variability with FeatureIDE. Springer.

[63] Metzger, A., Pohl, K., Heymans, P., Schobbens, P.Y., Saval, G., 2007. Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis, in: 15th IEEE International Requirements Engineering Conference, IEEE. pp. 243–253. URL: https://doi.org/10.1109/RE.2007.61, doi:10.1109/RE.2007.61.

[64] Mezini, M., Ostermann, K., 2004. Variability management with feature-oriented programming and aspects. ACM SIGSOFT Software Engineering Notes 29, 127–136. URL: https://doi.org/10.1145/1041685.1029915, doi:10.1145/1041685.1029915.

[65] Mohan, K., Ramesh, B., 2007. Tracing variations in software product families. Communications of the ACM 50, 68–73. URL: https://doi.org/10.1145/1323688.1323697, doi:10.1145/1323688.1323697.

[66] Mortara, J., Tërnava, Xh., Collet, P., 2020. Mapping features to automatically identified object-oriented variability implementations-the case of ArgoUML-SPL, in: 14th International Working Conference on Variability Modelling of Software-Intensive Systems, ACM. pp. 1–9. URL: https://doi.org/10.1145/3377024.3377037, doi:10.1145/3377024.3377037.

[67] Mukelabai, M., Behringer, B., Fey, M., Palz, J., Krüger, J., Berger, T., 2018. Multi-view editing of software product lines with PEoPL, in: 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion), IEEE. pp. 81–84. URL: https://doi.org/10.1145/3183440.3183499, doi:10.1145/3183440.3183499.

[68] Muthig, D., Patzke, T., 2002. Generic implementation of product line components, in: Net. ObjectDays: International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World, Springer. pp. 313–329. URL: https://doi.org/10.1007/3-540-36557-5_23, doi:10.1007/3-540-36557-5_23.

[69] OASIS, 2020. OASIS variability exchange language (VEL) TC. URL: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=vel. [Online; accessed 25-April-2020].

[70] Odersky, M., Spoon, L., Venners, B., 2007-2010. Programming in Scala. Artima Inc.

[71] de Oliveira Junior, E.A., Gimenes, I., Huzita, E.H.M., Maldonado, J.C., 2005. A variability management process for software product lines, in: Proceedings of the 2005 Conference of the Centre for Advanced Studies on Collaborative research, IBM Press. pp. 225–241.

[72] Patzke, T., Muthig, D., 2002. Product Line Implementation Technologies. Programming Language View. Technical Report. Fraunhofer IESE. URL: http://publica.fraunhofer.de/dokumente/N-14684.html.

[73] Patzke, T., Muthig, D., 2003. Product line implementation with frame technology: A case study. Fraunhofer IESE Report 18, 36–38. URL: http://publica.fraunhofer.de/documents/N-16381.html.

[74] Patzke, T.B., Fraunhofer, K., Rombach, D., Liggesmeyer, P., Bomarius, F., 2011. Sustainable Evolution of Product Line Infrastructure Code (PhD Theses in Experimental Software Engineering). Ph.D. thesis. URL: https://www.bookshop.fraunhofer.de/buch/sustainable-evolution-of-product-line-infrastructure-code/236533.

[75] Pohl, K., Böckle, G., van Der Linden, F.J., 2005. Software Product Line Engineering: Foundations, Principles and Techniques. Springer Science & Business Media.

[76] Rabiser, R., 2019. Feature modeling vs. decision modeling: History, comparison and perspectives, in: Proceedings of the 23rd International Systems and Software Product Line Conference-Volume B, ACM. pp. 134–136. URL: https://doi.org/10.1145/3307630.3342399, doi:10.1145/3307630.3342399.

[77] Rosenmüller, M., Siegmund, N., Apel, S., Saake, G., 2011. Flexible feature binding in software product lines. Automated Software Engineering 18, 163–197. URL: https://doi.org/10.1007/s10515-011-0080-5, doi:10.1007/s10515-011-0080-5.

[78] Schaefer, I., Bettini, L., Bono, V., Damiani, F., Tanzarella, N., 2010. Delta-oriented programming of software product lines, in: International Conference on Software Product Lines, Springer. pp. 77–91. URL: https://doi.org/10.1007/978-3-642-15579-6_6, doi:10.1007/978-3-642-15579-6_6.

[79] Schmid, K., John, I., 2003. Generic variability management and its application to product line modelling. Software Variability Management , 13–18.

[80] Schmid, K., John, I., 2004. A customizable approach to full lifecycle variability management. Science of Computer Programming 53, 259–284. URL: https://doi.org/10.1016/j.scico.2003.04.002, doi:10.1016/j.scico.2003.04.002.

[81] Sinnema, M., Deelstra, S., Nijhuis, J., Bosch, J., 2004a. COVAMOF: a framework for modeling variability in software product families, in: International Conference on Software Product Lines, Springer. pp. 197–213. URL: https://doi.org/10.1007/978-3-540-28630-1_12, doi:10.1007/978-3-540-28630-1_12.

[82] Sinnema, M., Deelstra, S., Nijhuis, J., Bosch, J., 2004b. Managing variability in software product families, in: Proceedings of the 2nd Groningen Workshop on Software Variability Management.

[83] Sobernig, S., Apel, S., Kolesnikov, S., Siegmund, N., 2016. Quantifying structural attributes of system decompositions in 28 feature-oriented software product lines. Empirical Software Engineering 21, 1670–1705. URL: https://doi.org/10.1007/s10664-014-9336-6, doi:10.1007/s10664-014-9336-6.

[84] Svahnberg, M., Van Gurp, J., Bosch, J., 2005. A taxonomy of variability realization techniques, Wiley Online Library. pp. 705–754. URL: https://doi.org/10.1002/spe.652, doi:10.1002/spe.652.

[85] Tërnava, Xh., 2017. Handling Variability at the Code Level: Modeling, Tracing and Checking Consistency. Ph.D. thesis. Université Côte d'Azur.

[86] Tërnava, Xh., Collet, P., 2017. Early consistency checking between specification and implementation variabilities, in: Proceedings of the 21st International Systems and Software Product Line Conference-Volume A, ACM. pp. 29–38. URL: https://doi.org/10.1145/3106195.

3106209, doi:10.1145/3106195.3106209.

[87] Tërnava, Xh., Collet, P., 2017. On the diversity of capturing variability at the implementation level, in: Proceedings of the 21st International Systems and Software Product Line Conference - Volume B, ACM. pp. 81–88. URL: https://doi.org/10.1145/3109729.3109733, doi:10.1145/3109729.3109733.

[88] Tërnava, Xh., Collet, P., 2017. Tracing imperfectly modular variability in software product line implementation, in: International Conference on Software Reuse, Springer. pp. 112–120. URL: https://doi.org/10.1007/978-3-319-56856-0_8, doi:10.1007/978-3-319-56856-0_8.

[89] Tërnava, Xh., Mortara, J., Collet, P., 2019. Identifying and visualizing variability in object-oriented variability-rich systems, in: Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A, pp. 231–243. URL: https://doi.org/10.1145/3336294.3336311, doi:10.1145/3336294.3336311.

[90] Thum, T., Kastner, C., Erdweg, S., Siegmund, N., 2011. Abstract features in feature modeling, in: 2011 15th International Software Product Line Conference, IEEE. pp. 191–200. URL: https://doi.org/10.1109/SPLC.2011.53, doi:10.1109/SPLC.2011.53.

[91] Turner, C.R., Fuggetta, A., Lavazza, L., Wolf, A.L., 1999. A conceptual basis for feature engineering. Journal of Systems and Software 49, 3–15. URL: https://doi.org/10.1016/S0164-1212(99)00062-X, doi:10.1016/S0164-1212(99)00062-X.

[92] Vierhauser, M., Grünbacher, P., Egyed, A., Rabiser, R., Heider, W., 2010. Flexible and scalable consistency checking on product line variability models, in: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, ACM. pp. 63–72. URL: https://doi.org/10.1145/1858996.1859009, doi:10.1145/1858996.1859009.

[93] Voelter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L., Visser, E., Wachsmuth, G., 2013. DSL Engineering. http://dslbook.org.

[94] Voelter, M., Visser, E., 2011. Product line engineering using domain-specific languages, in: 2011 15th International Software Product Line Conference, IEEE. pp. 70–79. URL: https://doi.org/10.1109/SPLC.2011.25, doi:10.1109/SPLC.2011.25.

[95] Zhang, B., Becker, M., Patzke, T., Sierszecki, K., Savolainen, J.E., 2013. Variability evolution and erosion in industrial product lines: A case study, in: Proceedings of the 17th International Software Product Line Conference, ACM. pp. 168–177. URL: https://doi.org/10.1145/2491627.2491645, doi:10.1145/2491627.2491645.

Acronyms

$\mathbb{BT}$ – The set of binding times. 11
$\mathbb{EV}$ – The evolution set. 12
$\mathbb{GR}$ – The set of granularity. 12
$\mathbb{LG}$ – The set of logical relations. 11
$\mathbb{T}$ – The set of implementation techniques. 8
$\mathbb{X}$ – The set of variation point's types. 13
CA – The set of all reusable code assets. 9
FM – The set of features. 17–19, 30
MTVM – Main Technical Variability Model. 16
TVM – Technical Variability Model. 16–19, 23, 25, 30
VP – The set of variation points. 9
V – The set of all realized variants. 9
$\mathbf{bt_x}$ – The binding time x. 13
$\mathbf{ca_x}$ – The reusable code asset x. 9
$\mathbf{ev_x}$ – The evolution property x. 13
$\mathbf{f_x}$ – The feature x. 17
$\mathbf{lg_x}$ – The logical relation x. 13
$\mathbf{t_x}$ – The implementation technique x. 8
$\mathbf{v_{xy}}$ – Variant x of a variation point y. 9
$\mathbf{vp_x}$ – Variation point x. 9
ALT – Alternative (*vp* or variant). 20
MND – Mandatory (*vp* or variant). 20
MUL – Or (*vp* or variant). 20
OPT – Optional (*vp* or variant). 20
VP – Ordinary variation point. 20
nVP – Nested variation point. 20
oVP – Optional variation point. 20
tVP – Technical variation point. 20
vp – Variation Point. 3, 4

CNF – Conjunctive Normal Form. 30

D – Dynamic binding time. 11
DSL – Domain Specific Language. 3, 19, 21, 34, 35

FM – Feature Model. 1, 2, 4, 6, 8, 10, 12, 14, 18, 27, 29–31, 33

S – Static binding time. 11
Sat4j – Sat solver. 30
SPL – Software Product Line. 1–5, 23, 24, 29, 32–34

TL – Trace Links. 29, 30

VEL – Variability Exchange Language. 35
VM-DSL – Variability Modeling DSL. 19–24, 27–30, 32–34