

A Framework for Managing the Imperfect Modularity of Variability Implementations

Xhevahire Tërnavaj^{a,*}, Philippe Collet^b

^aSorbonne Université, LIP6, Paris, France

^bUniversité 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 ¹domain specific language, and report on its application on four ²case studies 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, ¹the considered systems are commonly described in terms of their common and varying features, as their reusable units, in a *feature model* (FM) [38, 39, 20]. Then, in a forward engineering approach, their features are realized in different software assets, including reusable code assets. Consequently, the objective is that various systems can be systematically derived from a common and managed set of software assets.

²In many industrial settings, it is often the case that 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 features³. This hinders the management of the variability of code assets, which ⁴is extensive in realistic SPL settings.

⁵For example, let us consider the system of JavaGeom [48], as a realistic feature-rich system. It is an open-source geometry library for Java that is easily understandable. Although not presented ⁶as an SPL, it is architected around well-identified features⁷, being the size of medium-size SPLs. In Figure 1 is given an excerpt of its ⁸feature model, which consists of five features. The abstract feature [79] ⁹JavaGeomSPL does not require an

¹removed: domain-specific

²removed: subject systems

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

¹removed: the variability of considered systems is commonly

²removed: Although variability is largely studied in the context of software product lines and product families [20], most modern software-intensive systems, being organized as a software product line or not, are variability-intensive [? ?]. 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

³removed: or, simply put, neither features nor variation points with variants (their definition is given in Section 2) are by-product of traditional techniques [17].

⁴removed: seems to be extensive in real variability-rich systems [? 78].

⁵removed: For example, let us consider the system of JavaGeom [48], an example of a real variability-rich system, which is not organized as a software product line but is variability-intensive, based on its documentation, and is used in a previous study [78].

⁶removed: and organized

⁷removed: . Then, compared to similar real variability-rich systems that are of over 100K LoC and thousands of potential *vp*-s with variants [? 78], JavaGeom has over 35K LoC, being the size of medium-size variability-rich systems.

⁸removed: created

⁹removed: JavaGeomFM

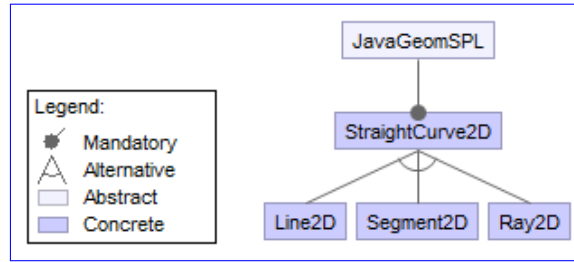


Figure 1: An excerpt of the feature model (FM) of JavaGeom variability-rich system (The FM is created using the FeatureIDE tool [51])

```

1  /* Class level variation point, vp_AbsLine2D */
2  public abstract class AbstractLine2D extends
    AbstractSmoothCurve2D implements
    SmoothOrientedCurve2D, LinearElement2D {
3
4      /* Method level variation point,
       vp_distance */
5
6      /* First variant of vp_distance */
7      public double distance(Point2D p) {
8          return distance(p.x(), p.y());
9      }
10
11     /* Second variant of vp_distance */
12     public double distance(double x, double y)
13     {
14         Point2D proj = projectedPoint(x, y);
15         if (contains(proj))
16             return proj.distance(x, y);
17         /* Omitted code */
18         return dist;
19     }
20     /* Omitted code */

```

```

21  /* First variant, v_Line2D, of vp_AbsLine2D */
22  public class StraightLine2D extends
    AbstractLine2D implements SmoothContour2D,
    Cloneable, CircleLine2D {
23      /* Omitted code */
24  }

```

```

25  /* Second variant, v_Segment2D, of vp_AbsLine2D
    */
26  public class LineSegment2D extends
    AbstractLine2D implements Cloneable,
    CirculinearElement2D {
27      /* Omitted code */
28  }

```

```

29  /* Third variant, v_Ray2D, of vp_AbsLine2D */
30  public class Ray2D extends AbstractLine2D
    implements Cloneable {
31      /* Omitted code */
32  }

```

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

implementation, it is used to represent conceptually the JavaGeom [..¹⁰] **product line**. 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. [..¹¹] 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 [..¹²] **variability implemented** by method overloading, *distance()* in lines 4 - 18, is not modeled in the FM. This single example, where only the techniques of inheritance and overloading are used, shows some difficulties in mapping features at the [..¹³] **reusable code assets**. 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 [42].

To overcome such variability management difficulties, existing approaches use annotations, often preprocessor directives [49], or have proposed to put into separate modules, for example, into aspects, feature modules, or delta

¹⁰removed: variability-rich system

¹¹removed: First, the

¹²removed: implemented variability

¹³removed: code assets of a given variability-rich system.

modules [6], all lines of code that belong to each specific domain feature [7, 71]. In addition, different tools are proposed, for example, pure::variants [73], code tagging [34], or those that propose to use colors, at the representation layer of code, for distinguishing the lines of code that implement each feature [40]. Analysing which technique, tool, or combination of them [56] 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. But, except for the tool-based approaches, most of the proposed solutions affect the initial decomposition or design of reusable code assets, such as their object-oriented design [7], and require a massive code refactoring[..¹⁴]. 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 [..¹⁵] reusable code assets and manage their variability in terms of *variation points* with *variants* (their definition is given in Section 2), as initial variability abstractions [36]. 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[..¹⁶] [45, 5, 13, 14, 63, 68], especially not when a subset of variability implementation techniques are used in combination [36, 23].

Towards an approach that meets [..¹⁷] our assumption and goal on managing variability implementations in code assets[..¹⁸], we proposed, in a previous work [77], 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. [..¹⁹]

The framework is devised by analysing up to ten common variability implementation techniques and by studying the related literature. Its main contributions are as follows.

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 [..²⁰] detail the concept of *imperfectly modular variability* further 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 [..²¹] and related illustrations (Section 4), mainly by using a running [..²²] case study introduced in Figure 1 and Listing 1.
- We present the implementation of the framework as a small [..²³] [..²⁴] 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 [..²⁵] case studies (Section 6). This

¹⁴removed: in case that the variability of a variability-rich system is managed by migrating it to an SPL approach.

¹⁵removed: code assets of a variability-rich system, that is, without the need to migrate it to an SPL, and manage its

¹⁶removed: of a variability-rich system

¹⁷removed: our

¹⁸removed: of a variability-rich system,

¹⁹removed: 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 [76] (cf. Section 2.2.3). This catalog is at first based on 7 previous frameworks, taxonomies, and catalogs [6, 29, 57, 60, 62, 61, 23]. For the purpose of our framework, we analysed the characteristics properties of these techniques, such as binding time.

²⁰removed: further

²¹removed: ,

²²removed: variability-rich system

²³removed: textual

²⁴removed: domain-specific

²⁵removed: subject systems

evaluation was conducted around four research questions to study the encountered implementation techniques, types of *vp*-s, [..²⁶] 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 [75] and showing how the [..²⁷] framework enables to integrate such [..²⁸] checking method in one of the [..²⁹] case studies (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

[..³⁰] In an SPL, the key aspect of variability management is the ability to trace the specified variability of software systems to its realization in reusable code assets.

2.1. Variability modeling

Since its first presentation by Kang et al. [38], 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 [26]. A feature model (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 [..³¹] product line. Its main feature, [..³²] JavaGeomSPL, 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 [..³³].

2.2. Variability realization

[..³⁴]

2.2.1. Reusable code assets

[..³⁵] In realistic SPL settings, despite the programming paradigm (*e.g.*, object-oriented, or functional), the implementation of reusable code assets complies, mostly, to a commonality and variability approach [23]. Specifically, a domain is decomposed into subdomains [..³⁶], then within each subdomain [..³⁷] the commonality is factorized from the variability. Thus, the reusable 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 [80], 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 [23]), 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

²⁶removed: the

²⁷removed: *tooled* framework enables us

²⁸removed: a

²⁹removed: subject systems

³⁰removed: 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.

³¹removed: variability-rich system

³²removed: JavaGeomFM

³³removed: or within a variability-rich system.

³⁴removed: We provide a background on the variability of code assets, the variability modeling, and the used techniques for its realization in variability-rich systems.

³⁵removed: In real variability-rich systems,

³⁶removed: then,

³⁷removed: ,

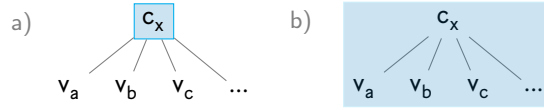


Figure 2: The variation point (vp) concept as: **a)** a *variable point* (i.e., the c_x), **b)** a *variable part* (i.e., the c_x with variants). The c_x is the common part for variants v_a , v_b , and v_c .

core with the commonalities and variabilities of all subdomains constitute the wholeness of reusable code assets in [\[..³⁸\]](#)an SPL.

2.2.2. Commonality and variability abstractions

While features are commonly used [\[..³⁹\]](#)as problem-oriented abstractions, the commonalities and variabilities in reusable code assets are usually abstracted in terms of variation points (vp-s) with variants, respectively, as solution-oriented abstractions [25, 26, 64]. Moreover, unlike features that are mere names, vp-s with variants are related to concrete elements in reusable code assets. Originally, "a variation point identifies one or more locations at which the variation will occur" [36], 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 [74, 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) [36], (ii) as a variable part (cf. Figure 2b) [9], or (iii) as an abstraction of a variability implementation technique [20, 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 [\[..⁴⁰\]](#)SPL are known as variability implementation techniques [28, 76]. They came from several programming paradigms and are supported by different constructs in different programming languages, which in turn offer different properties for [\[..⁴¹\]](#)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. [72] 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 [60] evaluate 8 techniques with up to 6 properties, Patzke and Muthig [61] evaluate 1 technique based on 10 properties, Muthig and Patzke [57] evaluate 8 techniques with up to 8 properties, Gacek and Anastasopoulos [29] evaluate 7 techniques with up to 7 properties, Patzke et al. [62] 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 [76] 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

³⁸removed: a variability-rich system

³⁹removed: 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) [25, 26, 64].

⁴⁰removed: SPL or variability-rich system

⁴¹removed: features or

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 [29, 60, 76].
- **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 [66]. 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, 76].

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 ⁴²capture and model the different properties of *vp*-s with variants which are implemented by traditional techniques. This enables to aim at variability management in variability-rich systems implemented with such ⁴³techniques. 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 ⁴⁴an SPL domain.

2.3. Variability traceability

The CoEST ⁴⁵ 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" [21].

The concept of traceability is already used for different reasons in *single* software engineering, mostly to trace requirements. But, in ⁴⁶SPL 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 their modeling of dependencies and relationships separate from other development artifacts is needed [14]. This is well known as variability traceability, which is specific to SPL engineering [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 [33], trace links are basically *established* to meet a specific *usage* purpose and need to be *maintained* during ⁴⁷an SPL domain engineering process [63]. Moreover, variability trace links can be established for different reasons and by different stakeholders [22, 5]. For example, they can be used for (semi)automating different processes during the application engineering [63], 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 ⁴⁸code level, or whether the *alternative* relation between three features, Line2D, Segment2D, and Ray2D, is properly considered during their implementation in code.

⁴²removed: model the implemented variability of variability-rich systems by capturing and modeling different properties of *vp*-s with variants which are implemented by traditional techniques. Specifically, considering the scope of our study, we consider 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.

⁴³removed: traditional techniques.

⁴⁴removed: a variability-rich system

⁴⁵Center of Excellence for Software Traceability: <http://coest.org/>

⁴⁶removed: SPL and variability-intensive systems engineering

⁴⁷removed: the whole

⁴⁸removed: the

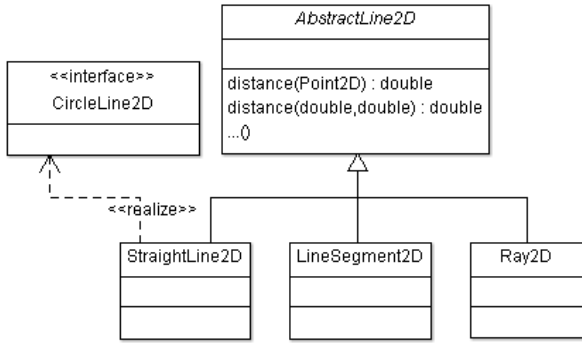


Figure 3: The detailed design of the given excerpt of JavaGeom variability-rich system in Listing 1

StraightCurve2D \mapsto *AbstractLine2D*
 Line2D \mapsto *StraightLine2D*
 Segment2D \mapsto *LineSegment2D*
 Ray2D \mapsto *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 \mapsto has the "implemented by" meaning

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 [..⁴⁹] in realistic SPL settings, 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 [..⁵⁰] 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 [..⁵¹] PPL 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 \mapsto 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 reusable 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 [..⁵²] an SPL, 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.

⁴⁹removed: in real variability-rich systems [78],

⁵⁰removed: us

⁵¹removed: variability-rich system

⁵²removed: the variability of a system,

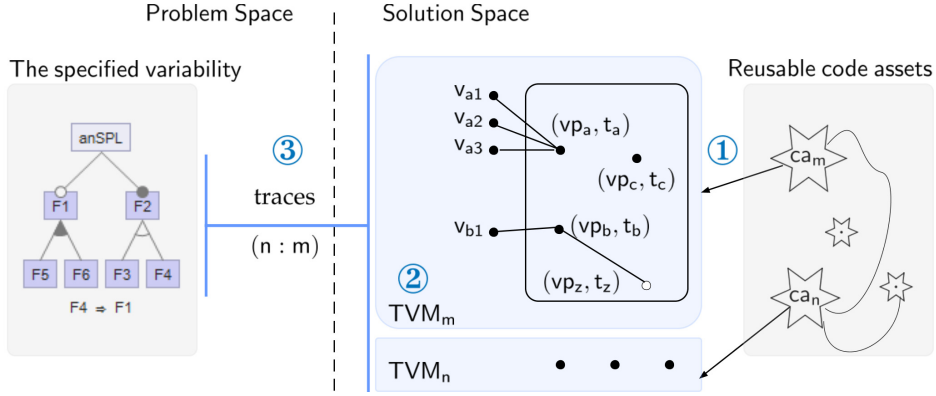


Figure 5: A three step framework for variability management of reusable code assets. TVM_m stands for technical variability model of the reusable code asset ca_m , with vp -s $\{vp_a, vp_b, \dots\}$ and their respective variants $\{v_{a1}, v_{a2}, \dots\}$ that are realized by different techniques $\{t_a,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

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^[.53]. Towards that, we determine the three following challenges.

Challenge₁: The diverse vp -s with variants of ^[.54] reusable code assets, realized by different traditional techniques, need to be captured and modeled in some way, in order to trace them to domain features.

Challenge₂: 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.

Challenge₃: The feasibility of an approach addressing *Challenge₁* and *Challenge₂* should be demonstrated by a tool support.

The *Challenge₁* is actually motivated by the *Challenge₂*. 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 ^[.55] an SPL, 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 *Challenge₁*, has a strong impact on the multiple usages of trace links, that is, in variability management (*Challenge₂*).

^[.56] In this article, we propose a toolled framework to capture the imperfectly modular variability at the implementation level, 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 *Challenge₁*, we propose a three step framework for managing the imperfectly modular variability of ^[.57] reusable code assets. It supports the following tasks, which are also depicted in Figure 5:

⁵³removed: of variability-rich systems.

⁵⁴removed: code assets of variability-rich systems

⁵⁵removed: a variability-rich system,

⁵⁶removed: In this article, we propose a toolled 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.

⁵⁷removed: code assets of a given variability-rich system

1. Capturing the imperfectly modular variability 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 reusable 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 [76]⁵⁸, which was briefly discussed in Section 2.2.3. We use this large set of summarized characteristic properties to understand what characterizes the variability in implementation⁵⁹, and what should be considered as abstractions so to model them properly in terms of variation points with variants.⁶⁰ Thus, we first⁶¹ analyse what characterizes only traditional techniques⁶² (cf. Section 2.2.3), which lead to the form of⁶³ imperfectly modular of variability. Then, ⁶⁴in addition to the established trace links of variability during the third step, we identified 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 the⁶⁵ logical relation between variants of a vp , their⁶⁶ binding time, evolution, and granularity. The third step, related to trace links, is based on some other existing frameworks on variability traceability [4, 5].

⁶⁷In the following, we give a detailed and formal representation of each of these three steps of the framework, which are ⁶⁸associated with illustrative examples.

4.1. Capturing the variability of reusable code assets

As previously discussed, ⁶⁹the variabilities in reusable code assets that ⁷⁰have a form of imperfect modularity usually ⁷¹consist of a common part ⁷²⁷³and a variable part. It may happen that a whole code asset is also a variable asset, such as for example a source file, a package, or a class⁷⁴⁷⁵. The variable part consists of ⁷⁶technique with a mechanism for ⁷⁷creating the variants, a way for resolving the variants, and the variants themselves (cf. Figure 2b). Toward addressing the first step of the framework, we naturally abstract them by using the concepts of variation points (vp -s) with variants. ⁷⁸The variable part 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 reusable code assets means to abstract the ⁷⁹place where the variability happens, namely ⁸⁰a variable asset element in a reusable code asset, or a whole variable code asset, and to represent ⁸¹it by a variation point (vp) or variant concept.

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

⁵⁸removed: . This study

⁵⁹removed: and how it can be captured and modeled

⁶⁰removed: Notably

⁶¹removed: analysed

⁶²removed: , being the focus of our study

⁶³removed: imperfect modularity

⁶⁴removed: we noticed

⁶⁵removed: logical relation

⁶⁶removed: binding time, evolution, and granularity. On the other hand, the third step regarding the trace of variabilities

⁶⁷removed: The second or third step begins only after the completion of its preceding step.

⁶⁸removed: illustrated by

⁶⁹removed: a variable part in

⁷⁰removed: has

⁷¹removed: consists

⁷²removed: with its variants (

⁷³removed: Figure 2b).

⁷⁴removed: , is a common part or varies (

⁷⁵removed: is a variant). In effect, the

⁷⁶removed: an implementation technique that denotes

⁷⁷removed: factorizing the commonality among related variants, for

⁷⁸removed: In general, it is expected that some targeted code assets of a system

⁷⁹removed: places

⁸⁰removed: the common code assets, or their common elements, with their varying code assets or their varying elements,

⁸¹removed: them by variation points (vp -s) with variants concepts, respectively

assume that the vp_x is implemented by a single traditional technique t_x . The set \mathbb{T} of possible techniques for vp_x [..⁸²] 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 [..⁸³] ⁸⁴.

A vp with variants is not a by-product of such [..⁸⁵] traditional technique [17, 69]. Therefore, to capture them properly, we need also to tag them in **reusable** 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 variable asset, or variable asset element, in reusable code assets.*

This mapping is needed to keep the association between the captured vp [..⁸⁶], or **variant**, concept and its respective **variable** 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 [..^b] tags are, (*StraightLine2D*, v_Line2D), (*LineSegment2D*, $v_Segment2D$), and (*Ray2D*, v_Ray2D).

^aIn the following, all illustrative examples will be given into similar blue boxes.

^bremoved: associations are:

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 [..⁸⁷] **reusable code assets** be:

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

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

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

The set CA of all [..⁸⁸] **reusable code assets**, with variability or [..⁸⁹] variable themselves, [..⁹⁰] is:

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

The [..⁹¹] **variable** elements in a **reusable** code asset, for example, for $ca_x \in CA$, which [..⁹²] **represent** some vp -s or their variants, [..⁹³] **are**:

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

According to Definition 3, the abstractions of each vp and variant from sets (1) and (2) are associated with their concrete implementation in **reusable** code assets, for example, with a varying file, class, or method, which we named

⁸²removed: with its variants

⁸³removed: in a given variability-rich system

⁸⁴Although 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).

⁸⁵removed: a

⁸⁶removed: or variant

⁸⁷removed: code assets of a targeted system

⁸⁸removed: code assets of the system

⁸⁹removed: being

⁹⁰removed: which have an association to some vp -s or their variants,

⁹¹removed: set of

⁹²removed: have an association to

⁹³removed: is

as in sets (3) and (4). Analysing some of the traditional variability implementation techniques, we ^[94] determine four possible associations for vp -s and ^[95] two for variants. They also represent all possible associations between a vp ^[96], or a variant and their respective ^[97] variable code assets, except for one association, which is explained in the following. 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 ^[98] a vp or variant ^[99] that abstracts a single location in reusable code assets. ^[100] The three possible cases are:

- (vp_x, ea_{xn}) , where n represents a specific element of the ca_x in (4). This is the case when vp_x represents only one element of an asset.

For instance, $(vp_AbsLine2D, AbstractLine2D)$, represented by a class.

- (vp_x, ca_x) , when the reusable code asset ca_x itself is variable.

If $vp_AbsLine2D$ is optional vp and together with its three variants is contained in a file, then the whole file is variable. Thus, $(vp_AbsLine2D, fileName)$.

- (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 variable asset of CA in (3) or a variable element of ca_x in (4). This is the case when the variant represents 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 reusable code assets.

Implicit. This kind of association represents the case when a vp ^[101] is implicit and cannot easily be tagged in reusable code assets. Therefore,

- (vp_x, \emptyset) , when the vp_x is implicit, hard to be tagged.

For example, this is the case when an *if-else* statement is used [72].

We do not include in the framework ^[102] the case when a ^[103] variant is unimplemented, (v_{xy}, \emptyset) , because, for example, an abstract class may have infinite subclasses as variants and abstracting them would be impossible and unnecessary.

⁹⁴removed: discerned five

⁹⁵removed: five

⁹⁶removed:

⁹⁷removed: code assets.

⁹⁸removed: the case when

⁹⁹removed: is tagged to

¹⁰⁰removed: The four

¹⁰¹removed: or variant

¹⁰²removed: two cases: (i)

¹⁰³removed: vp is introduced but has no variants [72], as we consider that it is part of the core assets [80], and (ii) the case when a

Table 1

Logical relations of variation points, and of variants in a variation point

Logical Relation	Description
Mandatory	The $[..^{109}]vp$ or variant is part of each software product
Optional	The $[..^{110}]vp$ or variant can be part of the software product or not
Multi Coexisting (Or)	One or more than one of the variants in a $[..^{111}]vp$ can be part of the software product
Alternative (Xor)	Only one of the alternative variants in a $[..^{112}]vp$ can be part of the software product
Mutual exclusion	When, during configuration, the selection of $[..^{113}]vp$ or variant requires the exclusion of another $[..^{114}]vp$ or variant, and vice versa
Requires	When the selection of a $[..^{115}]vp$ or variant requires the selection of another $[..^{116}]vp$ or variant

Spread. This kind of association represents the case when the same vp or variant is $[..^{104}]$ present in several locations in $[..^{105}]$ reusable code asset. Therefore,

- $(vp_x, \{ea_{x1}, ea_{x2}, \dots, ea_{xn}\})$, when the vp_x is found in more than one place in reusable code assets. The vp_x can be associated also with any code asset from the set CA in (3).
- $(v_{xy}, \{ea_{x1}, ea_{x2}, \dots, ea_{xn}\})$, when the variant is found in more than one place in reusable code assets. The v_{xy} can be associated also with any code asset from the set CA in (3).

$[..^a]$ For example, this is the case when $[..^b]$ parameterization is used as an implementation technique.

^aremoved: This

^bremoved: 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 overridden 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 [24]), or depending on their location in assets (e.g., kernel-param- vp , kernel-abstract- vp [31]). 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

$[..^{106}]$ 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 [76]. $[..^{107}]$ As part of our framework, we introduce here a formal definition of the four properties that need to be captured, $[..^{108}]$ as 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 $[..^{117}]$ variation point, vp_x , 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

¹⁰⁴removed: tagged to

¹⁰⁵removed: code assets

¹⁰⁶removed: In addition to the tagging properties, depending

¹⁰⁷removed: Therefore, as

¹⁰⁸removed: given that

¹¹⁷removed: vp_x variation point

Table 2Binding times of *vp*-s with variants (adapted from [16, 20])

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.

have dependencies [..¹¹⁸] [18], 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, $t_x \in \mathbb{T}$, 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, [..^a] in Listing 1 we capture the *alternative* logical relation between variants *v_Line2D*, *v_Segment2D*, and *v_Ray2D* [..^b] (cf. Section 4.1), which is realized by inheritance.

^aremoved: after the three

^bremoved: in Listing 1 are tagged, we capture their *alternative* logical relation,

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 [..¹¹⁹] a domain, 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, [..¹²⁰] *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) [..¹²¹] [15, 3].

At the level of a whole SPL 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 [20, Ch. 4], [47, 46].

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 [65]. 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. [..¹²²] 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 [..¹²³].

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.* [72]. As for the common kinds of binding times,

¹¹⁸removed: [19]

¹¹⁹removed: the code assets of a system

¹²⁰removed: different

¹²¹removed: [15, 3?]

¹²²removed: 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 [20, Ch. 4], [47, 46].

¹²³removed: and is bound independently

a taxonomy is available [20, 16][¹²⁴], which is also shown in Table 2.

Thus, the set of possible static and dynamic binding times (IBT) of a vp_x is:

$$IBT = \{\text{compilation, assembly, programming, configuration, deploy, startup, runtime, pure runtime}\}$$

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, [¹²⁵] a specific vp [¹²⁶] can evolve with new variants in the future [¹²⁷] or not. Thus, depending on the used technique, [¹²⁸] and on the constraints they create over the evolution of the variability they implement, vp -s can be *open* or *closed* to evolution.

Therefore, the evolution set (IEW) is:

$$IEW = \{\text{open, closed}\}$$

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 [^a] JavaGeom PL, the abstract class $vp_AbstractLine2D$ in Listing 1 is a vp that [^b] is captured as an *open* $vp_AbsLine2D$, meaning that its variants can evolve in the future.

^aremoved: the JavaGeom system

^bremoved: we capture

Granularity. A vp or variant in reusable code assets can have a specific granularity depending on the size of variability and the used technique [6, p.59][41]. 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 3¹²⁹. Thus, when traditional techniques are used, the following granularity set (GIR) contains the elements that actually show the granularity levels at which a variability implementation technique may be used to implement a vp or variant.

$$GIR = \{\text{file, package, interface, class, method, field, statement, generic type, argument, ...}\} \{\text{file, package, interface, class, method, ...}\}$$

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 reusable code assets

[¹³²] Each captured vp with its variants [¹³³] of reusable code assets are realized by a single technique, [¹³⁴] therefore they are somehow related.

¹²⁴removed: . While some approaches have a broader view on binding times, for instance, advocating for different binding times at runtime [16?], we use the seven most common static and dynamic binding times in our model, as

¹²⁵removed: new vp -s may emerge or

¹²⁶removed: may

¹²⁷removed: .

¹²⁸removed: the new or existing

¹²⁹A [¹³⁰] more complete table, [¹³¹] including the granularity of language constructs coming from the emerging techniques, such as feature modules as coarse-grained elements, is available elsewhere [76].

¹³²removed: Given that each

¹³³removed: is

¹³⁴removed: they are then

Table 3
Granularity of vp -s 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 the three variants, v_Line2D , $v_Segment2D$, and v_Ray2D , are related by inheritance with the $vp_AbsLine2D$, which technique offers an *alternative* ^a relation between variants, a *runtime* binding of variants, and *open* ^b evolution of the vp itself.

^aremoved: relationship

^bremoved: an

Therefore, we model their ¹³⁵ relations 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):

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

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

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

In this way, we aim to address the second step of our framework (cf. Figure 5) for modeling the implemented variability while keeping the ¹³⁶ consistency 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

¹³⁷ When modeling the implemented variability, we observed that the following five types of vp -s can be distinguished. ¹³⁸

$$\mathbb{X} = \{vp, vp_unimplemented, vp_technical, vp_nested, vp_optional\}$$

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). They are given in the set \mathbb{X} below.

$$\mathbb{X} = \{vp, vp_unimplemented, vp_technical, vp_nested, vp_optional\}$$

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 ¹³⁹ the variability

¹³⁵removed: relation

¹³⁶removed: association

¹³⁷removed: For

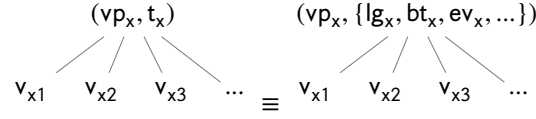
¹³⁸removed: They are given in the set \mathbb{X} below.

¹³⁹removed: this variability in code assets

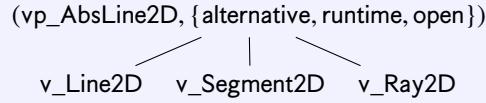
in a reusable code asset as a set of its variants and the characteristic properties derived from the vp_x 's implementation technique t_x . This leads to the following definition (cf. Equations (1) and (2)):

$$vp_x = \{V, t_x\} = \{\{v_{x1}, v_{x2}, v_{x3}, \dots, v_{xn}\}, t_x\}, \text{ where } n \in \mathbb{N} \quad (5)$$

We will represent it graphically as:

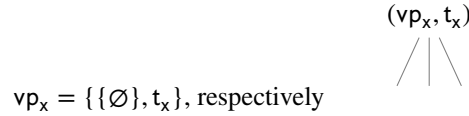


Such ordinary vp with its variants from Listing 1 is:



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:

$$vp_x = \{\{\emptyset\}, t_x\}$$



respectively,



It means that vp_x is introduced by the technique 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 vp_y is within a variant v_{x3} of an ordinary vp_x , it is modeled as below.

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

¹⁴⁰removed: ,

¹⁴¹removed: Based on this, we propose the following graphical representation of it

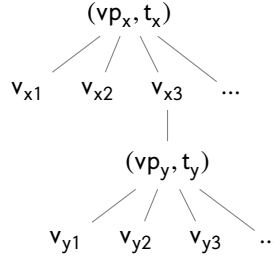
¹⁴²¹⁴³]

¹⁴³removed: :

¹⁴⁴removed: ,

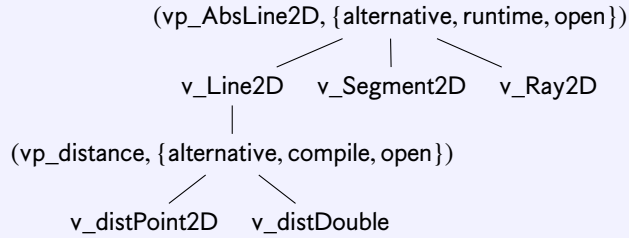
¹⁴⁵removed: ,

respectively,



For example, let us consider that `vp_distance` is the abstraction of the method level variability in Listing 1. It `[..a]` is a technical `vp` of `vp_AbsLine2D`, within the `[..b]varint` `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.

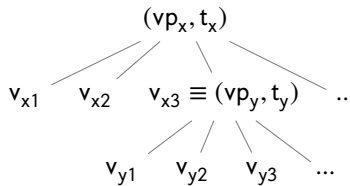
^aremoved: can be considered as
^bremoved: variant



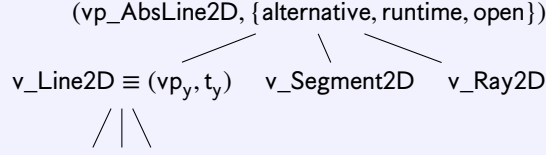
Nested. A nested variation point (*vp*) represents a variable part in a `reusable` code asset which becomes the common part for some other variants. On the contrary to a technical *vp*, when the vp_y is a nested variation point of vp_x then a variable part (e.g., the variant v_{x3}) becomes the common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

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

respectively,



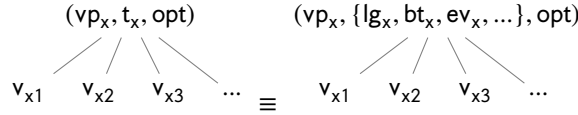
For example, v_Line2D can be a vp for its (un)implemented variants.



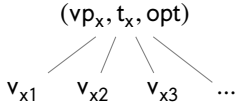
Optional. An optional variation point (vp) ^[..¹⁴⁶] is defined when the vp ^[..¹⁴⁷] itself is optional, but not its variants. However, when it is included or excluded in a ^[..¹⁴⁸] product, so are its variants. ^[..¹⁴⁹]

^[..¹⁵⁰] When vp_x is optional then ^[..¹⁵¹] the tuple (vp_x, t_x) ^[..¹⁵²] becomes a triple (vp_x, t_x, opt) . We use the acronym opt here in order to distinguish ^[..¹⁵³] between the optional variants in a vp and the optionality of the vp itself ^[..¹⁵⁴]. An optional vp can be ordinary, unimplemented, and can have other nested or technical vp -s. This is modeled as:

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



respectively,



^[..¹⁵⁵] Also, to make a vp as optional it may not require a technique ^[..¹⁵⁶] (i.e., simply the vp ^[..¹⁵⁷] is not included in ^[..¹⁵⁸] a final product). Therefore, we should document / model it.

4.2.2. Fragmented variability modeling

^[..¹⁵⁹] Since each vp ^[..¹⁶⁰] is implemented by a specific technique, it can be related or not with the other vp -s. In which case the whole modeled variability at the implementation level will have a forest-like structure ^[..¹⁶¹]

¹⁴⁶removed: represents a

¹⁴⁷removed: in code assets that together with its all variants is optional. This means that, whenever an optional vp

¹⁴⁸removed: software

¹⁴⁹removed: Moreover, variants may have any of the logical relations among themselves.

¹⁵⁰removed: In our framework, when

¹⁵¹removed: we represent the

¹⁵²removed: tuple as a triple, that is,

¹⁵³removed: the

¹⁵⁴removed: , which is different from the optional logical relation among variants in a vp

¹⁵⁵removed: To

¹⁵⁶removed: , that is,

¹⁵⁷removed: with its variants

¹⁵⁸removed: the final software product

¹⁵⁹removed: In addition to the logical relation among variants in a vp , different vp -s

¹⁶⁰removed: and their variants may also be related, for instance, the nested, technical, and/or ordinary vp -s with their variants. This implies that

¹⁶¹removed: , 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.

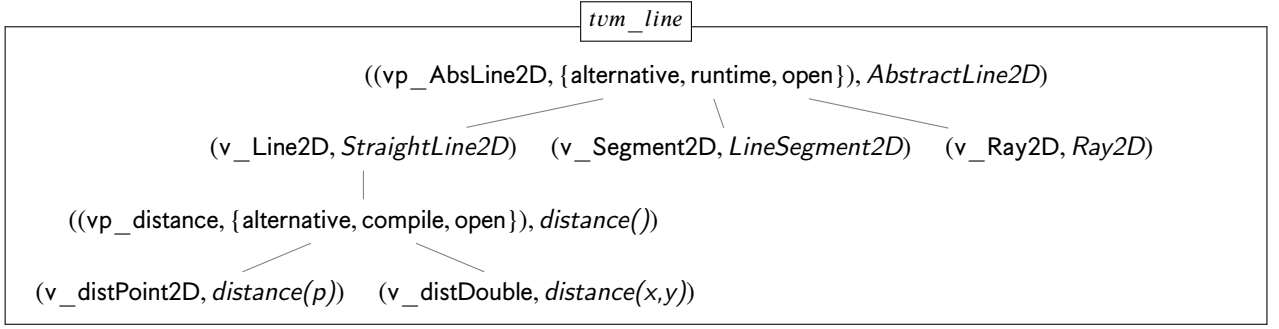


Figure 6: Documentation of the implemented variability in a TVM for the JavaGeom PL (cf. Listing 1). The italic names stand for tags of *vp*-s and variants to the variable asset elements

[..¹⁶²]. Therefore, instead of modeling the whole implemented variability at once and in one place, we propose to model it in a fragmented way. [..¹⁶³] What we consider as a fragment aims at being flexible. Specifically, it is a [..¹⁶⁴] reusable code asset that 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.

For this reason, we [..¹⁶⁵] designed special models, named as *technical variability models* (TVM). [..¹⁶⁶] They contain the abstractions of *vp*-s with variants, [..¹⁶⁷] their tags to the variable asset elements, and [..¹⁶⁸] describe the realized variability of specific reusable code assets. [..¹⁶⁹] Specifically, a TVM is the power set of all *vp*-s in [..¹⁷⁰] a reusable code asset. 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:

$$TVM_x = \text{Pow}(VP_x) = \{ \{ \{ v_{a1}, v_{a2}, \dots, v_{an} \}, t_a \}, \{ \{ v_{b1}, v_{b2}, \dots, v_{bn} \}, t_b \}, \dots, \{ \{ v_{z1}, v_{z2}, \dots, v_{zn} \}, t_z \} \} \quad (6)$$

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

As a result, the whole existing variability of code assets [..¹⁷¹] in [..¹⁷²] CA (in (3)) is modeled in technical variability models (TVMs) [..¹⁷³], which are created and maintained locally and closer to the code assets. It is important to note that, for simplicity, the tags of variants to the variable asset elements in the TVM_x in [..¹⁷⁴] (6) 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 [..¹⁷⁵] (3) and/or [..¹⁷⁶] (4).

¹⁶²removed: In our framework

¹⁶³removed: 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 [43].

¹⁶⁴removed: code asset with variability, which

¹⁶⁵removed: propose to model the variability of each fragment in a separate model

¹⁶⁶removed: It will

¹⁶⁷removed: tags to their respective code

¹⁶⁸removed: their characteristic properties within some given

¹⁶⁹removed: Then

¹⁷⁰removed: some given code assets

¹⁷¹removed: of a system, that is, of CA

¹⁷²removed: Equation (3),

¹⁷³removed: .

¹⁷⁴removed: Equation (6)

¹⁷⁵removed: Equation (3)

¹⁷⁶removed: 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 ^[..^a] PL. Similarly, the remained variability in JavaGeom PL can be modeled within the same or in other TVMs.

^aremoved: system

^[..¹⁷⁷] All these TVMs together constitute the *main technical variability model* (MTVM) ^[..¹⁷⁸]. Therefore, the MTVM ^[..¹⁷⁹] is defined as the set of all TVMs, given in the following.

$$\text{MTVM} = \{\text{TVM}_m, \text{TVM}_n, \text{TVM}_o, \dots, \text{TVM}_z\} \quad (7)$$

^[..¹⁸⁰] This MTVM contains the modeled variability of all code assets of ^[..¹⁸¹] an SPL. Unlike the organization of features in an FM as a tree structure, in MTVM the `vp`-s with variants ^[..¹⁸²] reside in a forest-like structure.

^[..¹⁸³] Moreover, the original meaning of a `vp` (given in Section 2.2.2 ^[..¹⁸⁴]) is extended:

Definition 4. A *variation point* is a location in ^[..¹⁸⁵] reusable code assets 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 [36], 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, 63]. The first ^[..¹⁸⁶] link is used in domain engineering for relating the specified variability at the domain level with the artifacts that realize it, such as in code level. ^[..¹⁸⁷] The trace link *use* relates an artifact during the application engineering process to its respective core assets that are developed during the domain engineering process [63]. 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 $f_x \in \text{FM}$ is a ^[..¹⁸⁸] variable feature at the specification level, then

$$\text{FM} = \{f_1, f_2, f_3, \dots, f_n\}, \text{ where } n \in \mathbb{N} \quad (8)$$

is the set of features in ^[..¹⁸⁹] the feature model (FM). ^[..¹⁹⁰] Whereas, a single compound feature f_x with its ^[..¹⁹¹] variant features and their logical relation (*i.e.*, *logic*) ^[..¹⁹²] in an illustrative way ^[..¹⁹³] is given as in the following. The *logic* can have any of the values from Table 1 in Section 4.1.2 ^[..¹⁹⁴] in the feature model.

¹⁷⁷removed: 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

¹⁷⁸removed: , which is specific to the given system. As part of our framework

¹⁷⁹removed: can be

¹⁸⁰removed: Thus, the

¹⁸¹removed: a given system. Moreover, unlike

¹⁸²removed: in the MTVM

¹⁸³removed: Based on these two steps of the framework, we noticed that

¹⁸⁴removed: can be further refined

¹⁸⁵removed: code assets of a system

¹⁸⁶removed: type of links

¹⁸⁷removed: Similar usage of this type of link can be found for example between application and platform features [?]. The *use* type of links

¹⁸⁸removed: domain feature of a given system

¹⁸⁹removed: its

¹⁹⁰removed: In a similar way with the illustration of a `vp` with variants during the framework's first two steps, a

¹⁹¹removed: varying

¹⁹²removed: can be given

¹⁹³removed: as follows

¹⁹⁴removed: .

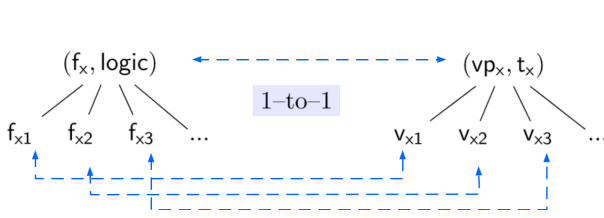


Figure 7: 1-to-1 mapping: ideal mapping

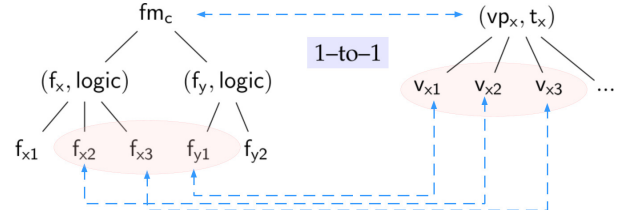
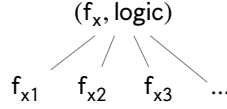


Figure 8: 1-to-1 mapping: a special case of this mapping



For mapping features [..¹⁹⁵] to vp -s and variants [..¹⁹⁶] in TVMs, we use a single bidirectional type of trace links implementedBy (\mapsto) or implements (\leftarrow), 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 [..¹⁹⁷] 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 [76], 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 [..¹⁹⁸] **variable unit** (i.e., **variable element in code assets**) that 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 [..¹⁹⁹] **variant** 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:

$$(f_x \mapsto vp_x), \text{ or implementedBy}(f_x, vp_x), \text{ respectively implements}(vp_x, f_x) \quad (9)$$

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

Similarly, based on the three other features illustrated in Figure 7, a [..²⁰⁰] **variant** feature f_{xn} [..²⁰¹] can be implemented by a single variant v_{xn} of vp_x , that is,

$$(f_{xn} \mapsto v_{xn}), \text{ or implementedBy}(f_{xn}, v_{xn}), \text{ respectively implements}(v_{xn}, f_{xn}) \quad (10)$$

¹⁹⁵removed: of an FM

¹⁹⁶removed: of TVMs

¹⁹⁷removed: expected to be

¹⁹⁸removed: code asset (element), which

¹⁹⁹removed: varying

²⁰⁰removed: varying

²⁰¹removed: is implemented ideally

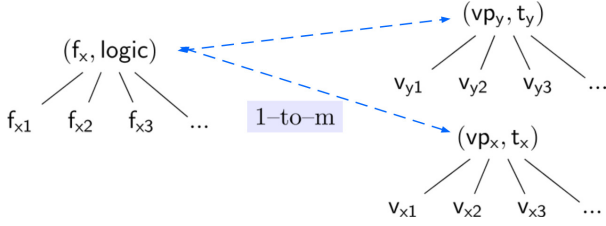


Figure 9: 1-to-m mapping: the mapping of a feature to several ordinary vp -s

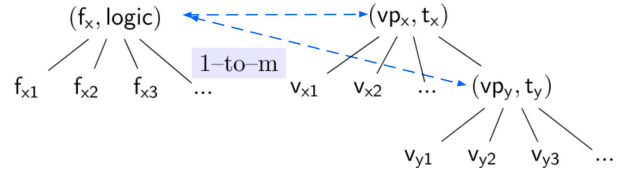


Figure 10: 1-to-m mapping: the mapping of a feature to several nested vp -s

For example, from Figures 1 and 3 in *JavaGeom PL*, ($\text{LineSegment2D} \mapsto \text{v_Segment2D}$). That is, the [\[..^a\]](#) feature *LineSegment2D* is implemented by the single [\[..^b\]](#) variant *v_Segment2D*, thus mapped to it.

^aremoved: *LineSegment2D* feature

^bremoved: *v_Segment2D* variant

[\[..²⁰²\]](#) The vp_x and v_{x_n} can be from different TVMs. Therefore, we may need to specify, for example, that $vp_x, v_{x_n} \in \text{TVM}_x$. Moreover, the vp -s with variants in a TVM can be from different code assets.

[\[..²⁰³\]](#) We distinguish a special case in the ideal mapping 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 [20], 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 [\[..²⁰⁴\]](#) 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 [\[..²⁰⁵\]](#) asset or not, then

$$(f_x \mapsto \{vp_x, vp_y, vp_z, \dots\}), \text{ or implementedBy } (f_x, \{vp_x, vp_y, vp_z, \dots\}) \quad (11)$$

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, ($\text{StraightCurve2D} \mapsto \{vp_AbsLine2D, vp_distance\}$). That is, the [\[..^a\]](#) feature *StraightCurve2D* is implemented by two variation points, *vp_AbsLine2D*, and *vp_distance*, thus mapped to them.

^aremoved: *StraightCurve2D* feature

n-to-1 Mapping. In this multiple mapping, a vp_x with its variants implement partially several variable 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 [\[..²⁰⁶\]](#)²⁰⁷

$$(\{f_x, f_y, f_z, \dots\} \mapsto vp_x), \text{ or implements } (vp_x, \{f_x, f_y, f_z, \dots\}) \quad (12)$$

[\[..²⁰⁸\]](#)

²⁰²removed: In general, the

²⁰³removed: In the ideal mapping we

²⁰⁴removed: by refactoring

²⁰⁵removed: assets

²⁰⁶removed:

²⁰⁷[\[..²⁰⁸\]](#)

²⁰⁸removed: 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.

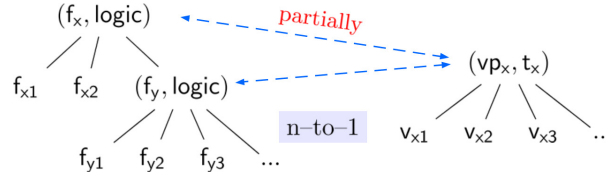


Figure 11: n-to-1 mapping: when several features are partially implemented by a single variation point

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 [79], 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 [209] PL we mostly used the same names for [210] 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. Challenge₁ in Section 3) for modeling and tracing the imperfectly modular variability of [211] reusable code assets.

5. Framework implementation

Aiming at a prototyped tool support to help in addressing Challenge₃ (cf. Section 3), we implemented the proposed framework as a textual *domain specific language* (DSL), a DSL being a small language that is narrowly focused on a particular problem domain [82, 83, 27]. We chose Scala to implement the framework as it is well-known for its first-class support for designing expressive DSLs [58, 30, Ch. 6]. In addition, the interoperability between Java and Scala enables one to use our DSL in almost all Java-based variability-rich systems.

In the following is provided the syntax of our *variability modeling DSL* (VM-DSL). It can be used to implement the framework as another internal DSL in other general-purpose languages, or as an external DSL. It consists of a set of language constructs for variability modeling and tracing, thus realizing the defined framework in Section 4. 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, 52, 75]. 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 detailed description and usage guidance are given at <https://ternava.github.io/vm-dsl/>, including detailed applications to four SPL case studies.

5.1. Variation points and variants support

In Listing 2 is shown the VM-DSL syntax for defining (i.e., capturing) vp -s with variants and modeling the

²⁰⁹removed: system

²¹⁰removed: vp -s, variants ,

²¹¹removed: a targeted system's

```

<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' | etc

<binding-time> ::= 'Compile' | 'StartUp' | 'Runtime' | etc

<evolution> ::= 'Open' | 'Close'

```

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

implemented variabilities into a technical variability model (TVM). [..²¹²][..²¹³][..²¹⁴][..²¹⁵]

[..²¹⁶]
 [..²¹⁷][..²¹⁸] The TVM consists of two parts:

- The captured abstractions of *vp*-s, $\langle vps-list \rangle$, with variants, $\langle variants-list \rangle$, that are associated with the variable elements in code assets, $\langle tag \rangle$.
- Then, the variabilities are modeled within a $\langle fragment \rangle$ module.

[..²¹⁹][..²²⁰][..²²¹]

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

²¹²removed: .Sin

²¹³removed: <trace-links> ::= 'traces' '

²¹⁴removed: ' <links> '

²¹⁵removed: '

²¹⁶removed: <links> ::= <vp-name> 'implements' <feature-name> | <variant-name> 'implements' <feature-name>

²¹⁷removed: <vp-name> | <variant-name> | <feature-name> := "string name"

²¹⁸removed: Syntax for defining trace links

²¹⁹removed: To address

²²⁰removed: (

²²¹removed: Section 3), we developed a prototyped tool to support the three steps of our framework. Different mapping studies [???] and surveys on industrial practices [? 34] show that, from the variability modeling toolled 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 toolled approaches are used. We thus propose the following textual variability modeling language in our context of code assets.

In the first part, for tagging the variabilities (i.e., $\langle tag \rangle$) the VM-DSL relies on the reflection capabilities in Scala.

[²²²..²²³..²²⁴..²²⁵] This helps to keep the relationship and strong consistency between the defined concepts of vp -s [²²⁶..²²⁷..²²⁸..²²⁹..²³⁰..²³¹] and variants (i.e., a $\langle vp\text{-name} \rangle$ [²²⁷..²²⁸..²²⁹..²³⁰..²³¹] and $\langle variant\text{-name} \rangle$) with their concrete implementation in reusable code assets. Then, the VM-DSL supports the five types of vp -s [²²⁸..²²⁹..²³⁰..²³¹..²³²..²³³..²³⁴..²³⁵] defined in our framework [²²⁹..²³⁰..²³¹..²³²..²³³..²³⁴..²³⁵]. Specifically, an ordinary vp is defined as VP , an optional vp is defined as oVP , a technical vp is defined as tVP , an unimplemented vp is defined as an ordinary vp (i.e., as VP) but without variants [²³¹..²³²..²³³..²³⁴..²³⁵].

[²³²..²³³..²³⁴..²³⁵] Whereas, in the second part [²³⁶..²³⁷..²³⁸..²³⁹..²⁴⁰..²⁴¹..²⁴²..²⁴³..²⁴⁴..²⁴⁵], variability is modeled within a **fragment** construct by specifying the different properties of a vp [²³⁷..²³⁸..²³⁹..²⁴⁰..²⁴¹..²⁴²..²⁴³..²⁴⁴..²⁴⁵] and its $\langle variants \rangle$, such as the $\langle logical\text{-relation} \rangle$, [²³⁸..²³⁹..²⁴⁰..²⁴¹..²⁴²..²⁴³..²⁴⁴..²⁴⁵] $\langle technique \rangle$, [²³⁹..²⁴⁰..²⁴¹..²⁴²..²⁴³..²⁴⁴..²⁴⁵] $\langle binding\text{-time} \rangle$, and $\langle evolution \rangle$ [²⁴⁰..²⁴¹..²⁴²..²⁴³..²⁴⁴..²⁴⁵].

Technically, we defined [²⁴¹..²⁴²..²⁴³..²⁴⁴..²⁴⁵] their values as extensions of their respective sealed classes [²⁴²..²⁴³..²⁴⁴..²⁴⁵] 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 [²⁴³..²⁴⁴..²⁴⁵] give the user the impression that methods are dynamically added to an existing class. We used it especially to implement the **fragment** [²⁴⁴..²⁴⁵..²⁴⁶..²⁴⁷..²⁴⁸..²⁴⁹..²⁵⁰..²⁵¹..²⁵²..²⁵³..²⁵⁴..²⁵⁵] **module**, in order to express in a more natural language the modeling of variability, as in Listing 2. As a result, the language

²²²removed: 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 [82, 83, 27]. Technically, we chose the Scala language to implement the framework as it is well-known for its first-class support for designing expressive DSLs [58, 30, 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.

²²³removed: 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.

²²⁴removed: 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.

²²⁵removed: To meet this demand, the first part of syntax in Listing 2 reveals the VM-DSL support for abstracting and tagging, that is, capturing,

²²⁶removed: with their variants. Specifically, the $\langle vp \rangle$ 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

²²⁷removed: $\langle vp\text{-type} \rangle$, and $\langle tag \rangle$, respectively. While the $\langle vp\text{-name} \rangle$ can have any string values, the $\langle vp\text{-type} \rangle$ can have one of the four predefined types of a vp

²²⁸removed: given

²²⁹removed: (

²³⁰removed: Section 4.2.1), namely VP , oVP , tVP , or nVP . It should be noted that

²³¹removed: $\langle tag \rangle$ is set to the path of the respective code assets, which can be a $\langle class\text{-name} \rangle$, a $\langle method\text{-name} \rangle$, or a $\langle field\text{-name} \rangle$. Similarly, the $\langle variant \rangle$ 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 $\langle variant\text{-name} \rangle$, using **Variant** construct, and setting the value for $\langle tag \rangle$, respectively.

²³²removed: Technically, all captured vp -s

²³³removed: and variants are kept into their list structures, $\langle vps\text{-list} \rangle$ and $\langle variants\text{-list} \rangle$, respectively. To set the $\langle tag \rangle$ 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 $\langle vp\text{-name} \rangle$ or a $\langle variant\text{-name} \rangle$, as well as their concrete implementation in code assets.

²³⁴removed: 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.

²³⁵removed: To meet this demand

²³⁶removed: of Listing 2 is given the language syntax for modeling the variability within the **fragment** construct where the four characteristic

²³⁷removed: and its variants are specified, namely the $\langle technique \rangle$,

²³⁸removed: $\langle binding\text{-time} \rangle$, and $\langle evolution \rangle$. The **fragment** construct can contain more than one vp with its variants and requires a $\langle tv\text{-name} \rangle$.

The values for the

²³⁹removed: $\langle logical\text{-relation} \rangle$,

²⁴⁰removed: 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 $\langle tv\text{-name} \rangle$ of the **fragment** construct are designed to distinguish different variability models within a given system.

²⁴¹removed: the $\langle technique \rangle$, $\langle logical\text{-relation} \rangle$, $\langle binding\text{-time} \rangle$, and $\langle evolution \rangle$

²⁴²A sealed class cannot have any new subclasses added except the ones that are defined in the same file [58]

²⁴³removed: gives

²⁴⁴removed: construct

```

<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

supports fragmented modeling as described in the previous section, meaning that several ^[245] ^[246] TVMs can be defined ^[247].

^[248]

5.2. Tracing support

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. ^[249] 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 ^[250] `<feature-name>` ^[251] can be mapped to several `vp`-s ^[252] `<vp-name>` or `variants` `<variant-name>`. ^[253] Obviously, 1-to-1 mapping ^[254] is also supported. 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 ^[255] mainly considers `vp`-s with variants at the implementation level ^[256] as a refinement of features from the domain level [35] ^[257], (ii) the amount of variability at the implementation level is expected to be larger than the variability at the domain level [63, Ch. 4]. This means that a feature ^[258] can 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` ^[259] are its value. Moreover, a key in the map structure can have several values.

^[260] ^[261]
^[262] ^[263] ^[264]

²⁴⁵removed: variability models (

²⁴⁶removed: technical variability models -TVMs)

²⁴⁷removed: for a given system

²⁴⁸removed: *One should be able to trace the modeled variability of a given system's code assets to its domain features.*

²⁴⁹removed: The language syntax is given in Listing 3.

²⁵⁰removed: , that is,

²⁵¹removed: ,

²⁵²removed: or variants, that is,

²⁵³removed: This type of trace link also includes the support for the

²⁵⁴removed: .

²⁵⁵removed: approaches mainly consider that

²⁵⁶removed: are

²⁵⁷removed: and

²⁵⁸removed: , in general, is expected to

²⁵⁹removed: is

²⁶⁰removed: 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.

²⁶¹removed: 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, 52, 75]. We will detail this usage of our framework in Section 7.

²⁶²removed: The prototype implementation of our DSL is publicly available at

²⁶³removed: . Its description and usage guidance are also given at

²⁶⁴removed: , including detailed applications to four variability-rich systems.


```

1  /* tvml_line.java */
2  package tvml // all TVMs are kept within a package
3
4  import dsl._
5  import scala.reflect.runtime.universe._
6  \PAR
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

5.3. VM-DSL usage example

Following the given syntax in Listings 2 and 3, Listings 4 and 5 show, respectively, ^[265] an usage example of the implemented VM-DSL. Specifically, Listing 4 ^[266] exhibits some modeling of the variability of code assets of ^[267] JavaGeom PL in Listing 1. ^[268] Listing 5 shows how to express the trace links ^[269] between the variability of code assets in ^[270] Listing 1 and features in Figure 1.

^[271] At first, the internal DSL has to be imported ^[272] 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 ^[273] 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). ^[274]

^[275] Whereas, in Listing 5 each *vp* and variant ^[276] 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 ^[277] SPL.

By providing the VM-DSL prototype as an implementation of the proposed framework, we consider that the *Challenge₃* is partially addressed. ^[278] In the following, we report ^[279] on the experimental evaluation of the proposed VM-DSL.

6. Applications

The provided framework is expected to be applicable to realistic ^[280] product line domains where traditional techniques are used. Therefore, to show its applicability, we decided to apply its tool support, the VM-DSL, ^[281] to

²⁶⁵removed: show a

²⁶⁶removed: shows the modeled

²⁶⁷removed: the JavaGeom system given

²⁶⁸removed: Then,

²⁶⁹removed: are established between the modeled

²⁷⁰removed: Listing 4 and domain

²⁷¹removed: From Listing 4, the VM-DSL

²⁷²removed: at first

²⁷³removed: ordinary

²⁷⁴removed: As mentioned in Demand ??, the relationship of 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.

²⁷⁵removed: From Listing 5, each captured

²⁷⁶removed: in Listing 4

²⁷⁷removed: system

²⁷⁸removed: To address the rest of it

²⁷⁹removed: in the following

²⁸⁰removed: variability-rich systems

²⁸¹removed: in several relevant variability-rich systems, as subject systems,

```

1  /* tracelinks.java */
2  import dsl.traces._
3  \PAR
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 defined *vp*-s with variants in Listing 4

model and trace [..²⁸²] the implemented variability in several relevant case studies.

6.1. Research questions

We organized the evaluation by first defining four main research questions [..²⁸³] that further address *Challenge*₃, on the applicability of our [..²⁸⁴] proposed framework (cf. Section 3). [..²⁸⁵] They are given below [..²⁸⁶], including their rationale.

RQ₁: What variability implementation techniques are encountered within [..²⁸⁷] an SPL that uses traditional techniques? This question is intended to answer [..²⁸⁸] whether several techniques are easily used in combination. To this end, we analyse how often each technique is used and [..²⁸⁹] what combination of techniques are often used within four SPL case study.

RQ₂: What are the encountered types of *vp*-s within [..²⁹⁰] an SPL? The purpose of this question is to highlight the types (*ordinary*, *technical*, *optional*, *nested*²⁹¹) of *vp*-s that are often found within [..²⁹²] an SPL, the diversity of the characteristic properties of these *vp*-s, and the number of TVMs needed to model the variability [..²⁹³] within an SPL. To this end, we [..²⁹⁴] analyse the modeled variability of reusable code assets of four [..²⁹⁵] case studies.

RQ₃: What is the encountered multiplicity of variability trace links within [..²⁹⁶] an SPL? 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 [..²⁹⁷] in one of the four [..²⁹⁸] case studies.

RQ₄: 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 [..²⁹⁹] an SPL and what is their report with the overall LoC of reusable code assets. To this

²⁸²removed: their implemented variability

²⁸³removed: , which purpose is to further address

²⁸⁴removed: tooled

²⁸⁵removed: The four research questions

²⁸⁶removed: .

²⁸⁷removed: a variability-rich system

²⁸⁸removed: the encountered combination of used variability implementation techniques in variability-rich systems

²⁸⁹removed: in what combination in four subject systems

²⁹⁰removed: a variability-rich system

²⁹¹As defined in Section 4.2.1

²⁹²removed: a variability-rich system

²⁹³removed: of a system

²⁹⁴removed: modeled and analysed the variability of

²⁹⁵removed: subject systems

²⁹⁶removed: a variability-rich system

²⁹⁷removed: between domain features and *vp*-s with variants at the code assets

²⁹⁸removed: subject systems

²⁹⁹removed: a variability-rich system

end, we define a measure that helps to evaluate this overhead, then we calculate it for four ³⁰⁰ case studies for observation.

6.2. Analysed subject systems

We expected that the application of our tool framework in different ³⁰¹ SPLs 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 ³⁰² case studies, but we could not build very large experiments. ³⁰³ We first selected three small ³⁰⁵ ³⁰⁶ case studies 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 ³⁰⁷ cases with a fourth ³⁰⁸ case study being a real feature-rich API ³⁰⁹ ³¹⁰, which broadens the scope of our validation. The list of ³¹¹ the fourth case study is as follows:

- **Graph PL** is a family of graph applications, which is proposed as a case study for evaluating the product line methodologies [50].
- **Arcade Game Maker PL** is a family of three arcade games: brickles, pong, and bowling. It is introduced by ³¹² the SEI ³¹³ as a case study for experimenting and learning the development of software product lines [2].
- **Microwave Oven PL** 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 [32].
- The fourth ³¹⁵ case study, **JavaGeom PL**, introduced in Section 1 is a large real open-source feature-rich API for creating geometric shapes in Java [48].

Before ³¹⁶ applying the VM-DSL, we had to enhance ³¹⁷ the four considered systems to get complete ones. ³¹⁸ For the first three case studies, we had the feature models at hand, and we implemented the SPL in Scala as the language supports the targeted ³²⁰ implementation techniques, and it was easier to apply the VM-DSL within the same development environment.

Obviously, implementing ourselves the ³²¹ SPLs that are going to use the VM-DSL is creating a bias on our experiments, especially on questions 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 ³²² feature-rich system would have done. In

³⁰⁰ removed: subject systems and make observations

³⁰¹ removed: variability-rich systems

³⁰² removed: subject systems

³⁰³ removed: For instance, we could not choose big systems using preprocessor directives, such as Linux, as they did not use object-oriented traditional techniques as we defined them 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 ³⁰⁴ and mostly those used in the SPL engineering community. From them, we

³⁰⁵ removed: subject systems (with less than 1K LoC,

³⁰⁶ removed: Table 4)

³⁰⁷ removed: subjects

³⁰⁸ removed: subject system

³⁰⁹ removed: (with over 35K LoC,

³¹⁰ removed: Table 4),

³¹¹ removed: four subject systems, with their LoC, used programming language, and the number of domain features, is summarized in Table 4.

Specifically,

³¹² removed: SEI

³¹³ <https://www.sei.cmu.edu/> ³¹⁴

³¹⁵ removed: subject system,

³¹⁶ removed: modeling the implemented variability of these subject systems, through

³¹⁷ removed: each of them

³¹⁸ removed: For the first three subject systems we had their feature models at hand. Therefore, we needed their implementations. Actually, different implementations of these systems exist. From those that we are aware of, for instance, Graph system is available in SPL2go ³¹⁹, but it is built as an SPL using FeatureIDE, while in our scope are variability-rich systems that are not organized as an SPL. Therefore, we implemented these subject systems using Scala, as it

³²⁰ removed: variability implementation techniques

³²¹ removed: variability-rich systems

³²² removed: variability-rich

Table 4

The implementation details of the four used subject systems

Subject system	Language	LoC	Files	Features	Implemented Features
Graph [50]	Scala	323	3	19	16
Arcade Game Maker [2]	Scala	651	17	27	26
Microwave Oven [32]	Scala	914	19	26	23
JavaGeom [48]	Java	35,456	142	941	941

particular, whenever it was possible, we used those techniques that were specified in the design of each given [\[..³²³\]](#) [case study](#). For instance, the Arcade Game Maker [PL](#) [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 [\[..³²⁴\]](#) [case studies](#), Graph [PL](#) [50] and Microwave Oven [PL](#) [32]. As for JavaGeom [PL](#), we had to analyse its domain [\[..³²⁵\]](#) 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 [\[..³²⁶\]](#) [case studies](#) can further be understood by their feature models, which are made available online ³²⁷.

[\[..³²⁸\]](#) Next, we used the VM-DSL to model the variability of [\[..³²⁹\]](#) [code assets in all these case studies in a forward approach](#). For the [\[..³³⁰\]](#) [three case studies we implemented, we model it](#) during their domain implementation or just after their whole implementation. For the JavaGeom [\[..³³¹\]](#) [PL case](#), 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 ³³². Finally, we analysed the modeled variability of all [\[..³³³\]](#) [case studies](#) 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 [\[..³³⁴\]](#) [research question \$RQ_1\$](#) [\[..³³⁵\]](#), we analysed the [\[..³³⁶\]](#) [variability modeled with the VM-DSL in each case study](#) and counted the occurrences of the variability implementation techniques. [\[..³³⁷\]](#) [Results](#) are shown in Figure 12. Specifically, in [\[..³³⁸\]](#) [Graph PL](#), 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 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 [\[..³³⁹\]](#) [PLs](#), 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

³²³removed: subject system

³²⁴removed: subjects, Graph

³²⁵removed: and build its feature model,

³²⁶removed: subject systems

³²⁷<https://github.com/terna/Variability-CChecking/wiki>

³²⁸removed: After we prepared the feature model and code assets for each subject system

³²⁹removed: their code assets

³³⁰removed: first three subject systems, we modeled their variability

³³¹removed: subject

³³²Details are available here: https://github.com/terna/Variability-CChecking/tree/master/src/SPL_Analysed_Examples

³³³removed: subject systems

³³⁴removed: the first research question(

³³⁵removed:)

³³⁶removed: resulted variability models from the VM-DSL usage in each subject system

³³⁷removed: The results

³³⁸removed: the Graph subject

³³⁹removed: subjects

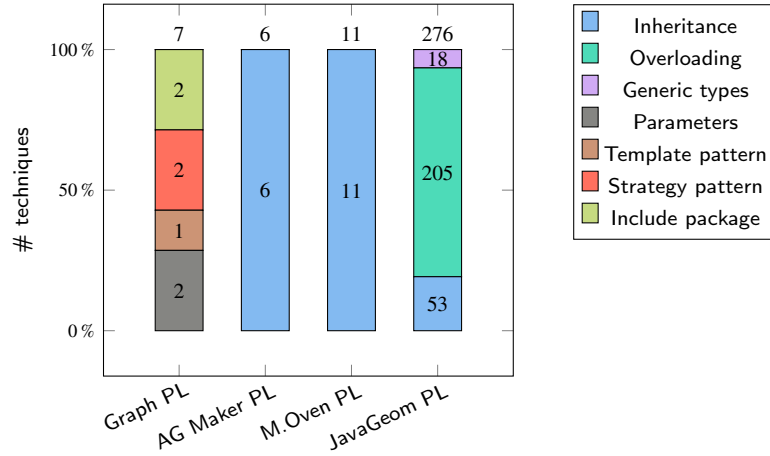


Figure 12: The used techniques, their number, and their combination, in each subject system

explained with their domain of variability and [..³⁴⁰]small size. In JavaGeom [..³⁴¹]PL 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 [..³⁴²]up to four techniques are used in combination. Then, within a [..³⁴³]case study a specific technique is used at least once within an SPL, such as inheritance in three case studies, Arcade Game Maker PL, Microwave Oven PL, and JavaGeom PL. In Arcade Game Maker PL it is used 6 times or 100%, in Microwave Oven PL it is used 11 times or 100%, whereas in JavaGeom PL it is used 53 times or 19%. In total, seven techniques are used in the four [..³⁴⁴]case studies. As a partial answer to the first research question RQ_1 , this shows that a combined set of different techniques [..³⁴⁵]are easily used to implement the variability [..³⁴⁶]within an SPL. 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* [78]. 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 [..³⁴⁷]case study we also analysed the [..³⁴⁸]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 [..³⁴⁹]Table 4.

6.4.1. Encountered characteristic properties of variation points

³⁴⁰removed: a few number of features.

³⁴¹removed: subject

³⁴²removed: of

³⁴³removed: subject system

³⁴⁴removed: subject systems

³⁴⁵removed: is

³⁴⁶removed: of a variability-rich system

³⁴⁷removed: subject system

³⁴⁸removed: modeled

³⁴⁹removed: Table 5

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

[.. ³⁵⁵]Case study	TVMs	<i>vp</i> -s	logical relation	binding time	evolution	granularity	variants
Graph PL	3	7	1 mandatory 6 optional	7 runtime	7 open 1 method	6 class	12
Arcade Game Maker PL	3	6	6 mandatory	6 runtime	6 open	6 class	17
Microwave Oven PL	8	11	11 mandatory	11 runtime	11 open	11 class	20
JavaGeom PL	11	276	276 mandatory	53 runtime 223 compile time	53 open 223 close	53 class 205 method 18 type	665

[..³⁵⁰]We show in Table 5 the resulting number of *vp*-s and variants in each considered [..³⁵¹]SPL. For instance, the Graph [..³⁵²]PL has 7 *vp*-s with 12 variants, the smallest number, whereas the JavaGeom [..³⁵³]PL 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 [..³⁵⁴]an SPL.

This table also gives details of the characteristic properties for the modeled *vp*-s within each [..³⁵⁶]PL. From the resulting 300 *vp*-s in four [..³⁵⁷]case studies (cf. Table 5), 294 *vp*-s are modeled as mandatory, [..³⁵⁸]and the remained 6 *vp*-s [..³⁵⁹]as optional, all being found in the Graph [..³⁶⁰]PL. 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 PL and Arcade Game Maker PL, as the implementation of *vp*-s crosscuts files. Regarding the JavaGeom PL, 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 [..³⁶¹][..³⁶²]are modeled as follows:

- 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 PL 41% of the medium-grained variabilities happen at the *constructor level* of a class, the rest is at the *method level*. Then, in all [..³⁶³]case studies, the *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 PL have only 2 variants.

These data show that *vp*-s [..³⁶⁴]within an SPL, or among them, have indeed diverse characteristic properties. Specifically, even with a small set of [..³⁶⁵]case studies, 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).

³⁵⁰removed: In Table 5 are shown

³⁵¹removed: subject system

³⁵²removed: subject

³⁵³removed: subject

³⁵⁴removed: a variability-rich system

³⁵⁶removed: subject system

³⁵⁷removed: subject systems

³⁵⁸removed: 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,

³⁵⁹removed: are

³⁶⁰removed: subject.

³⁶¹removed: (

³⁶²removed: values in the column *logical relation* in Table 5)

³⁶³removed: subject systems,

³⁶⁴removed: of a given variability-rich system

³⁶⁵removed: subject systems

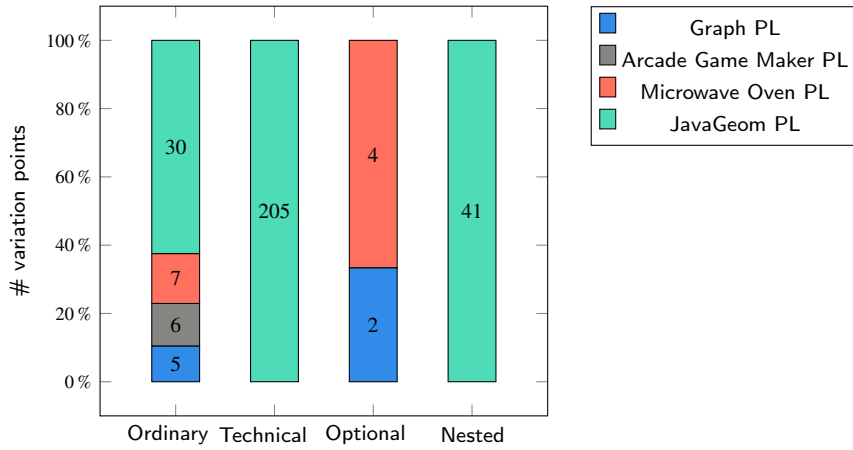


Figure 13: The used techniques, their number, and their combination, in four subject systems

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 [..³⁶⁶]case study. Thus, from Figure 12, the modeled variability in Graph [..³⁶⁷]PL has 7 *vp*-s, in Arcade Game Maker PL has 6 *vp*-s, in Microwave Oven PL has 11 *vp*-s, and in JavaGeom PL 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 PL, more than one type of *vp*-s exist in each [..³⁶⁸]case study. For instance, in JavaGeom PL 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 [..³⁶⁹]case study of JavaGeom PL, 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 [..³⁷⁰] *vp* are needed to model the variability of code assets in our [..³⁷¹]case studies. The presence of *technical* and *nested* *vp*-s indicate that *vp*-s are [..³⁷²]refinement of domain features, a characteristic that matters for establishing the variability trace links.

6.4.3. Number of TVMs

In general, we [..³⁷³]used more than one TVM from our framework to model the [..³⁷⁴]variability in each product line. Table 5 shows the resulted number of TVMs with their *vp*-s and variants. For example, we modeled the variability of the Graph [..³⁷⁵]PL in 3 TVMs, which contain in total 7 *vp*-s with 12 variants, whereas JavaGeom [..³⁷⁶]PL took 11 TVMs containing 276 *vp*-s with 665 variants.

It is important to mention that all TVMs in JavaGeom PL are at the package level, as we intended to minimize

³⁶⁶removed: subject system

³⁶⁷removed: subject

³⁶⁸removed: subject system

³⁶⁹removed: subject system of JavaGeom

³⁷⁰removed: *vp*-s

³⁷¹removed: subject systems

³⁷²removed: the

³⁷³removed: set up

³⁷⁴removed: entire variability of each subject system

³⁷⁵removed: subject

³⁷⁶removed: subject

the dependencies of *vp*-s across different TVMs, whereas in three other [..³⁷⁷] case studies they are at the file level. Although the granularity of TVMs seems to be related [..³⁷⁸] with the size of the [..³⁷⁹] product line, 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 [..³⁸⁰] file level for the Microwave Oven [..³⁸¹] PL.

Technically, we stored all modeled variability of a [..³⁸²] PL 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 [..³⁸³] an SPL into a single model. Still, from the beginning, someone may choose to model all variability of [..³⁸⁴] some given code assets 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 [..³⁸⁵] an SPL. This fragmented modeling has been shown to have notable advantages, for example, on variability consistency checking [75].

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 [..³⁸⁶] a given PL.

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 [..³⁸⁷] case study of the Microwave Oven PL. We decided for it as, in comparison to JavaGeom PL, we had knowledge for each of its *vp*-s with variants. Then, in comparison to Graph PL and Arcade Game Maker PL, it has the largest number of *vp*-s with variants (cf. Table 5).

After we prepared features of Microwave Oven PL into a textual format ³⁸⁸, 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 ³⁸⁹ showed that 17 *vp*-s and variants of Microwave Oven PL (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 PL 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 [..³⁹⁰] *vp_WeightSensor* and each of its alternative features is implemented by a 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_LightOff*. 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 [..³⁹¹] case study, which forms are also supported by the

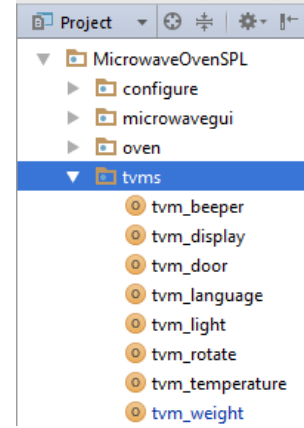


Figure 14: The eight TVMs into separate files for Microwave Oven PL

³⁷⁷removed: subject systems

³⁷⁸removed: to

³⁷⁹removed: variability-rich system

³⁸⁰removed: the

³⁸¹removed: subject system

³⁸²removed: variability-rich system

³⁸³removed: a variability-rich system

³⁸⁴removed: a given variability-rich system

³⁸⁵removed: a variability-rich system

³⁸⁶removed: code assets of a given variability-rich system

³⁸⁷removed: Microwave Oven subject system

³⁸⁸its 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

³⁸⁹The resulted mapping is available here: https://github.com/ternava/Variability-CChecking/blob/master/src/SPL_Analysed_Examples/MicrowaveOvenSPL/FeatureVPtraces.scala.

³⁹⁰removed: , *vp_WeightSensor* ,

³⁹¹removed: subject system

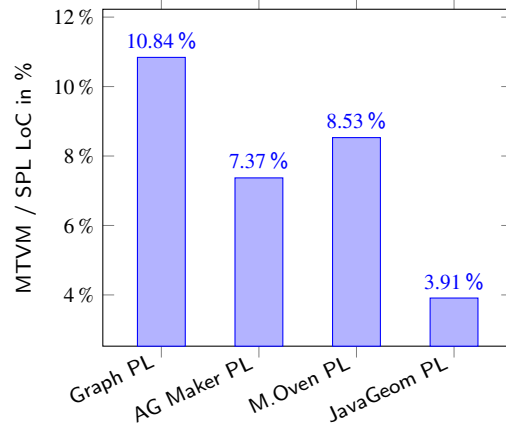


Figure 15: The MTVM's LoC to system's LoC ratio in four subject systems

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 ³⁹²product line, 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 ³⁹³an SPL. Therefore, we expect to answer the last research question (RQ_4) ³⁹⁴by studying the MTVM to ³⁹⁵SPL ratio, defined below.

Definition 5. The MTVM to SPL 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 ³⁹⁶an SPL.

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 ³⁹⁷and 13).

For each ³⁹⁸SPL, the calculated MTVM to ³⁹⁹SPL ratio is given in Figure 15. It shows that on average 7.66% of LoC are used to model their variabilities ⁴⁰⁰in TVMs and the rest ⁴⁰¹are used for realizing their respective ⁴⁰²SPL functionality. It can be observed that, for instance, Graph PL has the fewest lines of code (cf. Table 4, 323 LoC) but needs around 11% LoC for modeling ⁴⁰³their variability. Then, Microwave Oven ⁴⁰⁴PL has more lines of code (914 LoC) than Arcade Game Maker PL (651 LoC) ⁴⁰⁵, but it uses more LoC for variability modeling (8.53% compared to 7.37%, respectively).

³⁹²removed: variability-rich system, with few features and LoC,

³⁹³removed: a variability-rich system

³⁹⁴removed: , we studied the

³⁹⁵removed: System

³⁹⁶removed: a variability-rich system

³⁹⁷removed: ,

³⁹⁸removed: subject system

³⁹⁹removed: System

⁴⁰⁰removed: into

⁴⁰¹removed: is

⁴⁰²removed: system's functionalities

⁴⁰³removed: its

⁴⁰⁴removed: that

⁴⁰⁵removed: also

This indicates that the overhead for using the VM-DSL, calculated by the MTVM to ^[406]SPL ratio, does not depend directly on the lines of functional code ^[407]within an SPL. It actually is proportional to the number of *vp*-s, variants, and TVMs, thus answering the last research question (*RQ₄*). 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 reusable code assets and to produce their LoC.

6.7. VM-DSL limitations

As a means to further address *Challenge₃*, in this section we showed how our proposed VM-DSL addresses the four research questions (cf. ^[408]^[409]Section 6.1) by applying it in four ^[410]case studies. 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 ^[411]SPL. 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 ^[412]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 ^[413]reflection is not mandatory. Variability could also be captured by using a form of annotations [34]. 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, ^[414]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* [78]. 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 *Challenge₂* (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 [75], which was only focusing on consistency checking, and put it into perspective to show how its requirements are met by our ^[415]framework.

7.1. Requirements for consistency checking

Within an SPL, it is a well-known desirable property to check whether the specified ^[416]variabilities (e.g., ^[417]a feature model as in Figure 1) and implemented ^[418]variabilities (e.g., ^[419]in reusable code assets as in Listing 1) are consistent [52]. 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.

⁴⁰⁶removed: System

⁴⁰⁷removed: of a variability-rich system

⁴⁰⁸removed: -

⁴⁰⁹removed: in

⁴¹⁰removed: subject systems

⁴¹¹removed: variability-rich system

⁴¹²removed: reflection

⁴¹³removed: reflection

⁴¹⁴removed: reflection

⁴¹⁵removed: tooled

⁴¹⁶removed: variability

⁴¹⁷removed: the

⁴¹⁸removed: variability

⁴¹⁹removed: the variability model of

Definition 6. [Consistency rule] Within an SPL^[..⁴²⁰], 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.

Reqmt₁: One should be able to check the consistency of ^[..⁴²¹]variability when the mapping between features and *vp*-s with variants is 1-to-n, including ^[..⁴²²]an 1-to-1 mapping.

Reqmt₂: 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.

Reqmt₃: 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

^[..⁴²³]

The proposed consistency checking method takes as input a feature model (FM), representing the domain variability, as well as the trace links (TLs), and a TVM or ^[..⁴²⁴]subset of them, ^[..⁴²⁵]taken from our 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 logic⁴²⁶.

- **Slicing.** During this step, a single TVM, or a subset of them from the MTVM ^[..⁴²⁷]in an SPL, 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.
- **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.

⁴²⁰removed: or variability-rich system

⁴²¹removed: variabilities

⁴²²removed: the

⁴²³removed: 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 [?], harder to check, but also harder to trace and fix inconsistencies after all of them are shown at once [?]. 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 [?]. To overcome these problems, we propose a *consistency checking method* for detecting the variability inconsistencies earlier during the development process.

⁴²⁴removed: a

⁴²⁵removed: created with our tool

⁴²⁶In the following, the considered feature models are only propositional ones.

⁴²⁷removed: of a variability-rich system

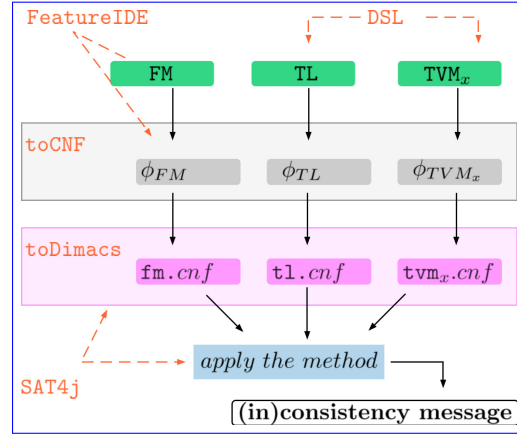


Figure 16: Prototype toolchain for consistency checking. Inputs: feature model (FM), trace links (TL), and technical variability model (TVM_x). Their respective propositional formula: ϕ_{FM} , ϕ_{TL} , and ϕ_{TVM_x} . Their respective DIMACS format: *fm.cnf*, *tl.cnf*, and *tvm_x.cnf*.

The consistency checking method has been implemented using the Scala language⁴²⁸. 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 ⁴²⁹TVMs (or a subset of them). While the FM is converted to a propositional formula using FeatureIDE [51], 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 [75].

The implemented method has also been applied to the Microwave Oven ⁴³⁰ ⁴³¹ ⁴³²PL case study. This PL 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 ⁴³³PL. 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 ⁴³⁴PL, which has 26 features and 720 configurations. It took around 0.13 *seconds*. As we 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 ⁴³⁵PL TVM(s) is 31, which indicates that

⁴²⁸Its implementation is publicly available at <https://github.com/ternava/variability-cchecking>.

⁴²⁹removed: TVM

⁴³⁰removed: 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 (

⁴³¹removed: Section 6.5), which is one of the requirements for our consistency checking method (

⁴³²removed:). Then, in comparison with the JavaGeom, we used its externally created feature model [32], 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

⁴³³removed: subject system

⁴³⁴removed: system

⁴³⁵removed: TVMs


```

tvm_language.scala x TVM_moven_toCNF.scala x
tvm_language
17 import module._
18 module("Language scala") {
19   Language is MUL with variants (English, French, Italian, Spanish, German) use
20   INHERITANCE with binding RUN_TIME and evolution OPEN
21 }
22 }

Run TVM_moven_toCNF
Nr of Configurations for the TVM(s) is: 31.0
1 10 13 -14 -15 -16 -17 0
1 10 -13 14 -15 -16 -17 0
1 10 -13 -14 15 -16 -17 0
1 10 -13 -14 -15 16 -17 0
1 10 -13 -14 -15 -16 17 0
Nr of Configurations for the sliced FM is: 5.0
1 10 13 -14 -15 -16 -17 101 102 -103 -104 -105 -106 0
1 10 -13 14 -15 -16 -17 101 -102 103 -104 -105 -106 0
1 10 -13 -14 -15 -16 17 101 -102 -103 104 -105 -106 0
1 10 -13 -14 -15 16 -17 101 -102 -103 -104 105 -106 0
1 10 -13 -14 15 -16 -17 101 -102 -103 -104 -105 106 0
Nr of Configurations for the sliced FM and TVM(s) is: 5.0
(Models are Inconsistent with each other!,5.0,31.0,5.0)
Process finished with exit code 0

```

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

the FM and the selected [..⁴³⁶]TVM(s) variabilities are inconsistent. [..⁴³⁷]

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 [..⁴³⁸]case study, 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 [..⁴³⁹]variability 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 PL 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. *Challenge₂* in Section 3). Further, the fulfilled requirements ($Reqmt_1$, $Reqmt_1$, and $Reqmt_3$) and the successful application to one of our [..⁴⁴⁰]case studies give us confidence in the good applicability of our framework and its [..⁴⁴¹]implemented VM-DSL, as a concrete solution for modeling and tracing the variability of reusable code assets.

⁴³⁶removed: TVMs

⁴³⁷removed: 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 [52], 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.

⁴³⁸removed: subject system

⁴³⁹removed: variabilities

⁴⁴⁰removed: subject systems

⁴⁴¹removed: implementation, the

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. Internal Validity

8.2. Construct Validity

A first threat to validity is related to ⁴⁴²the selected case studies. We used three academic ⁴⁴³case studies that we implemented ourselves. As we discussed in Section 6.2 ⁴⁴⁴this threatens the validity of the ⁴⁴⁵research questions that aim at characterizing the encountered implementation techniques. ⁴⁴⁶We tried to implement ⁴⁴⁷these three systems only by following good practices in object-oriented design⁴⁴⁸, 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 [29, 28, 60, 72, 76], which are themselves very common, such as inheritance and overloading.

8.2. Internal Validity

⁴⁴⁹
⁴⁵⁰A second threat is related to the ⁴⁵¹
⁴⁵²difficulty we faced during the application of the VM-DSL to ⁴⁵³the case studies, especially on recognizing whether all used techniques are variability related. For instance, it was sometimes hard to distinguish whether an inheritance hierarchy implements some ⁴⁵⁴product line variability (*i.e.*, which ⁴⁵⁵differentiates products in an SPL⁴⁵⁶) or some internal functionality that should not be modeled or traced. ⁴⁵⁷In our experiments, we decided to classify all used techniques as variability related techniques. However, in ⁴⁵⁸realistic SPL domains, this may not be always true ⁴⁵⁹[55]. For instance, there are approaches that consider that all C preprocessor directives in ⁴⁶⁰reusable code assets are used for implementing some PL variability, whereas some others disagree [84]. Specifically, Zhang et al. [84] 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 ⁴⁶¹[78, 55] 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 ⁴⁶²

⁴⁴²removed: *the experiment bias* regarding the selected subject systems

⁴⁴³removed: variability-rich systems as subject systems

⁴⁴⁴removed: ,

⁴⁴⁵removed: obtained results on

⁴⁴⁶removed: To reduce this bias, we

⁴⁴⁷removed: the variability of these systems

⁴⁴⁸removed: and, whenever was possible, by using the specified techniques in their original design, namely in [50, 2, 32]. But,

⁴⁴⁹removed: 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 [?]. 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.

⁴⁵⁰removed: 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

⁴⁵¹removed: *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.

⁴⁵²removed: Another threat is related to the

⁴⁵³removed: subject systems

⁴⁵⁴removed: (product line)

⁴⁵⁵removed: has in purpose to differentiate

⁴⁵⁶removed: or variability-rich system

⁴⁵⁷removed: Because of this *deficiency* of information during

⁴⁵⁸removed: real systems

⁴⁵⁹removed: [?]

⁴⁶⁰removed: a system

⁴⁶¹removed: [78?]

⁴⁶²removed: [?]

[55].

8.2. Conclusion Validity

[..⁴⁶³]

8.2. 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 [..⁴⁶⁴]studied cases do not enable us to conclude on a large applicability of the proposed framework and implementation.

On the covered techniques in the [..⁴⁶⁵]case studies, their number [..⁴⁶⁶]with the three academic [..⁴⁶⁷]case studies were four, but [..⁴⁶⁸]with the fourth real [..⁴⁶⁹]case study, JavaGeom, - a realistic feature-rich system, the total number of used techniques became seven. It cannot be determined whether more real [..⁴⁷⁰]case studies would enable the validation over a larger set of techniques. Nevertheless, as we discussed for the [..⁴⁷¹]internal 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 [78].

As for the framework usage, we only analysed one scenario with consistency checking, but we believe it is a very relevant one to also [..⁴⁷²]the feasibility of the usage. Obviously more extensions would bring a better validation on this aspect.

9. Related work

[..⁴⁷³]

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 [59], basic traceability model [54], a representation independent variability meta-model [68, 67], a conceptual model for traceability [14], or a framework for variability modeling and management in all abstraction levels [70, 20, 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.* [52], and we keep the variability information separated from the code assets as by Berg *et al.* [14] and Pohl *et al.* [63]. 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.

⁴⁶³removed: A threat to conclusion validity is the *lack of expert evaluation* of our tool 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 tool 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 tool framework in real systems and by their experts is also part of our future work.

⁴⁶⁴removed: variability-rich systems

⁴⁶⁵removed: subject systems

⁴⁶⁶removed: in

⁴⁶⁷removed: variability-rich systems

⁴⁶⁸removed: in

⁴⁶⁹removed: variability-rich system

⁴⁷⁰removed: systems

⁴⁷¹removed: construct validity

⁴⁷²removed: show

⁴⁷³removed: 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.

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 [37] 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.

[..⁴⁷⁴]

Variability modeling. [..⁴⁷⁵]As for the types of *vp-s* and *variants*, some other works exist[..⁴⁷⁶], for instance, by Jacobson *et al.* [36], Schmid *et al.* [68], Sinnema *et al.* [70, 69] and Capilla *et al.* [20]. [..⁴⁷⁷]Although they are not variability traceability approaches, we used their notion of *vp-s* and *variants* and several of their findings [..⁴⁷⁸]of the diversity of *vp-s*. Yet, we consider that the [..⁴⁷⁹]range of *vp-s* [..⁴⁸⁰]comes 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 [43] where each module defines its own local variability model. However, in this approach, a [..⁴⁸¹].c file is considered as a module with inner variability, which has only *compile time* binding. In [..⁴⁸²]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.

[..⁴⁸³]

Variability management tools. Furthermore, there are [..⁴⁸⁴]commercial tools for variability management, such as [..⁴⁸⁵]pure::variants [73][..⁴⁸⁶][..⁴⁸⁷]. 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 [..⁴⁸⁸]in the lines of code, we use a textual DSL to model *vp-s* with *variants*[..⁴⁸⁹], that is, the structure of code, through the capture of the used implementation technique inside a package, file, or class. [..⁴⁹⁰]

When preprocessors in C are used [49], 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

⁴⁷⁴removed: 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.

⁴⁷⁵removed: As for the variability modeling, some

⁴⁷⁶removed: regarding the types of *vp-s* and *variants*

⁴⁷⁷removed: We

⁴⁷⁸removed: regarding

⁴⁷⁹removed: types

⁴⁸⁰removed: come

⁴⁸¹removed: .c

⁴⁸²removed: the

⁴⁸³removed: When preprocessors in C are used [49], 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.

⁴⁸⁴removed: prototypes and

⁴⁸⁵removed: FeatureIDE [51],

⁴⁸⁶removed: , or domain-specific languages [?]. While FeatureIDE is a tool support for the orthogonal approaches to variability traceability (

⁴⁸⁷removed: ??),

⁴⁸⁸removed: to code assets of a variability-rich system

⁴⁸⁹removed: of code assets

⁴⁹⁰removed: Then,

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.

Finally, 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. Variability traceability approaches

Besides these variability management 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 [37] 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.

9.3. Orthogonal approaches to variability traceability

When traditional techniques are used for implementing the variability in an SPL^[..⁴⁹¹], the code is not shaped in terms of features [36, 17, 72, 20]. 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^[..⁴⁹²] which shape the reusable code assets in terms of features [6, 53, 66, 8], or the specified features hold a direct representation in code [34, 44], 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.

9.4. Textual variability modeling approaches

[..⁴⁹³]

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^[..⁴⁹⁴] 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.

⁴⁹¹removed: or variability-rich system

⁴⁹²removed: that shape the

⁴⁹³removed: 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 [?]. 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.

⁴⁹⁴removed: a

In this work, we provided a three step framework for managing this imperfectly modular variability of ⁴⁹⁵ reusable code assets. 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 ⁴⁹⁶ with their tags to specific code assets, ⁴⁹⁷ in technical variability models (TVMs), and (iii) establishing the n-to-m trace links between ⁴⁹⁸ 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 ⁴⁹⁹ within 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 ⁵⁰⁰ understanding the addressed variability in reusable code assets.

We implemented the framework as an internal ⁵⁰¹ / ⁵⁰² domain specific language (DSL) in Scala, and applied it to four different ⁵⁰³ SPL case studies. 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 ⁵⁰⁴ case studies, 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 ⁵⁰⁵ case study. While being on a small set of ⁵⁰⁶ case studies, these results show the feasibility of our framework and its potential application in realistic variability-rich systems with a small ⁵⁰⁷ 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 ⁵⁰⁸ a case study. 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* [19, 68], which can serve as a starting point for this extension. Further, we will evaluate the scalability of our ⁵⁰⁹ framework on a larger set of ⁵¹⁰ case studies, and with an application by other users. In particular, by applying it to a broader range of ⁵¹¹ case studies, 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 ⁵¹² variation points in similar kinds of

⁴⁹⁵ removed: code assets of a variability-rich system

⁴⁹⁶ removed: , with tags to their respective

⁴⁹⁷ removed: into

⁴⁹⁸ removed: domain

⁴⁹⁹ removed: into

⁵⁰⁰ removed: comprehending

⁵⁰¹ removed: textual

⁵⁰² removed: domain-specific

⁵⁰³ removed: subject systems being variability-rich systems

⁵⁰⁴ removed: subject systems

⁵⁰⁵ removed: variability-rich system.

⁵⁰⁶ removed: subject systems

⁵⁰⁷ removed: textual

⁵⁰⁸ removed: one of the subject systems

⁵⁰⁹ removed: tooled

⁵¹⁰ removed: variability-rich systems

⁵¹¹ removed: real variability-rich systems

⁵¹² removed: *vp-s*

code structure [..⁵¹³][78, 55] could also be considered to be coupled to the proposed contribution, so to reduce the burden of [..⁵¹⁴]variability capturing and modeling [..⁵¹⁵]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) [81], 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: [..⁵¹⁶]Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, IEEE Computer Society. pp. 424–427. [..⁵¹⁷][..⁵¹⁸]
- [2] AGM, 2009. Arcade game maker pedagogical product line. <https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=485941>. [Accessed 2019-01-05].
- [3] Alves, V., Schneider, D., Becker, M., Bencomo, N., Grace, P., 2009. Comparative 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: [..⁵¹⁹]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. [..⁵²⁰][..⁵²¹]
- [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: [..⁵²²]Software Engineering (ICSE), 2011 33rd International Conference on, IEEE. pp. 421–430. [..⁵²³][..⁵²⁴]
- [8] Apel, S., Leich, T., Saake, G., 2008. Aspectual feature modules. *Software Engineering, IEEE Transactions on* 34, 162–180. [..⁵²⁵][..⁵²⁶]
- [9] Bachmann, F., Clements, P.C., 2005. Variability in software product lines. Technical Report CMU/SEI-2005-TR-012. [..⁵²⁷]DTIC Document.
- [10] Bassett, P.G., 1996. Framing software reuse: lessons from the real world. Prentice-Hall, Inc.
- [11] Bayer, J., Widen, T., 2001. [..⁵²⁸][..⁵²⁹]Introducing traceability to product lines, in: International Workshop on Software Product-Family Engineering, Springer. pp. 409–416.
- [12] Becker, M., 2003a. Mapping variabilities onto product family assets, in: Proceedings of the International Colloquium of the Sonderforschungsbereich, [..⁵³⁰][..⁵³¹]Citeseer.
- [13] Becker, M., 2003b. Towards a general model of variability in product families, in: Workshop on Software Variability Management. Editors Jilles van Gurp and Jan Bosch. Groningen, The Netherlands. <http://www.cs.rug.nl/Research/SE/svm/proceedingsSVM2003Groningen.pdf>,

⁵¹³removed: [78?] or a recommendation approach based on our framework for modeling *vp-s* with variants, comparable to the code recommendation approaches for reuse [?] .

⁵¹⁴removed: manual

⁵¹⁵removed: and

⁵¹⁶removed: ,

⁵¹⁷removed: ,

⁵¹⁸removed: .

⁵¹⁹removed: ,

⁵²⁰removed: ,

⁵²¹removed: .

⁵²²removed: ,

⁵²³removed: ,

⁵²⁴removed: .

⁵²⁵removed: ,

⁵²⁶removed: .

⁵²⁷removed: Carnegie Mellon University Pittsburgh PA Software Engineering Institute

⁵²⁸removed: .

⁵²⁹removed: Master's thesis.

⁵³⁰removed: p.

⁵³¹removed: .

- pp. 19–27. [..⁵³²][..⁵³³][..⁵³⁴][..⁵³⁵][..⁵³⁶][..⁵³⁷][..⁵³⁸][..⁵³⁹][..⁵⁴⁰][..⁵⁴¹][..⁵⁴²][..⁵⁴³][..⁵⁴⁴][..⁵⁴⁵][..⁵⁴⁶][..⁵⁴⁷][..⁵⁴⁸][..⁵⁴⁹][..⁵⁵⁰]
- [14] Berg, K., Bishop, J., Muthig, D., 2005. Tracing software product line variability: from problem to solution space, in: [..⁵⁵¹]Proceedings of the 2005 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries, South African Institute for Computer Scientists and Information Technologists. pp. 182–191. [..⁵⁵²][..⁵⁵³][..⁵⁵⁴][..⁵⁵⁵][..⁵⁵⁶][..⁵⁵⁷][..⁵⁵⁸][..⁵⁵⁹][..⁵⁶⁰][..⁵⁶¹][..⁵⁶²][..⁵⁶³][..⁵⁶⁴][..⁵⁶⁵][..⁵⁶⁶][..⁵⁶⁷]
- [15] Bosch, J., 2000. Design and use of software architectures: adopting and evolving a product-line approach. Pearson Education.
- [16] Bosch, J., Capilla, R., 2012. Dynamic variability in software-intensive embedded system families. Computer 45, 28–35. URL: <http://dx.doi.org/10.1109/MC.2012.287>, doi:10.1109/MC.2012.287.
- [17] Bosch, J., Florijn, G., Greefhorst, D., Kuusela, J., Obbink, J.H., Pohl, K., 2001. Variability issues in software product lines, in: Software Product-Family Engineering. Springer, [..⁵⁶⁸]pp. 13–21.
- [18] Böhne, S., Halmans, G., Pohl, K., 2003a. [..⁵⁶⁹][..⁵⁷⁰]Modeling dependencies between variation points in use case diagrams, in: REFSQ, pp. 59–69.
- [19] Böhne, S., Halmans, G., Pohl, K., 2003b. Modelling dependencies between variation points in use case diagrams, in: REFSQ, Citeseer. pp. 59–69.
- [20] Capilla, R., Bosch, J., Kang, K.C., [..⁵⁷¹]2013. Systems and Software Variability Management. Springer.
- [21] Cleland-Huang, J., Gotel, O., Zisman, A., [..⁵⁷²]2012. [..⁵⁷³]Software and systems traceability. volume 2. Springer. [..⁵⁷⁴][..⁵⁷⁵]
- [22] Cleland-Huang, J., Gotel, O.C., Huffman Hayes, J., Mäder, P., Zisman, A., 2014. Software traceability: trends and future directions, in: Proceedings of the on Future of Software Engineering, ACM. [..⁵⁷⁶]pp. 55–69.

⁵³²removed: ,
⁵³³removed: ,
⁵³⁴removed: ,
⁵³⁵removed: .
⁵³⁶removed: , in:
⁵³⁷removed: ,
⁵³⁸removed: . pp.
⁵³⁹removed: .
⁵⁴⁰removed: ,
⁵⁴¹removed: .
⁵⁴²removed: ,
⁵⁴³removed: ,
⁵⁴⁴removed: ,
⁵⁴⁵removed: .
⁵⁴⁶removed: .
⁵⁴⁷removed: ,
⁵⁴⁸removed: .
⁵⁴⁹removed: ,
⁵⁵⁰removed: .
⁵⁵¹removed: ,
⁵⁵²removed: ,
⁵⁵³removed: .
⁵⁵⁴removed: ,
⁵⁵⁵removed: ,
⁵⁵⁶removed: ,
⁵⁵⁷removed: ,
⁵⁵⁸removed: ,
⁵⁵⁹removed: ,
⁵⁶⁰removed: ,
⁵⁶¹removed: .
⁵⁶²removed: , in:
⁵⁶³removed: ,
⁵⁶⁴removed: . pp.
⁵⁶⁵removed: .
⁵⁶⁶removed: ,
⁵⁶⁷removed: .
⁵⁶⁸removed: .
⁵⁶⁹removed: ,
⁵⁷⁰removed: .
⁵⁷¹removed: et al.,
⁵⁷²removed: et al.,
⁵⁷³removed: . volume
⁵⁷⁴removed: ,
⁵⁷⁵removed: .
⁵⁷⁶removed: . FOSE '14,

- [23] Coplien, J.O., 1999. ⁵⁷⁷ ⁵⁷⁸ ⁵⁷⁹ ⁵⁸⁰ ⁵⁸¹ Multi-paradigm Design for C+. ⁵⁸² ⁵⁸³ Addison-Wesley.
- [24] CVL, 2012. ⁵⁸⁴ ⁵⁸⁵ ⁵⁸⁶ Common variability language. <http://www.omgwiki.org/variability/doku.php>. Accessed: 2016-26-09.
- [25] Czarnecki, K., 2005. Overview of generative software development, in: Unconventional Programming Paradigms. Springer, ⁵⁸⁷ pp. 326–341. ⁵⁸⁸ ⁵⁸⁹
- [26] 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. ⁵⁹⁰ ⁵⁹¹ ⁵⁹² ⁵⁹³ ⁵⁹⁴ ⁵⁹⁵ ⁵⁹⁶ ⁵⁹⁷ ⁵⁹⁸ ⁵⁹⁹ ⁶⁰⁰ ⁶⁰¹ ⁶⁰² ⁶⁰³ ⁶⁰⁴ ⁶⁰⁵ ⁶⁰⁶ ⁶⁰⁷ ⁶⁰⁸
- [27] Fowler, M., Parsons, R., 2010. Domain-specific language. Pearson Education.
- [28] Fritsch, C., Lehn, A., Strohm, T., Bosch, R., 2002. Evaluating variability implementation mechanisms, in: ⁶⁰⁹ Proceedings of International Workshop on Product Line Engineering (PLEES), sn. pp. 59–64.
- [29] Gacek, C., Anastasopoulos, M., 2001. Implementing product line variabilities, in: ⁶¹⁰ ⁶¹¹ Proceedings of the 2001 symposium on Software reusability: putting software reuse in context, pp. 109–117. ⁶¹² ⁶¹³ ⁶¹⁴ ⁶¹⁵ ⁶¹⁶ ⁶¹⁷ ⁶¹⁸ ⁶¹⁹ ⁶²⁰ ⁶²¹ ⁶²² ⁶²³ ⁶²⁴

⁵⁷⁷removed: ,
⁵⁷⁸removed: .
⁵⁷⁹removed: ,
⁵⁸⁰removed: ,
⁵⁸¹removed: .
⁵⁸²removed: , in:
⁵⁸³removed: , pp.
⁵⁸⁴removed: ,
⁵⁸⁵removed: .
⁵⁸⁶removed: ,
⁵⁸⁷removed: .
⁵⁸⁸removed: ,
⁵⁸⁹removed: .
⁵⁹⁰removed: ,
⁵⁹¹removed: .
⁵⁹²removed: ,
⁵⁹³removed: ,
⁵⁹⁴removed: .
⁵⁹⁵removed: , in:
⁵⁹⁶removed: ,
⁵⁹⁷removed: . pp.
⁵⁹⁸removed: .
⁵⁹⁹removed: ,
⁶⁰⁰removed: .
⁶⁰¹removed: ,
⁶⁰²removed: ,
⁶⁰³removed: .
⁶⁰⁴removed: .
⁶⁰⁵removed: ,
⁶⁰⁶removed: .
⁶⁰⁷removed: ,
⁶⁰⁸removed: .
⁶⁰⁹removed: ,
⁶¹⁰removed: ,
⁶¹¹removed: .
⁶¹²removed: ,
⁶¹³removed: .
⁶¹⁴removed: ,
⁶¹⁵removed: ,
⁶¹⁶removed: ,
⁶¹⁷removed: ,
⁶¹⁸removed: ,
⁶¹⁹removed: .
⁶²⁰removed: .
⁶²¹removed: ,
⁶²²removed: .
⁶²³removed: ,
⁶²⁴removed: .

- [30] Ghosh, D., 2010. DSLs in action. Manning Publications Co. [..⁶²⁵][..⁶²⁶][..⁶²⁷][..⁶²⁸]
- [31] Gomaa, H., 2004. [..⁶²⁹][..⁶³⁰]Designing software product lines with uml: From use cases to pattern-based software architectures.
- [32] Gomaa, H., 2005. Designing software product lines with UML. IEEE.
- [33] 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. [..⁶³¹][..⁶³²][..⁶³³][..⁶³⁴][..⁶³⁵][..⁶³⁶][..⁶³⁷][..⁶³⁸][..⁶³⁹][..⁶⁴⁰][..⁶⁴¹][..⁶⁴²][..⁶⁴³][..⁶⁴⁴][..⁶⁴⁵][..⁶⁴⁶][..⁶⁴⁷][..⁶⁴⁸][..⁶⁴⁹][..⁶⁵⁰][..⁶⁵¹][..⁶⁵²][..⁶⁵³][..⁶⁵⁴][..⁶⁵⁵]
- [34] 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. [..⁶⁵⁶][..⁶⁵⁷][..⁶⁵⁸][..⁶⁵⁹][..⁶⁶⁰][..⁶⁶¹][..⁶⁶²][..⁶⁶³][..⁶⁶⁴][..⁶⁶⁵]
- [35] Hunt, J.M., McGregor, J.D., 2006. 9 implementing a variation point: A pattern language. [..⁶⁶⁶]Variability Management–Working with Variability Mechanisms , 83.
- [36] Jacobson, I., Griss, M., Jonsson, P., 1997. Software reuse: architecture, process and organization for business success. [..⁶⁶⁷]ACM Press/Addison-Wesley Publishing Co.
- [37] Jirapanthong, W., Zisman, A., 2009. Xtraque: traceability for product line systems. Software & Systems Modeling 8, 117–144. [..⁶⁶⁸][..⁶⁶⁹]
- [38] 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. [..⁶⁷⁰]DTIC Document.
- [39] Kang, K.C., Kim, S., Lee, J., Kim, K., Shin, E., Huh, M., 1998. FORM: A feature-oriented reuse method with domain-specific reference

⁶²⁵removed: ,
⁶²⁶removed: ,
⁶²⁷removed: ,
⁶²⁸removed: ,
⁶²⁹removed: ,
⁶³⁰removed: ,
⁶³¹removed: ,
⁶³²removed: ,
⁶³³removed: ,
⁶³⁴removed: ,
⁶³⁵removed: ,
⁶³⁶removed: ,
⁶³⁷removed: , in:
⁶³⁸removed: ,
⁶³⁹removed: ,
⁶⁴⁰removed: . p.
⁶⁴¹removed: ,
⁶⁴²removed: ,
⁶⁴³removed: ,
⁶⁴⁴removed: ,
⁶⁴⁵removed: ,
⁶⁴⁶removed: ,
⁶⁴⁷removed: ,
⁶⁴⁸removed: ,
⁶⁴⁹removed: ,
⁶⁵⁰removed: ,
⁶⁵¹removed: , in:
⁶⁵²removed: ,
⁶⁵³removed: . pp.
⁶⁵⁴removed: ,
⁶⁵⁵removed: ,
⁶⁵⁶removed: ,
⁶⁵⁷removed: ,
⁶⁵⁸removed: ,
⁶⁵⁹removed: ,
⁶⁶⁰removed: , in:
⁶⁶¹removed: ,
⁶⁶²removed: . p.
⁶⁶³removed: ,
⁶⁶⁴removed: ,
⁶⁶⁵removed: ,
⁶⁶⁶removed: ,
⁶⁶⁷removed: ,
⁶⁶⁸removed: ,
⁶⁶⁹removed: ,
⁶⁷⁰removed: Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst

- architectures. *Annals of Software Engineering* 5, pp. 143–168.
- [40] Kästner, C., 2010. Virtual Separation of Concerns: Toward Preprocessors 2.0. PhD dissertation. Otto-von-Guericke-Universität Magdeburg.
- [41] Kästner, C., Apel, S., Kuhlemann, M., 2008a. Granularity in software product lines, in: *Proceedings of the 30th international conference on Software engineering*, ACM. pp. 311–320.
- [42] 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. p. 5.
- [43] Kästner, C., Ostermann, K., Erdweg, S., 2012. A variability-aware module system, in: *ACM SIGPLAN Notices*, ACM. pp. 773–792.
- [44] Kästner, C., Trujillo, S., Apel, S., 2008b. Visualizing software product line variabilities in source code., in: *SPLC (2)*, pp. 303–312.
- [45] 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.
- [46] 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.
- [47] Lee, J., Muthig, D., 2008. Feature-oriented analysis and specification of dynamic product reconfiguration, in: *International Conference on Software Reuse*, Springer. pp. 154–165.
- [48] Legland, D., 2009 - 2017. JavaGeom. <http://geom-java.sourceforge.net/index.html>. Accessed June 15, 2017.
- [49] Liebig, J., Apel, S., Lengauer, C., Kästner, C., Schulze, M., 2010. An analysis of the variability in forty preprocessor-based software product lines, in: *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, IEEE. pp. 105–114.
- [50] 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.

671 removed: .
672 removed: ,
673 removed: .
674 removed: ,
675 removed: ,
676 removed: .
677 removed: . pp.
678 removed: .
679 removed: ,
680 removed: .
681 removed: ,
682 removed: ,
683 removed: .
684 removed: ,
685 removed: .
686 removed: ,
687 removed: .
688 removed: ,
689 removed: .
690 removed: .
691 removed: .
692 removed: ,
693 removed: ,
694 removed: ,
695 removed: ,
696 removed: .
697 removed: .
698 removed: ,
699 removed: .
700 removed: ,
701 removed: .
702 removed: ,
703 removed: ,
704 removed: .
705 removed: ,
706 removed: .
707 removed: ,
708 removed: ,
709 removed: ,
710 removed: ,
711 removed: ,
712 removed: .
713 removed: .

- [51] Meinicke, J., Thüm, T., Schröter, R., Benduhn, F., Leich, T., Saake, G., 2017. Mastering Software Variability with FeatureIDE. Springer.
- [52] 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: ⁷¹⁴Requirements Engineering Conference, 2007. RE'07. 15th IEEE International, IEEE. pp. 243–253. ⁷¹⁵ ⁷¹⁶ ⁷¹⁷ ⁷¹⁸ ⁷¹⁹ ⁷²⁰ ⁷²¹ ⁷²² ⁷²³ ⁷²⁴ ⁷²⁵ ⁷²⁶ ⁷²⁷ ⁷²⁸ ⁷²⁹ ⁷³⁰ ⁷³¹ ⁷³² ⁷³³ ⁷³⁴ ⁷³⁵ ⁷³⁶ ⁷³⁷ ⁷³⁸ ⁷³⁹ ⁷⁴⁰ ⁷⁴¹ ⁷⁴² ⁷⁴³ ⁷⁴⁴ ⁷⁴⁵ ⁷⁴⁶ ⁷⁴⁷ ⁷⁴⁸ ⁷⁴⁹ ⁷⁵⁰ ⁷⁵¹ ⁷⁵² ⁷⁵³ ⁷⁵⁴ ⁷⁵⁵ ⁷⁵⁶ ⁷⁵⁷ ⁷⁵⁸ ⁷⁵⁹ ⁷⁶⁰ ⁷⁶¹ ⁷⁶² ⁷⁶³ ⁷⁶⁴ ⁷⁶⁵ ⁷⁶⁶ ⁷⁶⁷ ⁷⁶⁸ ⁷⁶⁹ ⁷⁷⁰ ⁷⁷¹ ⁷⁷² ⁷⁷³ ⁷⁷⁴ ⁷⁷⁵ ⁷⁷⁶ ⁷⁷⁷ ⁷⁷⁸ ⁷⁷⁹ ⁷⁸⁰ ⁷⁸¹ ⁷⁸² ⁷⁸³ ⁷⁸⁴ ⁷⁸⁵ ⁷⁸⁶ ⁷⁸⁷ ⁷⁸⁸ ⁷⁸⁹ ⁷⁹⁰ ⁷⁹¹ ⁷⁹² ⁷⁹³ ⁷⁹⁴ ⁷⁹⁵ ⁷⁹⁶ ⁷⁹⁷ ⁷⁹⁸ ⁷⁹⁹ ⁸⁰⁰ ⁸⁰¹ ⁸⁰² ⁸⁰³ ⁸⁰⁴ ⁸⁰⁵ ⁸⁰⁶ ⁸⁰⁷ ⁸⁰⁸ ⁸⁰⁹ ⁸¹⁰ ⁸¹¹ ⁸¹² ⁸¹³ ⁸¹⁴ ⁸¹⁵ ⁸¹⁶ ⁸¹⁷ ⁸¹⁸ ⁸¹⁹ ⁸²⁰ ⁸²¹ ⁸²² ⁸²³ ⁸²⁴ ⁸²⁵ ⁸²⁶ ⁸²⁷ ⁸²⁸ ⁸²⁹ ⁸³⁰ ⁸³¹ ⁸³² ⁸³³ ⁸³⁴ ⁸³⁵ ⁸³⁶ ⁸³⁷ ⁸³⁸ ⁸³⁹ ⁸⁴⁰ ⁸⁴¹ ⁸⁴² ⁸⁴³ ⁸⁴⁴ ⁸⁴⁵ ⁸⁴⁶ ⁸⁴⁷ ⁸⁴⁸ ⁸⁴⁹ ⁸⁵⁰ ⁸⁵¹ ⁸⁵² ⁸⁵³ ⁸⁵⁴ ⁸⁵⁵ ⁸⁵⁶ ⁸⁵⁷ ⁸⁵⁸ ⁸⁵⁹ ⁸⁶⁰ ⁸⁶¹ ⁸⁶² ⁸⁶³ ⁸⁶⁴ ⁸⁶⁵ ⁸⁶⁶ ⁸⁶⁷ ⁸⁶⁸ ⁸⁶⁹ ⁸⁷⁰ ⁸⁷¹ ⁸⁷² ⁸⁷³ ⁸⁷⁴ ⁸⁷⁵ ⁸⁷⁶ ⁸⁷⁷ ⁸⁷⁸ ⁸⁷⁹ ⁸⁸⁰ ⁸⁸¹ ⁸⁸² ⁸⁸³ ⁸⁸⁴ ⁸⁸⁵ ⁸⁸⁶ ⁸⁸⁷ ⁸⁸⁸ ⁸⁸⁹ ⁸⁹⁰ ⁸⁹¹ ⁸⁹² ⁸⁹³ ⁸⁹⁴ ⁸⁹⁵ ⁸⁹⁶ ⁸⁹⁷ ⁸⁹⁸ ⁸⁹⁹ ⁹⁰⁰ ⁹⁰¹ ⁹⁰² ⁹⁰³ ⁹⁰⁴ ⁹⁰⁵ ⁹⁰⁶ ⁹⁰⁷ ⁹⁰⁸ ⁹⁰⁹ ⁹¹⁰ ⁹¹¹ ⁹¹² ⁹¹³ ⁹¹⁴ ⁹¹⁵ ⁹¹⁶ ⁹¹⁷ ⁹¹⁸ ⁹¹⁹ ⁹²⁰ ⁹²¹ ⁹²² ⁹²³ ⁹²⁴ ⁹²⁵ ⁹²⁶ ⁹²⁷ ⁹²⁸ ⁹²⁹ ⁹³⁰ ⁹³¹ ⁹³² ⁹³³ ⁹³⁴ ⁹³⁵ ⁹³⁶ ⁹³⁷ ⁹³⁸ ⁹³⁹ ⁹⁴⁰ ⁹⁴¹ ⁹⁴² ⁹⁴³ ⁹⁴⁴ ⁹⁴⁵ ⁹⁴⁶ ⁹⁴⁷ ⁹⁴⁸ ⁹⁴⁹ ⁹⁵⁰ ⁹⁵¹ ⁹⁵² ⁹⁵³ ⁹⁵⁴ ⁹⁵⁵ ⁹⁵⁶ ⁹⁵⁷ ⁹⁵⁸ ⁹⁵⁹ ⁹⁶⁰ ⁹⁶¹ ⁹⁶² ⁹⁶³ ⁹⁶⁴ ⁹⁶⁵ ⁹⁶⁶ ⁹⁶⁷ ⁹⁶⁸ ⁹⁶⁹ ⁹⁷⁰ ⁹⁷¹ ⁹⁷² ⁹⁷³ ⁹⁷⁴ ⁹⁷⁵ ⁹⁷⁶ ⁹⁷⁷ ⁹⁷⁸ ⁹⁷⁹ ⁹⁸⁰ ⁹⁸¹ ⁹⁸² ⁹⁸³ ⁹⁸⁴ ⁹⁸⁵ ⁹⁸⁶ ⁹⁸⁷ ⁹⁸⁸ ⁹⁸⁹ ⁹⁹⁰ ⁹⁹¹ ⁹⁹² ⁹⁹³ ⁹⁹⁴ ⁹⁹⁵ ⁹⁹⁶ ⁹⁹⁷ ⁹⁹⁸ ⁹⁹⁹ ¹⁰⁰⁰ ¹⁰⁰¹ ¹⁰⁰² ¹⁰⁰³ ¹⁰⁰⁴ ¹⁰⁰⁵ ¹⁰⁰⁶ ¹⁰⁰⁷ ¹⁰⁰⁸ ¹⁰⁰⁹ ¹⁰¹⁰ ¹⁰¹¹ ¹⁰¹² ¹⁰¹³ ¹⁰¹⁴ ¹⁰¹⁵ ¹⁰¹⁶ ¹⁰¹⁷ ¹⁰¹⁸ ¹⁰¹⁹ ¹⁰²⁰ ¹⁰²¹ ¹⁰²² ¹⁰²³ ¹⁰²⁴ ¹⁰²⁵ ¹⁰²⁶ ¹⁰²⁷ ¹⁰²⁸ ¹⁰²⁹ ¹⁰³⁰ ¹⁰³¹ ¹⁰³² ¹⁰³³ ¹⁰³⁴ ¹⁰³⁵ ¹⁰³⁶ ¹⁰³⁷ ¹⁰³⁸ ¹⁰³⁹ ¹⁰⁴⁰ ¹⁰⁴¹ ¹⁰⁴² ¹⁰⁴³ ¹⁰⁴⁴ ¹⁰⁴⁵ ¹⁰⁴⁶ ¹⁰⁴⁷ ¹⁰⁴⁸ ¹⁰⁴⁹ ¹⁰⁵⁰ ¹⁰⁵¹ ¹⁰⁵² ¹⁰⁵³ ¹⁰⁵⁴ ¹⁰⁵⁵ ¹⁰⁵⁶ ¹⁰⁵⁷ ¹⁰⁵⁸ ¹⁰⁵⁹ ¹⁰⁶⁰ ¹⁰⁶¹ ¹⁰⁶² ¹⁰⁶³ ¹⁰⁶⁴ ¹⁰⁶⁵ ¹⁰⁶⁶ ¹⁰⁶⁷ ¹⁰⁶⁸ ¹⁰⁶⁹ ¹⁰⁷⁰ ¹⁰⁷¹ ¹⁰⁷² ¹⁰⁷³ ¹⁰⁷⁴ ¹⁰⁷⁵ ¹⁰⁷⁶ ¹⁰⁷⁷ ¹⁰⁷⁸ ¹⁰⁷⁹ ¹⁰⁸⁰ ¹⁰⁸¹ ¹⁰⁸² ¹⁰⁸³ ¹⁰⁸⁴ ¹⁰⁸⁵ ¹⁰⁸⁶ ¹⁰⁸⁷ ¹⁰⁸⁸ ¹⁰⁸⁹ ¹⁰⁹⁰ ¹⁰⁹¹ ¹⁰⁹² ¹⁰⁹³ ¹⁰⁹⁴ ¹⁰⁹⁵ ¹⁰⁹⁶ ¹⁰⁹⁷ ¹⁰⁹⁸ ¹⁰⁹⁹ ¹¹⁰⁰ ¹¹⁰¹ ¹¹⁰² ¹¹⁰³ ¹¹⁰⁴ ¹¹⁰⁵ ¹¹⁰⁶ ¹¹⁰⁷ ¹¹⁰⁸ ¹¹⁰⁹ ¹¹¹⁰ ¹¹¹¹ ¹¹¹² ¹¹¹³ ¹¹¹⁴ ¹¹¹⁵ ¹¹¹⁶ ¹¹¹⁷ ¹¹¹⁸ ¹¹¹⁹ ¹¹²⁰ ¹¹²¹ ¹¹²² ¹¹²³ ¹¹²⁴ ¹¹²⁵ ¹¹²⁶ ¹¹²⁷ ¹¹²⁸ ¹¹²⁹ ¹¹³⁰ ¹¹³¹ ¹¹³² ¹¹³³ ¹¹³⁴ ¹¹³⁵ ¹¹³⁶ ¹¹³⁷ ¹¹³⁸ ¹¹³⁹ ¹¹⁴⁰ ¹¹⁴¹ ¹¹⁴² ¹¹⁴³ ¹¹⁴⁴ ¹¹⁴⁵ ¹¹⁴⁶ ¹¹⁴⁷ ¹¹⁴⁸ ¹¹⁴⁹ ¹¹⁵⁰ ¹¹⁵¹ ¹¹⁵² ¹¹⁵³ ¹¹⁵⁴ ¹¹⁵⁵ ¹¹⁵⁶ ¹¹⁵⁷ ¹¹⁵⁸ ¹¹⁵⁹ ¹¹⁶⁰ ¹¹⁶¹ ¹¹⁶² ¹¹⁶³ ¹¹⁶⁴ ¹¹⁶⁵ ¹¹⁶⁶ ¹¹⁶⁷ ¹¹⁶⁸ ¹¹⁶⁹ ¹¹⁷⁰ ¹¹⁷¹ ¹¹⁷² ¹¹⁷³ ¹¹⁷⁴ ¹¹⁷⁵ ¹¹⁷⁶ ¹¹⁷⁷ ¹¹⁷⁸ ¹¹⁷⁹ ¹¹⁸⁰ ¹¹⁸¹ ¹¹⁸² ¹¹⁸³ ¹¹⁸⁴ ¹¹⁸⁵ ¹¹⁸⁶ ¹¹⁸⁷ ¹¹⁸⁸ ¹¹⁸⁹ ¹¹⁹⁰ ¹¹⁹¹ ¹¹⁹² ¹¹⁹³ ¹¹⁹⁴ ¹¹⁹⁵ ¹¹⁹⁶ ¹¹⁹⁷ ¹¹⁹⁸ ¹¹⁹⁹ ¹²⁰⁰ ¹²⁰¹ ¹²⁰² ¹²⁰³ ¹²⁰⁴ ¹²⁰⁵ ¹²⁰⁶ ¹²⁰⁷ ¹²⁰⁸ ¹²⁰⁹ ¹²¹⁰ ¹²¹¹ ¹²¹² ¹²¹³ ¹²¹⁴ ¹²¹⁵ ¹²¹⁶ ¹²¹⁷ ¹²¹⁸ ¹²¹⁹ ¹²²⁰ ¹²²¹ ¹²²² ¹²²³ ¹²²⁴ ¹²²⁵ ¹²²⁶ ¹²²⁷ ¹²²⁸ ¹²²⁹ ¹²³⁰ ¹²³¹ ¹²³² ¹²³³ ¹²³⁴ ¹²³⁵ ¹²³⁶ ¹²³⁷ ¹²³⁸ ¹²³⁹ ¹²⁴⁰ ¹²⁴¹ ¹²⁴² ¹²⁴³ ¹²⁴⁴ ¹²⁴⁵ ¹²⁴⁶ ¹²⁴⁷ ¹²⁴⁸ ¹²⁴⁹ ¹²⁵⁰ ¹²⁵¹ ¹²⁵² ¹²⁵³ ¹²⁵⁴ ¹²⁵⁵ ¹²⁵⁶ ¹²⁵⁷ ¹²⁵⁸ ¹²⁵⁹ ¹²⁶⁰ ¹²⁶¹ ¹²⁶² ¹²⁶³ ¹²⁶⁴ ¹²⁶⁵ ¹²⁶⁶ ¹²⁶⁷ ¹²⁶⁸ ¹²⁶⁹ ¹²⁷⁰ ¹²⁷¹ ¹²⁷² ¹²⁷³ ¹²⁷⁴ ¹²⁷⁵ ¹²⁷⁶ ¹²⁷⁷ ¹²⁷⁸ ¹²⁷⁹ ¹²⁸⁰ ¹²⁸¹ ¹²⁸² ¹²⁸³ ¹²⁸⁴ ¹²⁸⁵ ¹²⁸⁶ ¹²⁸⁷ ¹²⁸⁸ ¹²⁸⁹ ¹²⁹⁰ ¹²⁹¹ ¹²⁹² ¹²⁹³ ¹²⁹⁴ ¹²⁹⁵ ¹²⁹⁶ ¹²⁹⁷ ¹²⁹⁸ ¹²⁹⁹ ¹³⁰⁰ ¹³⁰¹ ¹³⁰² ¹³⁰³ ¹³⁰⁴ ¹³⁰⁵ ¹³⁰⁶ ¹³⁰⁷ ¹³⁰⁸ ¹³⁰⁹ ¹³¹⁰ ¹³¹¹ ¹³¹² ¹³¹³ ¹³¹⁴ ¹³¹⁵ ¹³¹⁶ ¹³¹⁷ ¹³¹⁸ ¹³¹⁹ ¹³²⁰ ¹³²¹ ¹³²² ¹³²³ ¹³²⁴ ¹³²⁵ ¹³²⁶ ¹³²⁷ ¹³²⁸ ¹³²⁹ ¹³³⁰ ¹³³¹ ¹³³² ¹³³³ ¹³³⁴ ¹³³⁵ ¹³³⁶ ¹³³⁷ ¹³³⁸ ¹³³⁹ ¹³⁴⁰ ¹³⁴¹ ¹³⁴² ¹³⁴³ ¹³⁴⁴ ¹³⁴⁵ ¹³⁴⁶ ¹³⁴⁷ ¹³⁴⁸ ¹³⁴⁹ ¹³⁵⁰ ¹³⁵¹ ¹³⁵² ¹³⁵³ ¹³⁵⁴ ¹³⁵⁵ ¹³⁵⁶ ¹³⁵⁷ ¹³⁵⁸ ¹³⁵⁹ ¹³⁶⁰ ¹³⁶¹ ¹³⁶² ¹³⁶³ ¹³⁶⁴ ¹³⁶⁵ ¹³⁶⁶ ¹³⁶⁷ ¹³⁶⁸ ¹³⁶⁹ ¹³⁷⁰ ¹³⁷¹ ¹³⁷² ¹³⁷³ ¹³⁷⁴ ¹³⁷⁵ ¹³⁷⁶ ¹³⁷⁷ ¹³⁷⁸ ¹³⁷⁹ ¹³⁸⁰ ¹³⁸¹ ¹³⁸² ¹³⁸³ ¹³⁸⁴ ¹³⁸⁵ ¹³⁸⁶ ¹³⁸⁷ ¹³⁸⁸ ¹³⁸⁹ ¹³⁹⁰ ¹³⁹¹ ¹³⁹² ¹³⁹³ ¹³⁹⁴ ¹³⁹⁵ ¹³⁹⁶ ¹³⁹⁷ ¹³⁹⁸ ¹³⁹⁹ ¹⁴⁰⁰ ¹⁴⁰¹ ¹⁴⁰² ¹⁴⁰³ ¹⁴⁰⁴ ¹⁴⁰⁵ ¹⁴⁰⁶ ¹⁴⁰⁷ ¹⁴⁰⁸ ¹⁴⁰⁹ ¹⁴¹⁰ ¹⁴¹¹ ¹⁴¹² ¹⁴¹³ ¹⁴¹⁴ ¹⁴¹⁵ ¹⁴¹⁶ ¹⁴¹⁷ ¹⁴¹⁸ ¹⁴¹⁹ ¹⁴²⁰ ¹⁴²¹ ¹⁴²² ¹⁴²³ ¹⁴²⁴ ¹⁴²⁵ ¹⁴²⁶ ¹⁴²⁷ ¹⁴²⁸ ¹⁴²⁹ ¹⁴³⁰ ¹⁴³¹ ¹⁴³² ¹⁴³³ ¹⁴³⁴ ¹⁴³⁵ ¹⁴³⁶ ¹⁴³⁷ ¹⁴³⁸ ¹⁴³⁹ ¹⁴⁴⁰ ¹⁴⁴¹ ¹⁴⁴² ¹⁴⁴³ ¹⁴⁴⁴ ¹⁴⁴⁵ ¹⁴⁴⁶ ¹⁴⁴⁷ ¹⁴⁴⁸ ¹⁴⁴⁹ ¹⁴⁵⁰ ¹⁴⁵¹ ¹⁴⁵² ¹⁴⁵³ ¹⁴⁵⁴ ¹⁴⁵⁵ ¹⁴⁵⁶ ¹⁴⁵⁷ ¹⁴⁵⁸ ¹⁴⁵⁹ ¹⁴⁶⁰ ¹⁴⁶¹ ¹⁴⁶² ¹⁴⁶³ ¹⁴⁶⁴ ¹⁴⁶⁵ ¹⁴⁶⁶ ¹⁴⁶⁷ ¹⁴⁶⁸ ¹⁴⁶⁹ ¹⁴⁷⁰ ¹⁴⁷¹ ¹⁴⁷² ¹⁴⁷³ ¹⁴⁷⁴ ¹⁴⁷⁵ ¹⁴⁷⁶ ¹⁴⁷⁷ ¹⁴⁷⁸ ¹⁴⁷⁹ ¹⁴⁸⁰ ¹⁴⁸¹ ¹⁴⁸² ¹⁴⁸³ ¹⁴⁸⁴ ¹⁴⁸⁵ ¹⁴⁸⁶ ¹⁴⁸⁷ ¹⁴⁸⁸ ¹⁴⁸⁹ ¹⁴⁹⁰ ¹⁴⁹¹ ¹⁴⁹² ¹⁴⁹³ ¹⁴⁹⁴ ¹⁴⁹⁵ ¹⁴⁹⁶ ¹⁴⁹⁷ ¹⁴⁹⁸ ¹⁴⁹⁹ ¹⁵⁰⁰ ¹⁵⁰¹ ¹⁵⁰² ¹⁵⁰³ ¹⁵⁰⁴ ¹⁵⁰⁵ ¹⁵⁰⁶ ¹⁵⁰⁷ ¹⁵⁰⁸ ¹⁵⁰⁹ ¹⁵¹⁰ ¹⁵¹¹ ¹⁵¹² ¹⁵¹³ ¹⁵¹⁴ ¹⁵¹⁵ ¹⁵¹⁶ ¹⁵¹⁷ ¹⁵¹⁸ ¹⁵¹⁹ ¹⁵²⁰ ¹⁵²¹ ¹⁵²² ¹⁵²³ ¹⁵²⁴ ¹⁵²⁵ ¹⁵²⁶ ¹⁵²⁷ ¹⁵²⁸ ¹⁵²⁹ ¹⁵³⁰ ¹⁵³¹ ¹⁵³² ¹⁵³³ ¹⁵³⁴ ¹⁵³⁵ ¹⁵³⁶ ¹⁵³⁷ ¹⁵³⁸ ¹⁵³⁹ ¹⁵⁴⁰ ¹⁵⁴¹ ¹⁵⁴² ¹⁵⁴³ ¹⁵⁴⁴ ¹⁵⁴⁵ ¹⁵⁴⁶ ¹⁵⁴⁷ ¹⁵⁴⁸ ¹⁵⁴⁹ ¹⁵⁵⁰ ¹⁵⁵¹ ¹⁵⁵² ¹⁵⁵³ ¹⁵⁵⁴ ¹⁵⁵⁵ ¹⁵⁵⁶ ¹⁵⁵⁷ ¹⁵⁵⁸ ¹⁵⁵⁹ ¹⁵⁶⁰ ¹⁵⁶¹ ¹⁵⁶² ¹⁵⁶³ ¹⁵⁶⁴ ¹⁵⁶⁵ ¹⁵⁶⁶ ¹⁵⁶⁷ ¹⁵⁶⁸ ¹⁵⁶⁹ ¹⁵⁷⁰ ¹⁵⁷¹ ¹⁵⁷² ¹⁵⁷³ ¹⁵⁷⁴ ¹⁵⁷⁵ ¹⁵⁷⁶ ¹⁵⁷⁷ ¹⁵⁷⁸ ¹⁵⁷⁹ ¹⁵⁸⁰ ¹⁵⁸¹ ¹⁵⁸² ¹⁵⁸³ ¹⁵⁸⁴ ¹⁵⁸⁵ ¹⁵⁸⁶ ¹⁵⁸⁷ ¹⁵⁸⁸ ¹⁵⁸⁹ ¹⁵⁹⁰ ¹⁵⁹¹ ¹⁵⁹² ¹⁵⁹³ ¹⁵⁹⁴ ¹⁵⁹⁵ ¹⁵⁹⁶ ¹⁵⁹⁷ ¹⁵⁹⁸ ¹⁵⁹⁹ ¹⁶⁰⁰ ¹⁶⁰¹ ¹⁶⁰² ¹⁶⁰³ ¹⁶⁰⁴ ¹⁶⁰⁵ ¹⁶⁰⁶ ¹⁶⁰⁷ ¹⁶⁰⁸ ¹⁶⁰⁹ ¹⁶¹⁰ ¹⁶¹¹ ¹⁶¹² ¹⁶¹³ ¹⁶¹⁴ ¹⁶¹⁵ ¹⁶¹⁶ ¹⁶¹⁷ ¹⁶¹⁸ ¹⁶¹⁹ ¹⁶²⁰ ¹⁶²¹ ¹⁶²² ¹⁶²³ ¹⁶²⁴ ¹⁶²⁵ ¹⁶²⁶ ¹⁶²⁷ ¹⁶²⁸ ¹⁶²⁹ ¹⁶³⁰ ¹⁶³¹ ¹⁶³² ¹⁶³³ ¹⁶³⁴ ¹⁶³⁵ ¹⁶³⁶ ¹⁶³⁷ ¹⁶³⁸ ¹⁶³⁹ ¹⁶⁴⁰ ¹⁶⁴¹ ¹⁶⁴² ¹⁶⁴³ ¹⁶⁴⁴ ¹⁶⁴⁵ ¹⁶⁴⁶ ¹⁶⁴⁷ ¹⁶⁴⁸ ¹⁶⁴⁹ ¹⁶⁵⁰ ¹⁶⁵¹ ¹⁶⁵² ¹⁶⁵³ ¹⁶⁵⁴ ¹⁶⁵⁵ ¹⁶⁵⁶ ¹⁶⁵⁷ ¹⁶⁵⁸ ¹⁶⁵⁹ ¹⁶⁶⁰ ¹⁶⁶¹ ¹⁶⁶² ¹⁶⁶³ ¹⁶⁶⁴ ¹⁶⁶⁵ ¹⁶⁶⁶ ¹⁶⁶⁷ ¹⁶⁶⁸ ¹⁶⁶⁹ ¹⁶⁷⁰ ¹⁶⁷¹ ¹⁶⁷² ¹⁶⁷³ ¹⁶⁷⁴ ¹⁶⁷⁵ ¹⁶⁷⁶ ¹⁶⁷⁷ ¹⁶⁷⁸ ¹⁶⁷⁹ ¹⁶⁸⁰ ¹⁶⁸¹ ¹⁶⁸² ¹⁶⁸³ ¹⁶⁸⁴ ¹⁶⁸⁵ ¹⁶⁸⁶ ¹⁶⁸⁷ ¹⁶⁸⁸ ¹⁶⁸⁹ ¹⁶⁹⁰ ¹⁶⁹¹ ¹⁶⁹² ¹⁶⁹³ ¹⁶⁹⁴ ¹⁶⁹⁵ ¹⁶⁹⁶ ¹⁶⁹⁷ ¹⁶⁹⁸ ¹⁶⁹⁹ ¹⁷⁰⁰ ¹⁷⁰¹ ¹⁷⁰² ¹⁷⁰³ ¹⁷⁰⁴ ¹⁷⁰⁵ ¹⁷⁰⁶ ¹⁷⁰⁷ ¹⁷⁰⁸ ¹⁷⁰⁹ ¹⁷¹⁰ ¹⁷¹¹ ¹⁷¹² ¹⁷¹³ ¹⁷¹⁴ ¹⁷¹⁵ ¹⁷¹⁶ ¹⁷¹⁷ ¹⁷¹⁸ ¹⁷¹⁹ ¹⁷²⁰ ¹⁷²¹ ¹⁷²² ¹⁷²³ ¹⁷²⁴ ¹⁷²⁵ ¹⁷²⁶ ¹⁷²⁷ ¹⁷²⁸ ¹⁷²⁹ ¹⁷³⁰ ¹⁷³¹ ¹⁷³² ¹⁷³³ ¹⁷³⁴ ¹⁷³⁵ ¹⁷³⁶ ¹⁷³⁷ ¹⁷³⁸ ¹⁷³⁹ ¹⁷⁴⁰ ¹⁷⁴¹ ¹⁷⁴² ¹⁷⁴³ ¹⁷⁴⁴ ¹⁷⁴⁵ ¹⁷⁴⁶ ¹⁷⁴⁷ ¹⁷⁴⁸ ¹⁷⁴⁹ ¹⁷⁵⁰ ¹⁷⁵¹ ¹⁷⁵² ¹⁷⁵³ ¹⁷⁵⁴ ¹⁷⁵⁵ ¹⁷⁵⁶ ¹⁷⁵⁷ ¹⁷⁵⁸ ¹⁷⁵⁹ ¹⁷⁶⁰ ¹⁷⁶¹ ¹⁷⁶² ¹⁷⁶³ ¹⁷⁶⁴ ¹⁷⁶⁵ ¹⁷⁶⁶ ¹⁷⁶⁷ ¹⁷⁶⁸ ¹⁷⁶⁹ ¹⁷⁷⁰ ¹⁷⁷¹ ¹⁷⁷² ¹⁷⁷³ ¹⁷⁷⁴ ¹⁷⁷⁵ ¹⁷⁷⁶ ¹⁷⁷⁷ ¹⁷⁷⁸ ¹⁷⁷⁹ ¹⁷⁸⁰ ¹⁷⁸¹ ¹⁷⁸² ¹⁷⁸³ ¹⁷⁸⁴ ¹⁷⁸⁵ ¹⁷⁸⁶ ¹⁷⁸⁷ ¹⁷⁸⁸ ¹⁷⁸⁹ ¹⁷⁹⁰ ¹⁷⁹¹ ¹⁷⁹² ¹⁷⁹³ ¹⁷⁹⁴ ¹⁷⁹⁵ ¹⁷⁹⁶ ¹⁷⁹⁷ ¹⁷⁹⁸ ¹⁷⁹⁹ ¹⁸⁰⁰ ¹⁸⁰¹ ¹⁸⁰² ¹⁸⁰³ ¹⁸⁰⁴ ¹⁸⁰⁵ ¹⁸⁰⁶ ¹⁸⁰⁷ ¹⁸⁰⁸ ¹⁸⁰⁹ ¹⁸¹⁰ ¹⁸¹¹ ¹⁸¹² ¹⁸¹³ ¹⁸¹⁴ ¹⁸¹⁵ ¹⁸¹⁶ ¹⁸¹⁷ ¹⁸¹⁸ ¹⁸¹⁹ ¹⁸²⁰ ¹⁸²¹ ¹⁸²² ¹⁸²³ ¹⁸²⁴ ¹⁸²⁵ ¹⁸²⁶ ¹⁸²⁷ ¹⁸²⁸ ¹⁸²⁹ ¹⁸³⁰ ¹⁸³¹ ¹⁸³² ¹⁸³³ ¹⁸³⁴ ¹⁸³⁵ ¹⁸³⁶ ¹⁸³⁷ ¹⁸³⁸ ¹⁸³⁹ ¹⁸⁴⁰ ¹⁸⁴¹ ¹⁸⁴² ¹⁸⁴³ ¹⁸⁴⁴ ¹⁸⁴⁵ ¹⁸⁴⁶ ¹⁸⁴⁷ ¹⁸⁴⁸ ¹⁸⁴⁹ ¹⁸⁵⁰ ¹⁸⁵¹ ¹⁸⁵² ¹⁸⁵³ ¹⁸⁵⁴ ¹⁸⁵⁵ ¹⁸⁵⁶ ¹⁸⁵⁷ ¹⁸⁵⁸ ¹⁸⁵⁹ ¹⁸⁶⁰ ¹⁸⁶¹ ¹⁸⁶² ¹⁸⁶³ ¹⁸⁶⁴ ¹⁸⁶⁵ ¹⁸⁶⁶ ¹⁸⁶⁷ ¹⁸⁶⁸ ¹⁸⁶⁹ ¹⁸⁷⁰ ¹⁸⁷¹ ¹⁸⁷² ¹⁸⁷³ ¹⁸⁷⁴ ¹⁸⁷⁵ ¹⁸⁷⁶ ¹⁸⁷⁷ ¹⁸⁷⁸ ¹⁸⁷⁹ ¹⁸⁸⁰ ¹⁸⁸¹ ¹⁸⁸² ¹⁸⁸³ ¹⁸⁸⁴ ¹⁸⁸⁵ ¹⁸⁸⁶ ¹⁸⁸⁷ ¹⁸⁸⁸ ¹⁸⁸⁹ ¹⁸⁹⁰ ¹⁸⁹¹ ¹⁸⁹² ¹⁸⁹³ ¹⁸⁹⁴ ¹⁸⁹⁵ ¹⁸⁹⁶ ¹⁸⁹⁷ ¹⁸⁹⁸ ¹⁸⁹⁹ ¹⁹⁰⁰ ¹⁹⁰¹ ¹⁹⁰² ¹⁹⁰³ ¹⁹⁰⁴ ¹⁹⁰⁵ ¹⁹⁰⁶ ¹⁹⁰⁷ ¹⁹⁰⁸ ¹⁹⁰⁹ ¹⁹¹⁰ ¹⁹¹¹ ¹⁹¹² ¹⁹¹³ ¹⁹¹⁴ ¹⁹¹⁵ ¹⁹¹⁶ ¹⁹¹⁷ ¹⁹¹⁸ ¹⁹¹⁹ ¹⁹²⁰ ¹⁹²¹ ¹⁹²² ¹⁹²³ ¹⁹²⁴ ¹⁹²⁵ ¹⁹²⁶ ¹⁹²⁷ ¹⁹²⁸ ¹⁹²⁹ ¹⁹³⁰ ¹⁹³¹ ¹⁹³² ¹⁹³³ ¹⁹³⁴ ¹⁹³⁵ ¹⁹³⁶ ¹⁹³⁷ ¹⁹³⁸ ¹⁹³⁹ ¹⁹⁴⁰ ¹⁹⁴¹ ¹⁹⁴² ¹⁹⁴³ ¹⁹⁴⁴ ¹⁹⁴⁵ ¹⁹⁴⁶ ¹⁹⁴⁷ ¹⁹⁴⁸ ¹⁹⁴⁹ ¹⁹⁵⁰ ¹⁹⁵¹ ¹⁹⁵² ¹⁹⁵³ ¹⁹⁵⁴ ¹⁹⁵⁵ ¹⁹⁵⁶ ¹⁹⁵⁷

- Systems and Software Product Line Conference-Volume B, [..⁷⁴⁹]pp. 134–136. [..⁷⁵⁰] [..⁷⁵¹]
- [65] Rosenmüller, M., Siegmund, N., Apel, S., Saake, G., 2011. Flexible feature binding in software product lines. *Automated Software Engineering* 18, 163–197. [..⁷⁵²] [..⁷⁵³]
- [66] Schaefer, I., Bettini, L., Bono, V., Damiani, F., Tanzarella, N., 2010. Delta-oriented programming of software product lines, in: *Software Product Lines: Going Beyond*. Springer, [..⁷⁵⁴]pp. 77–91. [..⁷⁵⁵] [..⁷⁵⁶]
- [67] Schmid, K., John, I., 2003. Generic variability management and its application to product line modelling. *Software Variability Management*, 13.
- [68] Schmid, K., John, I., 2004. A customizable approach to full lifecycle variability management. *Science of Computer Programming* 53, 259–284. [..⁷⁵⁷] [..⁷⁵⁸]
- [69] Sinnema, M., Deelstra, S., Nijhuis, J., Bosch, J., 2004a. Covamof: A framework for modeling variability in software product families, in: *Software product lines*. Springer, [..⁷⁵⁹]pp. 197–213. [..⁷⁶⁰] [..⁷⁶¹]
- [70] 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*.
- [71] 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. [..⁷⁶²] [..⁷⁶³]
- [72] Svahnberg, M., Van Gorp, J., Bosch, J., 2005. A taxonomy of variability realization techniques [..⁷⁶⁴] [..⁷⁶⁵]. *Software: Practice and Experience* 35, 705–754.
- [73] Systems, P., [..⁷⁶⁶]2018. pure::variants. <https://www.pure-systems.com/products/pure-variants-9.html>. Accessed January 17, 2019.
- [74] Tërnavá, Xh., 2017. Handling variability at the code level: modeling, tracing and checking consistency. Ph.D. thesis. Université Côte d’Azur.
- [75] Tërnavá, 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. [..⁷⁶⁷] [..⁷⁶⁸]
- [76] Tërnavá, 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. [..⁷⁶⁹] [..⁷⁷⁰]
- [77] Tërnavá, Xh., Collet, P., 2017. Tracing imperfectly modular variability in software product line implementation, in: *International Conference on Software Reuse*, Springer. pp. 112–120. [..⁷⁷¹] [..⁷⁷²]
- [78] Tërnavá, 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. [..⁷⁷³] [..⁷⁷⁴]
- [79] Thum, T., Kastner, C., Erdweg, S., Siegmund, N., 2011. Abstract features in feature modeling, in: [..⁷⁷⁵] *Software Product Line Conference (SPLC), 2011 15th International*, IEEE. pp. 191–200. [..⁷⁷⁶] [..⁷⁷⁷]
- [80] 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.

⁷⁴⁹removed: .
⁷⁵⁰removed: ,
⁷⁵¹removed: .
⁷⁵²removed: ,
⁷⁵³removed: .
⁷⁵⁴removed: .
⁷⁵⁵removed: ,
⁷⁵⁶removed: .
⁷⁵⁷removed: ,
⁷⁵⁸removed: .
⁷⁵⁹removed: .
⁷⁶⁰removed: ,
⁷⁶¹removed: .
⁷⁶²removed: ,
⁷⁶³removed: .
⁷⁶⁴removed: ,
⁷⁶⁵removed: . pp.
⁷⁶⁶removed: .
⁷⁶⁷removed: ,
⁷⁶⁸removed: .
⁷⁶⁹removed: ,
⁷⁷⁰removed: .
⁷⁷¹removed: ,
⁷⁷²removed: .
⁷⁷³removed: ,
⁷⁷⁴removed: .
⁷⁷⁵removed: ,
⁷⁷⁶removed: ,
⁷⁷⁷removed: .

- [81] VEL, 2018. ⁷⁷⁸ ⁷⁷⁹ ⁷⁸⁰ ⁷⁸¹ ⁷⁸² ⁷⁸³ ⁷⁸⁴ Variability Exchange Language. ⁷⁸⁵ ⁷⁸⁶ ⁷⁸⁷ ⁷⁸⁸ <https://www.variability-exchange-language.org/>. ⁷⁸⁹ Daimler AG, pure systems GmbH, and Fraunhofer Institute. Accessed August 22, 2019.
- [82] Voelter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L., Visser, E., Wachsmuth, G., 2013. DSL engineering. <http://dslbook.org>.
- [83] Voelter, M., Visser, E., 2011. Product line engineering using domain-specific languages, in: ⁷⁹⁰ Software Product Line Conference (SPLC), 2011 15th International, IEEE. pp. 70–79. ⁷⁹¹ ⁷⁹²
- [84] 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. 168–177⁷⁹³ ⁷⁹⁴.

⁷⁷⁸removed: ,
⁷⁷⁹removed: .
⁷⁸⁰removed: ,
⁷⁸¹removed: ,
⁷⁸²removed: ,
⁷⁸³removed: ,
⁷⁸⁴removed: ,
⁷⁸⁵removed: , in:
⁷⁸⁶removed: ,
⁷⁸⁷removed: . pp.
⁷⁸⁸removed: .
⁷⁸⁹removed: ,
⁷⁹⁰removed: ,
⁷⁹¹removed: ,
⁷⁹²removed: .
⁷⁹³removed: .
⁷⁹⁴removed: ,