Handling variability at the code level : modeling, tracing and checking consistency

Xhevahire Tërnava

Université Côte d'Azur

DOCTORAL SCHOOL STIC

SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

P H D T H E S I S

to obtain the title of

PhD of Science

of the Université Côte d'Azur

Specialty : COMPUTER SCIENCE

Defended by

Xhevahire TËRNAVA

Handling Variability at the Code Level: Modeling, Tracing and Checking Consistency

Thesis Advisor: Philippe COLLET

CNRS, I3S, Sophia Antipolis, France

defended on 01 December 2017

Jury :

Olivier BARAIS	-	Professor, Université de Rennes 1	Reviewer
Mireille BLAY-FORNARINO	-	Professor, Université Côte d'Azur	Examiner
Philippe COLLET	-	Professor, Université Côte d'Azur	Advisor
Laurence DUCHIEN	-	Professor, Université de Lille	Examiner
Tewfik ZIADI	-	Associate Professor, HDR, UPMC - LIP6	Reviewer

Université Côte d'Azur

ECOLE DOCTORALE STIC

SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

Thèse de doctorat

Présentée en vue de l'obtention du grade de

docteur en Sciences

de l'Université Côte d'Azur

Mention : INFORMATIQUE

Présentée et soutenue par

Xhevahire TËRNAVA

Gestion de la Variabilité au Niveau du Code : Modélisation, Traçabilité et Vérification de Cohérence

Dirigée par: Philippe COLLET

CNRS, I3S, Sophia Antipolis, France

Soutenance prévue le 01 décembre 2017

Devant le jury composé de:

Olivier BARAIS	-	Professeur des Universités, Rennes 1	Rapporteur
Mireille BLAY-FORNARINO	-	Professeur des Universités, Côte d'Azur	Examinatrice
Philippe COLLET	-	Professeur des Universités, Côte d'Azur	Directeur
Laurence DUCHIEN	-	Professeur des Universités, Lille	Examinatrice
Tewfik ZIADI	-	Maître de conférences, HDR, UPMC - LIP6	Rapporteur

Abstract. When large software product lines are engineered, a combined set of traditional techniques, such as inheritance, or design patterns, is likely to be used for implementing variability. In these techniques, the concept of feature, as a reusable unit, does not have a first-class representation at the implementation level. Further, an inappropriate choice of techniques becomes the source of variability inconsistencies between the domain and the implemented variabilities.

In this thesis, we study the diversity of the majority of variability implementation techniques and provide a catalog that covers an enriched set of them. Then, we propose a framework to explicitly capture and model, in a fragmented way, the variability implemented by several combined techniques into technical variability models. These models use variation points and variants, with their logical relation and binding time, to abstract the implementation techniques.

We show how to extend the framework to trace features with their respective implementation. In addition, we use this framework and provide a tooled approach to check the consistency of the implemented variability. Our method uses slicing to partially check the corresponding propositional formulas at the domain and implementation levels in case of 1–to–m mapping. It offers an early and automatic detection of inconsistencies.

As validation, we report on the implementation in Scala of the framework as an internal domain specific language, and of the consistency checking method. These implementations have been applied on a real feature-rich system and on three product line case studies, showing the feasibility of the proposed contributions.

Résumé. Durant le développement de grandes lignes de produits logiciels, un ensemble de techniques d'implémentation traditionnelles, comme l'héritage ou les patrons de conception, est utilisé pour implémenter la variabilité. La notion de feature, en tant qu'unité réutilisable, n'a alors pas de représentation de première classe dans le code, et un choix inapproprié de techniques entraîne des incohérences entre variabilités du domaine et de l'implémentation.

Dans cette thèse, nous étudions la diversité de la majorité des techniques d'implémentation de la variabilité, que nous organisons dans un catalogue étendu. Nous proposons un framework pour capturer et modéliser, de façon fragmentée, dans des modèles techniques de variabilité, la variabilité implémentée par plusieurs techniques combinées. Ces modèles utilisent les points de variation et les variantes, avec leur relation logique et leur moment de résolution, pour abstraire les techniques d'implémentation.

Nous montrons comment étendre le framework pour obtenir la traçabilité de feature avec leurs implémentations respectives. De plus, nous fournissons une approche outillée pour vérifier la cohérence de la variabilité implémentée. Notre méthode utilise du slicing pour vérifier partiellement les formules de logique propositionnelles correspondantes aux deux niveaux dans le cas de correspondance 1–m entre ces niveaux. Ceci permet d'obtenir une détection automatique et anticipée des incohérences.

Concernant la validation, le framework et la méthode de vérification ont été implémentés en Scala. Ces implémentations ont été appliquées à un vrai système hautement variable et à trois études de cas de lignes de produits.

Acknowledgments

I began this work in the name of God, the Beneficent, the Merciful, and I thank Him for every single progress in my life. Then, many people supported, encouraged, and motivated me, in one way or another, during this work. My sincere thanks to all of them! I shall mention here those who did much more than I expected from them.

I would like to express my nearly boundless gratitude to my supervisor Philippe Collet, who replied to my very first email to him and accepted to supervise me in this work, even without knowing me. I thank him for his unwavering support, for his readiness, and for all those constructive discussions, which kept me constantly engaged, especially during my research work in the distance. Literally, it was a complete privilege to be mentored by him, and hopefully, our cooperation will continue. So, I say to him: *"thank you for your continued support, thoughtful guidance, encouragement, the shared knowledge, and wisdom. Above all, thanks for your patience!"*

I wish to extend my deep appreciation to the reviewers, Olivier Barais and Tewfik Ziadi, who gave their time and attention to reviewing my manuscript and evaluating my work. I thank Mireille Blay-Fornarino and Laurence Duchien for accepting to be on the jury of my thesis. Also, special thanks go to all anonymous and known reviewers of our published work.

I would really like to thank the French Embassy in Kosovo for awarding me a six months per year scholarship for the Ph.D. studies for four years. Also, I thank Nexhat Humolli for saving my workplace at his company, where I worked part-time during the time that I was in Kosovo, for almost the first three years of my Ph.D.

Many thanks to the University of Nice Sophia Antipolis for supporting my Ph.D. in so many ways; especially, by making possible my participation in scientific conferences and for granting me a financial support for the three months on the last year. Also, thank you for being in the most amazing city.

Warm thanks to my family for being my biggest supporters, especially during these past four years. I cannot ever say this enough, but *"faleminderit për gjithçka"*! Also, I thank my closest friends (you know who you are) for their enthusiastic support.

Contents

List of Figures

List of Tables

Listings

List of Acronyms

vp	Variation Point
CNF	Conjuctive Normal Forme
CVL	Common Variability Language
DM	Decision Model
DSL	Domain Specific Language
FAMILIAR	FeAture Model scrIpt Language for manIpulation and Automatic Reasoning
FM	Feature Model
KCVL	CVL implementation in Kermeta
MTVM	Main Technical Variability Model
OVM	Orthogonal Variability Model
SPL	Software Product Line
TLs	Trace Links
TVM	Technical Variability Model
VM	Variability Model

Glossary

$[\![\phi_{FM}]\!]$ The set of feature configurations for the FM, ϕ_{FM}

$[\![\phi_{MTVM}]\!]$ The set of *vp*-s with variants configurations for the MTVM, ϕ_{MTVM}

$\mathcal{BT}$ The set of possible binding times of variation points

$\mathcal{EV}$ The evolution properties of variation points

$\mathcal{LG}$ The set of possible logical relations of *vp*-s with variants

$\mathcal{T}$ The set of possible traditional variability implementation techniques

$\mathcal{X}$ The possible types of variation points

ϕ_{FM} The propositional formula for the feature model (FM)

ϕ'_{FM} The sliced formula of feature model, ϕ_{FM}

ϕ_{MTVM} The propositional formula for the MTVM

ϕ_{TL} The propositional formula for the bidirectional trace links

ϕ'_{TL} The sliced formula of trace links, ϕ_{TL}

ϕ_{TVM_x} The propositional formula of a specific technical variability model (TVM)

core-code assets are those reusable code artifacts that form the basis for eliciting the software products in a software product line

MTVM The power set of all technical variability models (TVMs) in an SPL

TVMs The technical variability models for modeling the realized variability in core-code assets in a fragmented way

CHAPTER 1

Introduction

1.1 Context

1.1.1 Software Product Line Engineering

The *software product line* (SPL) engineering paradigm has emerged to foster methodological reuse between the related software products in a domain or an organization. By adopting it, an organization expects to achieve mass-customization for a family of products, such as decreasing their cost and their time to market while increasing their quality. An SPL is then usually defined as "*a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way* [Clements and Northrop, 2002; Northrop et al., 2007]. Originally, a feature is defined as *"a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems"* [1] [Kang et al., 1990] within a domain. Thus, features are used to describe and scope a set of related software products within a domain through defining their common aspects and their differences.

There are three models for adopting an SPL approach in an organization: proactive, when all products in the scoped domain are preplanned to be built; extractive, when from a set of legacy applications is build/extracted an SPL model; and reactive one, when in either case the SPL itself get evolved [Krueger, 2002a]. In addition, the whole development cycle of an SPL consists of (1) domain engineering, or development for-reuse, and (2) application engineering, or development with-reuse [Clements and Northrop, 2002; Kang et al., 1998; Pohl et al., 2005; Weiss et al., 1999]. Despite the SPL adaptation model, domain engineering and application engineering processes are highly interactive and can occur in any order, which are shown also as rotating cycles by Northrop et al. [2007].

Domain engineering and application engineering. Figure 1.1 shows the domain engineering and application engineering, as two main processes for the development of software product lines. The domain engineering process is characterized by the development of core assets, which represent those reusable artifacts and resources that form the basis for eliciting the single software products during the application engineering process. Core assets are made reusable by modeling and realizing what is common and what is going to vary (*i.e.*, the common and

[1] Based also on the American Heritage [1985]

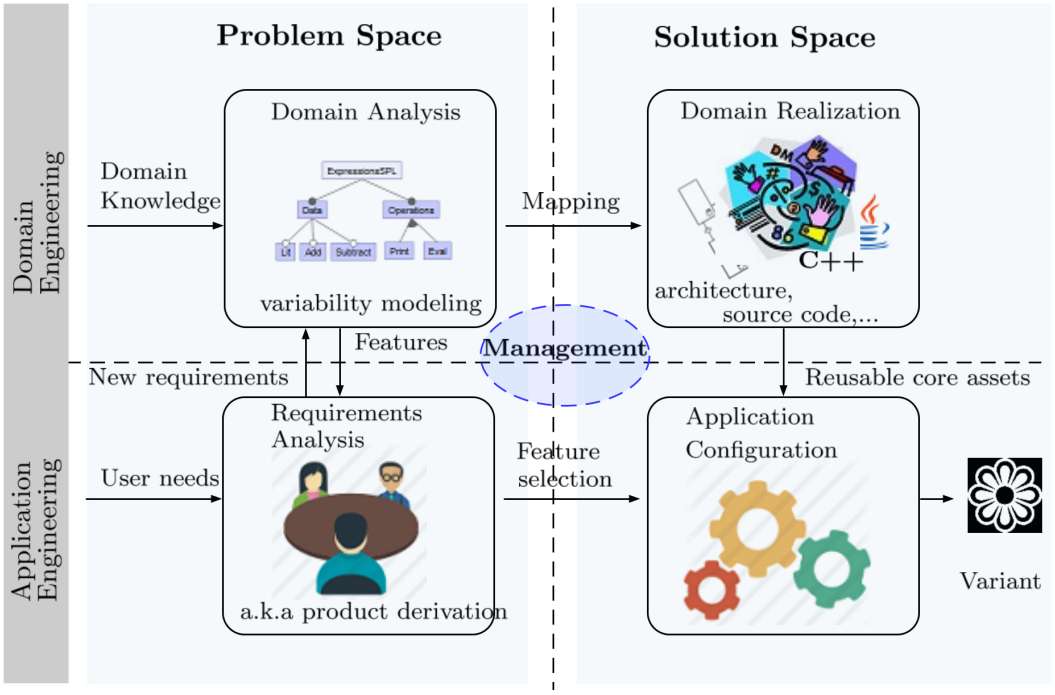


Figure 1.1: The software product line engineering processes, with the problem space and solution space separation for software assets (adapted from Czarnecki [2005]; Pohl et al. [2005, Ch. 2])

variable features) between the related products in a methodological way. Variability can be part of any core assets, which "*often include, but are not limited to, the architecture, reusable software components, domain models, requirements statements, documentation,...*" [Clements and Northrop, 2002; Northrop et al., 2007]. In this thesis, in the focus is the variability of core assets at the implementation level, that is, the core-code assets.

Problem space and solution space. Orthogonal with the domain engineering and application engineering is the problem space and solution space dimensions for all software assets [Czarnecki, 2005; Czarnecki and Eisenecker, 2000, Ch. 3], as is shown in Figure 1.1. While in problem space all valid feature combinations and requirements of software products are specified, in solution space these specifications are realized in concrete software systems, such as their reusable architecture, detailed design, or implementation.

1.1.2 Software Variability Management

An essential activity of the whole product line development is the management activity (*cf.* Figure 1.1). It can be (1) organizational management, which represents the *"authority responsible for the ultimate success or failure of the product line effort"* [Northrop et al., 2007], and (2) technical management that deals with the

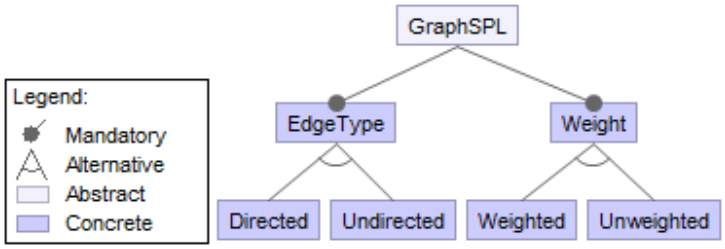


Figure 1.2: Excerpt of the feature model for the Graph product line

development of core assets and product derivation activities.

Technical management has several practice areas, such as configuration management, make/buy/mine/commission analysis, scoping, tool support, etc. [Clements and Northrop, 2002]. From them, the well-known configuration management comes from traditional single software system development that is extended and represents the variation management in software product lines [Krueger, 2002b]. According to Krueger [2002b] and Clements and Northrop [2002], variation management is multi-dimensional and basically deals with (1) the variability in space, that is, the management of the variability of core assets at a fixed moment in time, and (2) the variability in time, that is, the management of versions of core assets during the time (as in single software development), maintenance (evolution of domain), and resolution of variability during a product derivation with different binding times (*cf.* Section 3.1.1). Thus, variability in space, known as software variability management, is specific only to product lines but, it interferes with the variability in time regarding the binding time, during each product derivation.

In realistic SPLs, where the variability is extensive, a crucial issue is the ability to manage this variability in different core assets and among different abstraction levels [Bosch et al., 2001]. An important aspect of this activity is then the ability to identify or capture and trace a variable unit, commonly known as a feature, in core assets of an SPL. In particular, from Figure 1.1, the specified domain variability in the problem space needs to be mapped to their respective realization, such as in architecture assets, and core-code assets, in the solution space.

More concretely, the specified domain variability in terms of features is commonly captured in a variability model (VM), such as a feature model (FM) [Kang et al., 1990] (*cf.* Figure 1.1). The 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. Semantically, an FM represents the valid software products (*i.e.*, the feature configurations) within an SPL. In Figure 1.2 is given an excerpt of the feature model for the Graph product line [Lopez-Herrejon and Batory, 2001] [2], which is well-known in the academia. Whereas, one of its possible realizations at the

[2] A complete feature model for Graph product line is given in Section 2.1

implementation level is also given in Listing 1.1.

In this thesis, we address specifically the variability management activities between the specified variability in problem space, for example, the FM in Figure 1.2, and core-code assets in solution space, for example, the implemented variability in Listing 1.1, during the domain engineering process. In this context, we have identified several challenges to be addressed.

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

Listing 1.1: Graph PL implementation using two different techniques (parameters and strategy pattern), in the Scala language

1.2 Overview of Challenges and Contributions

We now determine the addressed challenges, then we summarize our contributions and we give an outline of this dissertation.

1.2.1 Challenges

In the described context, there are several challenges to be addressed regarding the modeling and management of variability in core-code assets. These challenges are grouped into three main challenges, as in the following.

Challenge A. Variability modeling at the implementation level

A1. *Understanding the diversity of variability implementation techniques.*

Variability can be implemented by different variability implementation techniques, such as inheritance, generic types, design patterns (see Section 2.2.3). Depending from the used technique, the units that vary can be diverse at the implementation level. For example, some variability can be implemented at the class level (*e.g.*, using the technique of inheritance), or at method level (*e.g.*, using the technique of overloading). Also, variability can be resolved earlier (*e.g.*, at compile time) or later (*e.g.*, at runtime) during the product derivation. This implies that, when traditional techniques are used (*cf.* Section 3.1.3), the variability of core-code assets is realized by using a combined set of techniques (*e.g.*, inheritance, overriding, design patterns). In this case, choosing and combining the right techniques to implement some variability of an SPL domain is not trivial. Whereas, when variability is implemented by using a single technique, such as preprocessors in C, the implementation is more straightforward with the whole variability being implemented by using the preprocessor directives.

For evaluating and choosing a technique to address some domain variability at the implementation level, pieces of advice can be found in studies, catalogs, or taxonomies (*e.g.*, Apel et al. [2013]; Bachmann and Clements [2005]; Coplien [1999]; Gacek and Anastasopoules [2001]; Muthig and Patzke [2003]; Patzke and Muthig [2002]; Svahnberg et al. [2005]), but their studied techniques are evaluated by different subsets of criteria that remain scattered among different research works.

Moreover, towards an approach for documenting the variability at the implementation level, it is necessary to understand the diversity of variability implementation techniques, analysed by the same set of properties. To the best of our knowledge, there is no up-to-date classification and catalog for variability implementation techniques. It is also expected that an enriched catalog would guide the deveopers for choosing the right techniques in the majority of cases.

A2. *Capturing and modeling the implemented variability when a combined set of traditional variability implementation techniques is used.*

When traditional variability implementation techniques are used to implement some variability, such as inheritance, generic types, or design patterns, the code is not shaped in terms of features from the domain level. Usually, the implemented variability is modeled in terms of variation points (*vp*-s) with variants [Coplien, 1999; Czarnecki et al., 2012; Jacobson et al., 1997; Schmid and John, 2004] (they are defined in Section 2.2.1). Further, in realistic product lines, variability is implemented by using a combined set of such traditional techniques, meaning that the *vp*-s with variants are diverse (*e.g.*, with different binding times, Section 3.1) compared to the case when a single technique is used to implement the whole variability, for example, by using the preprocessors in C the whole variability is resolved during the compile time in case of a product derivation.

Consequently, the capturing and modeling of the implemented variability in core-code assets when a combined set of traditional variability implementation techniques are used is a challenging task that should be facilitated. Moreover, this modeling phase is essential for managing the variability of core-code assets.

A3. *Separating the development dimension and variability dimension while maintaining their corresponding relationship (a.k.a. consistency).*

Variation points with variants (defined in Section 2.2.1) are not a by-product of the traditional variability implementation techniques [Bosch et al., 2001; Sinnema et al., 2004a], meaning that the places in core-code assets where some variability is realized are implicit. Therefore, the implemented variability needs to be captured and modeled in some way, in order to be managed. The variability of core-code assets can become explicit (1) by annotating the places where the variability happens directly in code, for example, using preprocessor directives [Tartler et al., 2012], tags [Heymans et al., 2012], or a visual representation [Kästner, 2012], (2) abstracting and documenting the variability separately from the code, for example, using decisions for variation points with variants in a decision model [Schmid and John, 2003], or (3) implementing/factorizing the code in physically separated modules, for example, in feature modules [Apel et al., 2013]. Here we do not consider the reverse engineering approaches, such as feature location techniques for migrating some related software products in an SPL approach.

Specifically, the development dimension, or the functionality in code, and the variability dimension, or the variability-related issues, can stay either mixed or separated. To ease the variability management, as mentioned by Berg et al. [2005]; John et al. [2007]; Muthig and Atkinson [2002], we aim at keeping separated the variability and development dimensions. However, when they are separated, the consistency between the variability dimension and the development dimension becomes an issue as it needs to be maintained; therefore, it is one of our challenges.

A4. *Modeling the implemented variability in a fragmented way.*

In realistic product lines, the number of features in a feature model at the specification level may be considerable (*e.g.*, [Tartler et al., 2012]). On the other hand, the variation points with variants at the implementation level are a refinement of features from the domain level [Hunt and McGregor, 2006]. Consequently, the amount of variability at the implementation level is expected to be even larger than the variability at the domain level [Pohl et al., 2005, Ch. 4]. The capturing and modeling of the implemented variability at once and in one place may then be harder and will weaken, instead of supporting, the management of variability. For such reason, the documentation of the implemented variability in a fragmented way can be practical and helpful, as is suggested by the variability-aware module system [Kästner et al., 2012] where each implemented *module* defines its own variability model.

When traditional programming paradigms are used (such as the object-

oriented or functional ones) there is no concept of *modules* that can be considered as fragments with variability, but variability may happen, for example, within a package, file, or class. Therefore, we aim at modeling the variability of core-code assets in a fragmented way, where a fragment can be flexible, such as a package, file, or class with some variability that is worth to be modeled separately.

Challenge B. Variability traceability

B. *Supporting the variability traceability between the specification and implementation levels.*

Originally, in the FORM method [Ch. 8 Capilla et al., 2013; Kang et al., 1998], the need to model separately the variability at the specification and realization levels is already present. While the variability at the realization level is more about the software variability, the specification one represents the variability between the software products themselves within an SPL [Metzger et al., 2007]. In most variability management approaches, it is up to the reader to understand whether a variability model is used to describe the variability at the specification or realization levels [Metzger and Heymans, 2007]. Moreover, the mapping of features to variable units in implementation is mostly 1 – to –1, for example, between features and preprocessor directives in C [Le et al., 2013; Lotufo et al., 2010; Tartler et al., 2012], although a directive can be scattered in core-code assets. In such cases, the FM is used to model only the implemented variability, or from both levels into a single model. In reality, the mapping of features from the specification level to their implementation is n – to – m [Pohl et al., 2005, Ch.4], and also the variability abstractions in both levels may use different names.

Therefore, an approach for tracing the specified and implemented variabilities, when a combined set of techniques are used, is needed. Moreover, these trace links make possible the variability management and can also be used for other purposes, for example, to resolve the implemented variability during a product derivation, or to check its consistency.

Challenge C. Consistency checking of variability

C1. *Checking the consistency of variability between the specification and implementation levels.*

According to a recent survey about consistency checking in SPL engineering, by Santos et al. [2015], there are three major approaches to address consistency issues: (1) within the variability models (*i.e.*, FMs), (2) between the FM and other software models, or (3) between the FM and its implementation (*i.e.*, core-code assets). These also correspond to the locations where inconsistencies can happen, as mentioned by Vierhauser et al. [2010].

Currently, the support for checking variability consistency between an FM and core-code assets is limited. This means that the specified variability in terms of

features at the domain level and their respective implementation, using the traditional techniques, in core-code assets must represent the same number of software products.

Checking the consistency between the implemented and specified variabilities is also a challenging task, as the mapping of the specified features in an FM to the core-code assets is n – to – m. But, we expect to check their variability consistency within the context of their trace links.

C2. *Checking the consistency of variability when a combined set of traditional variability implementation techniques is used.*

The existing approaches are mostly conceived for resolving inconsistencies within a specific software, such as the Linux kernel [Lotufo et al., 2010; Tartler et al., 2012], or when the variability is implemented by a single variability implementation technique, for instance, using preprocessors in C [Le et al., 2013]. However, in realistic SPL settings, variability is implemented by using a combined set of traditional techniques, such as inheritance, overloading, design patterns. An inappropriate choice and combination of such techniques become the source of variability inconsistencies that cannot be detected by existing approaches. In this work, we consider that the variability of core-code assets is implemented by using a combined set of traditional techniques.

In case that some variability is implemented by using an improper technique, several inconsistencies may appear, for example, when an alternative logical relation between features in a variability model is implemented by a variation point, with an Or logical relation between its variants. In such case, the number of possible products at the specification level is inconsistent with the possible products that can be derived from the core-code assets. Therefore, in complement to Challenge C1, a consistency checking approach should be able to check whether the right technique to implement some variability is used.

C3. *Achieving an early detection of variability inconsistencies.*

The consistency of some specified variability can be checked only after it is realized in core-code assets, but not necessarily only after the whole specified variability is addressed. Commonly, some variability is deferred to be implemented later or during the application engineering phase. In addition, it becomes harder to fix the inconsistencies after all of them are shown at the same time at the end [Vierhauser et al., 2010]. In particular, it has been shown that trying to change the implementation technique for some addressed variability, only after the whole SPL is implemented, can be very costly [Bachmann and Clements, 2005]. Therefore, an approach for detecting earlier the variability inconsistencies is needed, for example, to be able to select a single or a group of variation points with variants and to check them against the specified features at the domain level early during the development process. Typically, we could expect that the earlier a variability inconsistency is identified, the cheaper becomes the fix.

1.2.2 Contributions

In this dissertation, we address these challenges by offering a detailed and tooled framework for modeling and managing the variability of core-code assets. Specifically, our contributions can be summarized as follows.

A catalog of variability implementation techniques. First, we study the diversity for the majority of the existing variability implementation techniques and provide a unified set of comparison criteria for them. Then, we build a catalog that covers an enriched set of techniques, which are compared by using the same set of criteria.

A framework for modeling and tracing the variability of core-code assets. We used the analysed diversity of variability implementation techniques to create our framework for capturing and modeling the variability of core-code assets. Specifically, the framework supports the capturing and modeling of variability when a combined set of traditional variability implementation techniques are used at the implementation level, which exposes a form of what we define as imperfectly modular variability. Moreover, the framework is based on capturing or abstracting the variability implementation techniques, with their characteristic properties, in terms of variation points with variants, which are used for modeling the variability of core-code assets into so-called *technical variability models*, in a fragmented way.

In addition, the framework supports traceability, by defining the n–to–m trace links between the features in a feature model at the domain level and the variation points with variants in technical variability models at the implementation level.

A method for checking the consistency of the implemented variability. Through modeling the variability in a fragmented way, we provide a method for checking the consistency between the specified and implemented variabilities, as earlier during the realization of core-code assets. Concretely, the method uses a slicing technique over the variability models to partially check the corresponding propositional formulas at the domain and implementation levels in case of their 1–to–m mapping.

Tool support and validation. Finally, we present a concrete implementation of our framework as an internal DSL in the Scala language. Further, we use this DSL for implementing the method for variability consistency checking. Both these implementations are applied on a real feature-rich system and on three product line case studies, showing the feasibility of the proposed contributions.

1.2.3 Outline

In the following, Chapter 2 gives a background for the used concepts in this thesis, and the state of the art in variability management, which is not specific to core-code assets. An introduction on domain specific languages (DSL), as a mean for

building tool support in variability modeling is also given. Our contribution is then presented, organized in three parts.

Part I: Design of variability models at the implementation level. In this part, Chapter 3 presents the studied diversity of variability implementation techniques, an updated catalog of them, and shows a way for using the catalog during the evaluation and choice of techniques. In Chapter 4 we describe our framework for capturing and modeling the imperfectly modular variability (defined in Section 4.1.1) at the implementation level. Also, the importance of capturing the variability implementation techniques during a reverse engineering process.

Part II: Usage of technical variability models. Chapter 5 proposes an extension of the framework, for tracing the specified and implemented variabilities under their n–to–m trace links. Chapter 6 describes the method for checking the consistency of the variabilities between the specification and implementation levels.

Part III: A tool support. In Chapter 7 an implementation of the framework, as a domain specific language, is presented, as well as its usage in four case studies. Similarly, Chapter 8 presents a concrete implementation of the consistency checking method and reports on it application in four case studies.

Finally, Chapter 9 concludes the thesis by summarizing the addressed challenges, and gives some further perspectives and our future work.

CHAPTER 2

Background and State of the Art

This chapter is an introduction to the main topics and related approaches of this thesis. Specifically, we introduce variability modeling in Section 2.1, variability realization at the implementation level in Section 2.2, and variability traceability as a key aspect for variability management of core-code assets in Section 2.3. Then, we discuss automated analysis of feature models in Section 2.4, as a means for automatic consistency checking of variability, Section 2.5 is dedicated to domain specific languages as a support for tooling the approach proposed in this thesis.

2.1 Variability Modeling

Since its first presentation by Kang et al. [1990], feature modeling (FM) is widely adopted as a means for documenting the commonalities and variabilities in terms of features of software products in an SPL. In different approaches or usage contexts, a feature is defined. Appendix A gives 20 of its definitions. Similarly, the original notion of feature model has been extended over time. Capilla et al. [2013, Ch. 2] compares 11 FM variations (by Benavides et al. [2005]; Czarnecki et al. [2002]; Czarnecki and Eisenecker [2000]; Czarnecki et al. [2004]; Eriksson et al. [2005]; Griss et al. [1998]; Hein et al. [2000]; Kang et al. [1998]; Riebisch et al. [2002]; Van Gurp et al. [2001]), and a more holistic approach to feature modeling is also available [Lee et al., 2014].

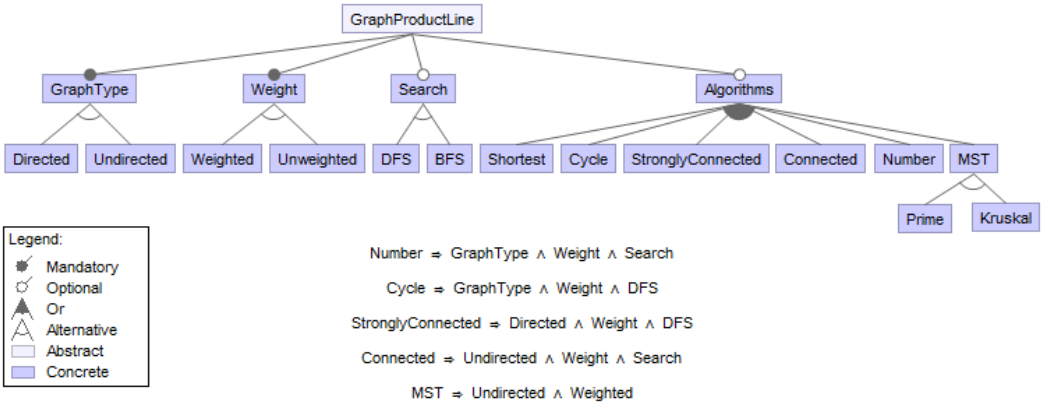


Figure 2.1: The feature model of the Graph product line

As an illustration, a concrete feature model, the Graph PL, is given in Figure 2.1. This SPL is quite well understood and used by the community [Lopez-Herrejon and Batory, 2001]. It is a hierarchically arranged set of features with the main

feature, `GraphProductLine`, representing conceptually the SPL domain. It has two compound mandatory features, `GraphType` and `Weight`, with their alternative variant features, «`Directed`, `Undirected`» and «`Weighted`, `Unweighted`», respectively. Also, it has two compound optional features, `Search` and `Algorithms`. The first one has two alternative variant features, «`DFS`, `BFS`», and the second one has six features in an Or relation, «`Shortest`, `Cycle`, `StronglyConnected`, `Connected`, `Number`, and `MST` with alternative features `Prime` and `Kruskal`». Features in an FM can have cross-tree constraints, which are commonly expressed in propositional logic. This Graph PL has five cross-tree constraints, such as `Number` $\rightarrow$ (`GraphType` $\wedge$ `Weight` $\wedge$ `Search`), meaning that the selection of feature `Number` requires features `GraphType`, `Weight` and `Search` to be selected.

Thus, the software variability of an SPL is documented in a variability model (VM), which is commonly expressed as a feature model (FM) [Kang et al., 1990], but it can also be a decision model (DM) (first introduced by the Synthesis method [Corporation, 1993; Schmid et al., 2011]), an orthogonal variability model (OVM) [Pohl et al., 2005], or FMs complemented by a Domain Specific Language (DSL) [Voelter and Visser, 2011]. The FM and DM were introduced almost at the same time and used quite equally [Czarnecki et al., 2012]. According to the Synthesis method, *"a Decision Model identifies, for each work product family, the application engineering requirements and engineering decisions that determine how members of the work product family can vary"* [Corporation, 1993]. Overall, any variability model (FM, DM, etc.) can be considered as a decision model whenever it is used for taking the decisions during the product derivation [Capilla et al., 2013, Ch. 20]. While the variability in an FM has a graphical representation, in a DM it has a tabular representation [Dhungana and Grünbacher, 2008; Forster et al., 2008; Muthig and Atkinson, 2002; Schmid and John, 2004]. Besides, there are also formal approaches for transforming an FM to a DM and vice versa [El-Sharkawy et al., 2012].

Problem space and solution space variability. Similar to the problem and solution space for core assets, given in Section 1.1.1 [Czarnecki, 2005; Czarnecki and Eisenecker, 2000; Turner et al., 1999], the problem space and solution space for variability are distinguished [Ch. 2 Capilla et al., 2013; Lee et al., 2014]. This variability space is given in Figure 2.2, which shows that, depending on the viewpoints, features can be in problem variability space (*i.e.*, goal/objective, usage context, and quality attribute features) or solution variability space (*i.e.*, capability, operating environment, and design features). The approach by Metzger et al. [2007] recognizes these two variability spaces as the product line (PL) variability and software variability.

There are few approaches that distinguish clearly these two variability spaces. Their difference is introduced earlier in the FORM method [Kang et al., 1998] then, among the first, in the variability management approach by Becker [2003a,b]. In the majority of other approaches, it is up to the reader to understand when an FM represents the PL variability or the software variability [Metzger and Heymans,

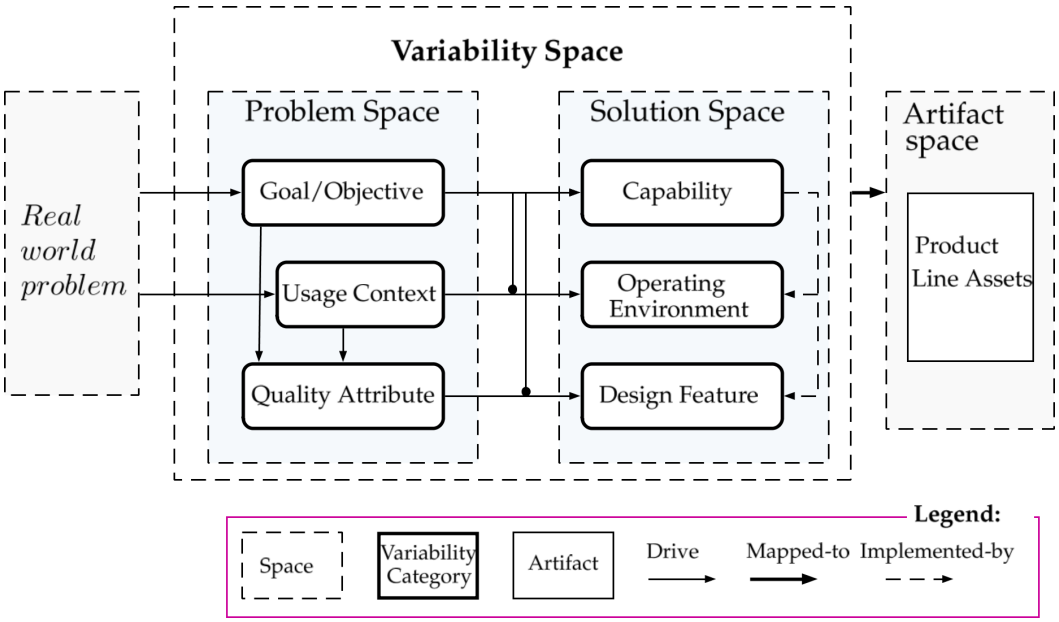


Figure 2.2: Problem space and solution space for variability modeling (borrowed from Capilla et al. [2013, Ch. 2])

2007].

According to a study by Czarnecki et al. [2012], an FM is mainly used at the specification level for scoping the software products within an SPL in terms of features, whereas a DM is used closer to the realization level (*i.e.*, architecture, implementation, etc.) for supporting the resolution of variation points (*vp*-s) with variants in form of decisions (their definition is given in Section 2.2.1). In our work, we consider specifically the implementation level (*i.e.*, the core-code assets).

2.2 Variability Implementation

2.2.1 Variable Parts in Core-Code Assets

In realistic SPL settings, the implementation of variability in core-code assets complies, mostly, to a commonality and variability approach, despite the programming paradigm (*e.g.*, object-oriented, or functional) [Coplien, 1999]. Specifically, a domain is decomposed into subdomains, then within each subdomain the commonality is factorized from the variability that is used to differentiate the software products within the domain.

The core-code assets consist of three parts: the core, commonalities, and variabilities. The core part is what remains of the system in the absence of any particular feature [Turner et al., 1999], that is, the assets that are included in any software product of the SPL. A commonality is a common part for 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 [Coplien, 1999]), except when it represents some optional variability. The variant parts are used to distinguish the software products in the domain. A subdomain can have more than one common and variant part. The core with the commonalities and variabilities of all subdomains constitute the wholeness of core-code assets in an SPL.

The commonalities and variabilities in core-code assets are commonly abstracted in terms of variation points (*vp*-s) with variants, as solution oriented abstractions. Unlike features in problem space, *vp*-s with variants are related to concrete elements in core-code assets. Originally, *"a variation point identifies one or more locations at which the variation will occur"* [Jacobson et al., 1997]. Respectively, variation points are known as a manifestation of variability in architecture, in design, and, eventually, in implementation [Fritsch et al., 2002]. The way that a variation point is going to vary is expressed by its variants. Over time, several other definitions of *vp*-s with variants have appeared (see 16 of their definitions in Appendix A).

The *variable part* is like an organizing container. It contains the location where some variability happens in core-code assets (*i.e.*, the variation point), the variants, and the used technique to implement them.

2.2.2 Variability Abstractions

In order to understand in terms of what abstractions we should capture some implemented variability, we analyse in this section the usage of features and variation points (*vp*-s) with variants in the literature. In the following, we show their definitions as problem or solution oriented abstractions, the diversity of their terminology, and the different meanings of a *vp* concept.

Problem or solution oriented abstractions. To understand whether the concept of features or *vp*-s with variants are used solely as problem oriented or solution oriented variability abstractions, we analysed their definitions in 34 approaches (see Appendix A).

We compared 20 definitions of the feature concept, also analysed by Classen et al. [2008], Patzke et al. [2011], and Apel et al. [2013, Ch. 2]. From them, eleven definitions consider a feature as a problem oriented abstraction (including the original definition by Kang et al. [1990]), four of them as a solution oriented abstraction (including the extension of the original definition [Kang et al., 1998]), and five are general definitions. Apel et al. [2013] give explicitly a broader (*i.e.*, more inclusive) definition of a feature by subsuming the variability of problem and solution spaces.

Similarly, we analysed 14 definitions of the variation point. Twelve of them emphasize that a *vp* is a solution oriented concept (including the original definition by Jacobson et al. [1997]), a single one uses the notion of *variable part* in solution space instead of the variation point [Bachmann and Clements, 2005], and the last

Figure 2.3: The *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.

one defines it as a problem oriented concept [Pohl et al., 2005]. In a subsequent approach [see Metzger et al., 2007], Pohl et al. [2005] make even clearer the usage of *vp*-s only as domain abstractions. Although, in the standard [ISO/IEC 26550:2015, 2015], by Pohl et al. [2005], the feature concept is also defined and used as a problem oriented abstraction.

In two definitions of *vp*, the abstractions of feature and variation point are used together. The first one considers that an FM is also a core asset and a *vp* is used to point the varying places in the FM [Czarnecki and Eisenecker, 2000]. The second definition, by Hunt and McGregor [2006], stresses that *vp*-s are refinements of features.

Naming. In the literature, features and *vp*-s with variants are sometimes called differently. Mostly, the alternative names for features reflect their usage as problem or solution oriented abstractions. For example, a feature is known as a delivery unit, configuration unit, reuse unit, semantic unit (in problem space), or as syntactic unit [Kästner et al., 2011], design decision (design feature), changeable decision (alternative design feature) [Capilla et al., 2013], variation point feature [Griss, 2000], or variant feature (in solution space). On the other hand, a *vp* is known as a design decision, delayed design decision [Van Gurp et al., 2001], spot [Becker, 2003a], extension point [Apel et al., 2013], or generic element [Becker, 2003b].

Sometimes, the notions of feature and *vp* with variants are used alternatively. For example, in problem space, the node at which varying features are attached (*i.e.*, the compound feature) is known as *vp* [Czarnecki and Eisenecker, 2000; Griss, 2000]. In solution space, a design feature is the *vp* and an alternative design feature (or variant feature) is the variant [Capilla et al., 2013].

Semantics of variability abstractions. Variability abstractions in problem space, such as features, are mere names which are not related to any variability realization technique or core asset in realization levels. On the contrary, *vp*-s and solution space features are related to a concrete variability implementation technique or a varying element within core-code assets, that is, they concretely point the elements of core-code assets that are going to vary. For example, a solution space feature can point to a file [Beuche et al., 2004], a feature module in Feature-Oriented programming [Apel et al., 2013], or a macro in C using its preprocessors [Kästner, 2012]. Similarly, a *vp* can be a (super) class, an interface, a parameter [Jacobson et al., 1997], or an #ifdef in C [Schmid and John, 2004].

Mostly, a *vp* is understood as (1) a *handle* for relating the specified variability to its respective varying element(s) in core assets on realization levels [CVL, 2012], (2) an abstraction of a location or point that vary in core-code assets (*cf.* Figure 2.3a) [Jacobson et al., 1997], (3) a variable part (*cf.* Figure 2.3b) [Bachmann and Clements, 2005], (4) an abstraction of a variability implementation technique [Capilla et al., 2013, pg. 48], or (5) as the symmetry (*i.e.*, commonality invariance) and symmetry breaking places in software [Coplien and Zhao, 2000]. Regarding the first meaning, in a prototype implementation of the CVL, KCVL [Barais et al., 2013; KCVL, 2015], is stated that "*Using variation points, it is possible to express and manipulate the links between the variability abstraction model and the base model*". Although the *vp* is used here as a link, it also supports some semantics for reassigning references or excluding its other related model elements from the base model. The third, fourth, and fifth meanings are related, as the commonality is factorized by using a variability implementation technique, whereas the technique represents the mechanism to accommodate the variants and thus to resolve the variability. In addition, most of the traditional variability implementation techniques have the properties of the symmetry or symmetry breaking in software [Castellani, 2003; Coplien and Zhao, 2000; Rosen, 1995; Zhao and Coplien, 2003, 2002]. In geometry, the symmetry is defined as the immunity to a possible change [Rosen, 1995]. Whereas, in software, a symmetry can be identified between a class in object-oriented (which can be a *vp*) and its objects (its variants). Specifically, a class establishes an invariance relationship between the class and its objects, which are analogue (*i.e.*, changed but still the same) with respect to the class structure [Coplien and Zhao, 2000].

Independent from the implementation technique. There are approaches for managing the implemented variability in terms of features (*e.g.*, [Apel et al., 2013; Heymans et al., 2012; Kästner and Apel, 2009]) and those in terms of variation points with variants (*e.g.*, [Becker, 2003b; Capilla et al., 2013; Jacobson et al., 1997; Pohl et al., 2005; Schmid and John, 2004]). Thereby, in the literature, both abstractions of feature and *vp* are used almost equally and sometimes in combination.

We observed that the differences between a feature and a *vp* oriented implementation are not related to a specific variability implementation technique. In most cases, a *vp* with variants are implemented by a traditional technique such as inheritance, parameterization, etc. On the other hand, there are approaches where a feature represents a class, a file [Beuche et al., 2004] or a plugin. As another example, for Thüm et al. [2014] preprocessors are used as a technique for implementing features, whereas by Schmid and John [2004] they are used as a technique for implementing *vp*-s. But, still, the concepts of features and *vp*-s with variants are not the same thing that is just called differently in different approaches.

An essential difference. Basically, the concept of a feature is used to annotate or put in a separate module all those lines of code that may belong to a specific fea-

ture (*e.g.*, [Apel et al., 2013; Heymans et al., 2012; Kästner, 2010; Lee et al., 2000]). Whereas, a *vp* is used as a way for abstracting a varying place without making a commitment about which lines of code are exactly in or out the *vp* concept. Specifically, in our understanding, a *vp* with variants represents some entities that have a fuzzy border in core-code assets.

In this work, unless explicitly specified, we consider that variation points with variants are solution oriented abstractions that we use for capturing and managing the variability in core-code assets, whereas features as problem oriented abstractions.

2.2.3 Variability Implementation Techniques

The techniques that are used for realizing the variability in an SPL are called variability realization techniques [Capilla et al., 2013; Svahnberg et al., 2005], variability mechanisms [Bachmann and Clements, 2005; Jacobson et al., 1997; Muthig and Patzke, 2003], or variability implementation techniques [Fritsch et al., 2002]. *Variability realization technique* is a general term used for techniques acting at the architecture, design, or code level; whereas, the term *variability implementation technique* will be only used for techniques at the code. All variability implementation techniques came from several programming paradigms and are supported by different constructs in different programming languages, which in turn offer different properties.

An SPL is structured around a set of features of several kinds, which represent different functional or non-functional software products' requirements. These features have to be properly realized and their reflection to *vp*-s with variants is manifold in architecture, design, and especially in implementation, forming an *n-to-m* relation [Gacek and Anastasopoules, 2001; Metzger and Pohl, 2014]. Each *vp* is associated with one or more variability implementation techniques (*e.g.*, when several implemented versions of the same *vp* are needed for achieving different binding times [Dolstra et al., 2003a,b; Rosenmüller et al., 2011]) and vice versa. In principle, one *vp* is associated with only one technique, whereas the same technique can be used to implement several *vp*-s within a domain. Examples of variability implementation techniques are inheritance in object orientation, preprocessors, feature modules or some design patterns. In Listing 1.1 (Section 1.1.2) we show the usage of parameters and strategy pattern in the Scala language as variability implementation techniques for implementing the two compound features, `GraphType` and `Weight`, for the Graph PL (*cf.* Figure 2.1).

2.3 Variability Management Approaches

Variability management is a complex activity, first because variability is realized in different types of core assets (*e.g.*, requirements, architecture, code, etc.) and then, within each of these abstraction levels, different variability realization techniques

can be used. For example, at the requirements or architecture level, various extensions of the UML (Unified Modeling Language) are proposed by using notes or stereotypes [Clauß and Jena, 2001; Ziadi et al., 2003; Ziadi and Jézéquel, 2006]. Whereas, at the implementation level, with a textual representation, different textual or language annotations [Heymans et al., 2012; Tartler et al., 2012], language constructs [Apel et al., 2013; Schaefer et al., 2010], or visualization tools [Kästner et al., 2008b] are used. Therefore, modeling and tracing the variability of these core assets is a primary step for the variability management during the SPL engineering [Berg et al., 2005].

2.3.1 Variability Traceability

The CoEST [1] [Cleland-Huang et al., 2012] 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"*.

The concept of traceability is already used for different reasons in single (traditional) software engineering, mostly to trace requirements [Winkler and Pilgrim, 2010]. Its meaning and techniques need to be enhanced for use in SPL engineering, because the relevant entities that are required to be traced and the semantics of trace links in SPL engineering and in traditional software engineering are quite different.

Traceability dimensions. Principally, four dimensions of traceability are distinguished among the literature: horizontal (intra), vertical (inter), versioning, and variability traceability [Anquetil et al., 2008, 2010; Cleland-Huang et al., 2012; Krueger, 2002b; Schmid and John, 2004; Winkler and Pilgrim, 2010]. The first three dimensions are similar as in the traditional software engineering, whereas variability traceability is specific to SPL engineering and deals with variability in space (*i.e.*, the variability of core assets in a specific moment of time).

Variability traceability *"is dealt by capturing variability information explicitly and modeling the dependencies and relationships separate from other development artifacts"* [Berg et al., 2005]. It has two other dimensions: tracing the realized variability (*i.e.*, the variability from the problem space to the solution space), and tracing the resolved variability (*i.e.*, the software artifacts between application engineering and domain engineering) [Anquetil et al., 2010; Krueger, 2002b; Pohl et al., 2005]. They are shown as *realize/implement* and *use* trace links, respectively, in Figure 2.4. Further, the term *variability traceability* is used mostly for the first dimension. The horizontal and vertical traceability came from the single system engineering, but can also have the meaning of tracing the variability in an SPL among the artifacts within the same abstraction level or between different abstraction levels, respectively. And, the versioning traceability conducts the evolution of a software artifact

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

[2] Sometimes we will use the term **trace** as a *verb*, see its definition by Cleland-Huang et al. [2012].

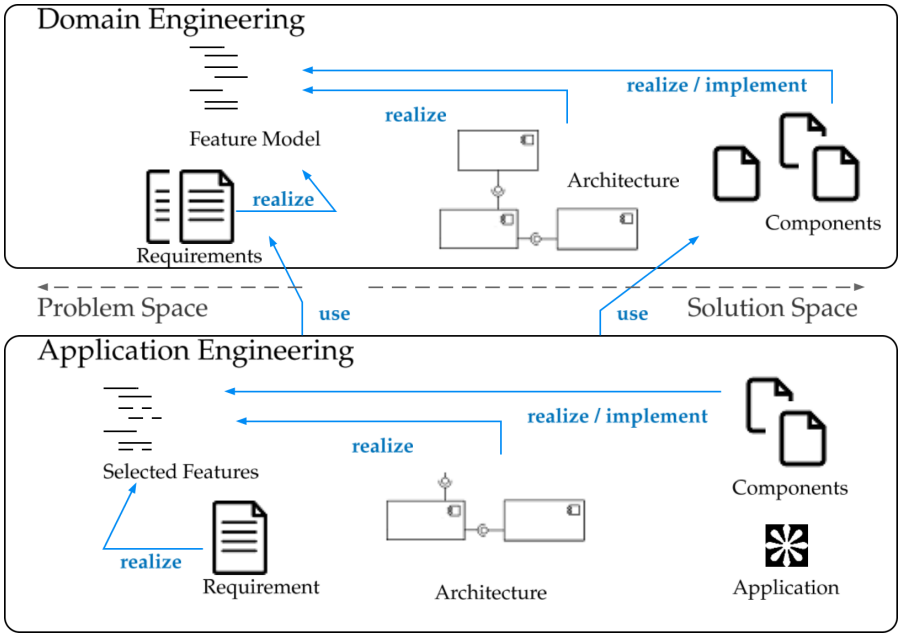


Figure 2.4: Illustration of two variability traceability dimensions, *realize/implement* and *use* - the blue links (adapted from Anquetil et al. [2010])

(in traditional engineering) and/or the evolution of variability (with the variability in time, in SPL engineering).

All these traceability dimensions are orthogonal to each other: (1) they may interact, such that, the variability traceability links may be subject to evolution over time, and (2) can be applied in a hierarchical way, that is, versioning traceability can be applied to variability, inter, and intra traceability, then variability traceability can be applied to inter and intra traceability [Anquetil et al., 2008].

Usage of trace links. A generic traceability process model comprises the creation, usage, and maintenance of trace links [Gotel et al., 2012], as the trace links are established to meet a specific usage purpose during a software engineering process and need to be maintained. In SPL engineering, variability traceability can be established and used for different reasons, and by different stakeholders [Anquetil et al., 2010; Cleland-Huang et al., 2012, 2014]. It is mainly used for (semi)automating different processes in SPL engineering, for example, for resolving the variability during product derivation [Deelstra et al., 2004, 2005], evolving, checking consistency, addressing, or comprehending variability.

2.3.2 Modeling and Tracing the Variability of Core Assets

For identifying and modeling the variability in different core assets, several approaches are available, which are specific to a type of core assets [*e.g.*, Clauß, 2001; Czarnecki and Antkiewicz, 2005; Gomaa, 2005; Heymans et al., 2012; Ziadi et al.,

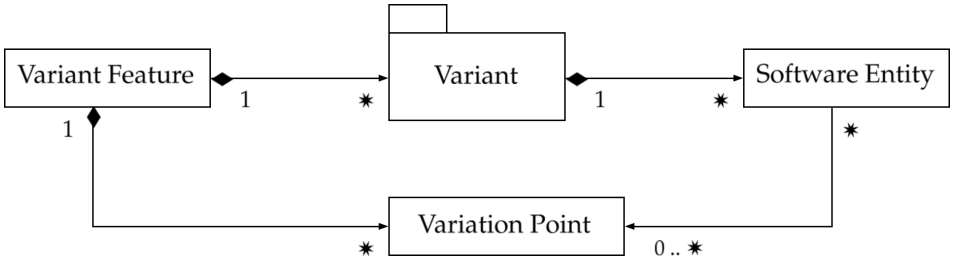


Figure 2.5: The relation between the concepts of feature, variation point with variants and software entities (borrowed from Svahnberg et al. [2005])

2003] or general for all core assets [*e.g.*, Bachmann et al., 2003; Pohl et al., 2005; Sinnema et al., 2004b].

Further, several approaches for variability management provide detailed relationship or mapping models between the variability of core assets at different abstraction levels. Mostly, they came in form of a general mapping model [Becker, 2003a,b], metadata model [de Oliveira Junior et al., 2005], basic traceability model [Mohan, 2003], a representation independent variability metamodel [Schmid and John, 2003, 2004], a conceptual model for traceability [Berg et al., 2005], or a framework for variability modeling and management in all abstraction levels [Ch. 9 Capilla et al., 2013; Sinnema et al., 2004b]. A common aspect of these approaches is a formal specification of the relations or associations between the variation points with variants, as variability abstractions in different core assets, the specified variability (*e.g.*, features in an FM), and core assets themselves. For example, in Figure 2.5 we show a simple mapping model between the variant features, variation points with variants in core assets and their relationship with the software entities, without specifying the type of core assets (*e.g.*, requirements, architecture, code, etc.).

Besides these variability management approaches, there are other approaches that address specifically the variability traceability. Mostly, they address the general issues of variability traceability management in SPL engineering. For example, AMPLE [Anquetil et al., 2010] 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 an SPL engineering, specifically when model-driven engineering and aspect-orientation are applied. Quite similarly, XTraQue [Jirapanthong and Zisman, 2009] and the approach by Bayer and Widen [2001] target tracing the variability in different types of artifacts. But, the main idea in XTraQue is to translate the product line documents, the UML documents, into XML documents and to trace their variability by using several types of trace links. So, their main focus is in 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. They consider that the variability in core assets is already made explicit in someway and need just to be traced.

Variability dimension and development dimension. The artifact space (*i.e.*, the development dimension) and the variability space (*i.e.*, the variability dimension), in Figure 2.2, can be amalgamated or not. For example, the preprocessors directives in C for variability realization stay mixed with the functional code. Also, the usage of UML notes or stereotypes for annotating the variable places in requirements or architecture artifacts. Thus, there are approaches for separating the variability dimension and the development dimension [Berg et al., 2005; John et al., 2007; Muthig and Atkinson, 2002]. According to these approaches, the variability dimension is orthogonal with the artifacts space, that is, the phases of development (*i.e.*, requirements, architecture, detailed design), and with their problem space and solution space dimension. Separating these two dimensions is important for supporting the variability management in an SPL.

2.3.3 Orthogonal Approaches to Variability Traceability

When traditional techniques are used for implementing the variability in an SPL, such as inheritance, design patterns, generic types, the code is not shaped in terms of features. In these techniques, the main concerns being separated are objects and/or functions (depending from the programming paradigm, *e.g.*, object oriented or functional). Then, features or *vp*-s with variants are not by-product of any of these techniques, in which case a feature can be implemented as a file, class, method, plugin, package, a combination of them or by any part of them. Therefore, the trace relation is *n–to–m* between the specified features to the *vp*-s with variants at the implementation level. Traditional techniques encompass those methods that are used for single system development but provide the necessary mechanisms to be good candidates for SPL engineering [Coplien, 1999; Muthig and Patzke, 2003; Svahnberg et al., 2005]. There are approaches which shape the core-code assets in terms of features [Apel et al., 2013, 2008; Mezini and Ostermann, 2004; Schaefer et al., 2010], or the specified features hold a direct representation in code [Heymans et al., 2012; Kästner et al., 2008b], that is, their mapping is 1–to–1. We consider that these are alternative approaches to the variability traceability itself, as the traceability is straightforward.

2.3.4 Reverse Engineering Approaches

In the reverse engineering approaches, some aim to find the feature locations in a single software product or different software (product) variants, within a similar domain, and to migrate them to an SPL [Xue et al., 2012; Ziadi et al., 2012]. Basically, the feature location techniques aim at locating all pieces of code that implement a specific functionality, well known as a feature, that is used to differentiate the software applications within an SPL domain. Mainly, the existing feature location techniques are based on information retrieval (IR), static or dynamic program analysis, search based, or a combination of them [Assunção and Vergilio, 2014; Rubin and Chechik, 2013]. Whereas, by locating a *vp* with its variants we

aim at finding the places where some variability is concentrated, that is, their variability implementation technique or mechanism for realizing and resolving the variants of a *vp*. On the other side, there are approaches that show how to reconstruct an FM from the described variability in a propositional formula [Czarnecki and Wasowski, 2007]. In both cases, the capturing of variability is abstracted from the implementation technique, for example, finding the feature locations through analysing the abstract syntaxt tree (AST) of code in several product variants [Thüm et al., 2014; Ziadi et al., 2014]. There are also approaches that describe how to capture the implemented variability and reconstruct the FM when a single technique is used to implement the variability (*e.g.*, using preprocessors in C as implementation technique [Le et al., 2013]). But, an overview on the available techniques for detecting the concepts of features or *vp*-s with variants in source code, by Lozano [2011], shows that there are several approaches for detecting features in code, whereas there is a complete lack of approaches for detecting *vp*-s with variants. *"The lack of approaches may be due to the wide variety of possibilities to translate a conceptual variation point (*i.e.*, a delayed decision) to the implementation of a variation point, as well as to the difficulty to trace this translation"* [Lozano, 2011]. Therefore, the role and characteristics of variability implementation techniques (analysed in Section 3.1.1) in the reverse engineering processes are generally not considered.

2.4 Automated Analysis of Feature Models

In realistic SPLs, variability tends to be large with hundred or thousand of features [*e.g.*, Deelstra et al., 2004; Tartler et al., 2012]. Therefore, an automated analyses is needed for checking the described software products by an FM and whether the cross-tree constraints of features in an FM are well-established and do not invalidate the desired set of software products (*i.e.*, all valid feature configurations).

The current approaches for the automated analysis of feature models are mainly based on propositional logic or constraint programming [Batory, 2005; Benavides et al., 2010; Mannion, 2002]. In this dissertation we use the solving techniques that rely on propositional logic, in which case an FM is translated into a propositional formula and solved by an off-the-shelf logic solver (*e.g.*, SAT or BDD solvers).

Table 2.1 shows the well-known translation rules for translating each logical relation of features in an FM into a propositional formula. Cross-tree constraints (*implies* and *excludes*) are considered to be already in propositional logic. For example, in the third column of Table 2.1 are shown the translation examples for the Graph PL case study. First, each feature is represented by a boolean variable (usually with the same feature's name) and then each logical relation of variant features in a compound feature is translated into propositional logic. Finally, the propositional formula for the whole FM is gained as a conjunction (*i.e.*, connected with the logical conjunction operator $\wedge$) of all these subformulas and cross-tree constraints.

Then, the propositional formula for a feature model is denoted as ϕ_{FM}, for

Table 2.1: Translation rules from an FM to propositional logic

Where P is a compound feature and $C_1, C_2, ..., C_n$ its subfeatures (variant features).

Logical Relation	Propositional Logic	Graph PL Example
Mandatory	$P \leftrightarrow C_i$	GraphProductLine $\leftrightarrow$ GraphType
Optional	$C_i \rightarrow P$	GraphProductLine $\rightarrow$ Search
Or-group	$P \leftrightarrow \bigvee_{1 \leq i \leq n} C_i$	Algorithms $\leftrightarrow$ (Shortest $\vee$ Cycle $\vee$ Strongly-Connected $\vee$ Connected $\vee$ Number $\vee$ MST)
Alternative-group	$\left(P \leftrightarrow \bigvee_{1 \leq i \leq n} C_i\right) \wedge \bigwedge_{i<j} (\neg C_i \vee \neg C_j)$	(Search $\leftrightarrow$ (DFS $\vee$ BFS)) $\wedge$ ($\neg$DFS $\vee$ $\neg$BFS)
Implies	$C_i \rightarrow C_j$	MST $\rightarrow$ (Undirected $\wedge$ Weighted)
Excludes	$\neg(C_i \wedge C_j)$	– (*i.e.*, C_i excludes C_j)

example, for Graph PL (*cf.* Figure 2.1) it is denoted as ϕ_{FM_g} [3] (this whole formula is given in Equation (*Ex. 6.1*), Section 6.1), whereas its valid feature configurations (*i.e.*, valid software products) are given as a set $[\![\phi_{FM_g}]\!]$ (see Table 6.1, Section 6.1). Further, when a propositional formula ϕ_{FM} is evaluated to *true*, by assigning *true* or *false* values to its boolean variables that represent features of software products, we get an interpretation of this formula [4] that represents a valid configuration of features (denoted as $c \in [\![\phi_{FM}]\!]$) or a software variant of an SPL.

In the literature, different analysis operators for automated consistency checking of an FM have been devised [Benavides et al., 2010]. The most important ones concern the detection of the three following anomalies (*i.e.*, inconsistencies):

- *Validity*. A feature model is valid if it represents at least a valid configuration, that is, at least a single software product.
- *Dead features*. A feature is dead if it is not part of any software product, that is, of any configuration. It is denoted as

  $deads(FM) = \{f \in FM | \forall c \in [\![\phi_{FM}]\!], f \notin c\}$
- *False optional features*. When a variable feature is part of every configuration, thus becoming a mandatory feature (*a.k.a.*, common features). It is denoted as

  $common(FM) = \{f \in FM | \forall c \in [\![\phi_{FM}]\!], f \in c\}$,

  where f is a variable feature.

These anomalies are common within a single FM, or between FMs at the same abstraction level. Whereas, there are few approaches for checking the variability

[3] FM_g we used as an abbreviation for $FM_{GraphProductLine}$

[4] Known also as a model of the propositional formula.

inconsistencies between two variability models that are supposed to represent the same variability but in two different abstraction levels, for example, at the specification and implementation levels [Metzger et al., 2007].

2.5 Domain Specific Languages

A Domain Specific Language (DSL) is a small language that is narrowly focused on a particular problem domain. It is defined as "*a focused, processable language for describing a specific concern when building a system in a specific domain. The abstractions and notations used are natural/suitable for the stakeholders who specify that particular concern*" [Volter, 2011], and it is similarly defined by others [Van Deursen et al., 2000].

There are distinguished internal DSLs and external DSLs. In contrast to an external DSL that uses an IDE that is aware of the language itself, an internal DSL is embedded in a host language and is implemented by using its language constructs. For example, one can use general purpose languages, such as Java or Scala, as host languages [Ch. 2,6., Ghosh, 2010; Odersky et al., 2010]. The Scala language supports building also external DSLs using the parser combinators [Ghosh, 2010, Ch. 8]. Moreover, any of these DSLs can be textual, graphical, tabular, or any combination thereof [Voelter et al., 2013; Volter, 2011].

Except in single software development, domain specific languages have an important role also in SPL engineering. Specifically, there are several approaches that suggest the usage of a DSL, built for a specific PL domain, as an intermediate layer between the feature model and the core-code assets. In such case, the FM is related with the implementation of software product variants indirectly as they are expressed in a DSL. The role of DSL in this case is to raise the abstraction level, from the pure implementation of core-code assets, and to ease or automate the product derivation; such DSL usage examples are Voelter et al. [2013]; Weiss et al. [1999]. In such case, the built DSL is tailored specifically for the considered product line domain, meaning that it is used to define the software products.

But, there are other usages of DSLs in the context of SPL engineering. Specifically, when the modeling of software products at the domain level in terms of features with their logical relations, attributes and cross-tree constraints is the domain itself. In this case, a DSL is tailored for modeling the software products within a domain in terms of features and/or different operators for composing and slicing FMs or detecting their anomalies automatically. There are several graphical and textual feature modeling approaches, such as the well-known TVL [Classen et al., 2011], FAMILIAR [Acher et al., 2013] (as textual DSLs), SPLOT [Mendonca et al., 2009] (web-based graphical tool), an extension of FMs [Voelter and Visser, 2011], the CVL [CVL, 2012] (a proposed standard in the OMG), or FeatureIDE [Thüm et al., 2014]. The CVL is itself a DSL not only for modeling the variability in a variability abstraction model (VAM), but also for defining the *vp*-s in the variability realization part (VRM), and resolving the variability (RM) of base models as

MOF-compliant models. A recent prototype implementation of CVL, the KCVL is a textual DSL [Barais et al., 2013; KCVL, 2015], which supports these three parts of CVL. Further, as an integrated development environment, FeatureIDE aims in supporting the development of core assets in the whole engineering cycle of an SPL by integrating different tools, whereas the creation of an FM is supported by a graphical interface.

Such feature modeling tools or DSLs, textual or graphical, differ regarding their support and automated analysis that they offer for feature models. In this work, we use the DSL in this last meaning, as we aim in providing a tool support for modeling the variability of core-code assets. But, we cannot used the existing tools or DSLs for modeling the variability of core-code assets for four main reasons. (1) *vp*-s with variants as abstractions at the implementation level need to be related with the variable elements of core-code assets (*e.g.*, the CVL maintains such relationship with the models, but not with the core-code assets). Then, (2) except their logical relation, we need to model other properties of *vp*-s with variants, such as their binding time, and evolution properties. Further, (3) at the same time these *vp*-s with variants should be traced with the features at the specification level. Overall, (4) we aim to model the variability of core-code assets in a fragmented way instead of modeling it in a single place or variability model. The existing DSLs, such as FAMILIAR or TVL, support a fragmented view and operators for composing, slicing, or analysing multiple FMs within a domain.

Part I:

Design of Variability Models at the Implementation Level

CHAPTER 3

Diversity in Variability Implementations

This chapter shares material with the following paper:

Tërnava, Xh. and Collet, P. (2017b). On the diversity of capturing variability at the implementation level. In *Proceedings of the 21st International Systems and Software Product Line Conference - Volume B*, SPLC '17, pages 81–88. ACM

In this chapter, we analyse the diversity in variability implementations. Specifically, we study the diverse characteristic properties and the quality criteria of variable parts in core-code assets (introduced in Section 2.2.1), then the classification of variability implementation techniques at the code level. With respect to these dimensions, we analysed 21 variability implementation techniques and created a catalog of them. Toward using this catalog, an illustrative example is given with some additional facets for evaluating and choosing a technique.

3.1 Dimensions of Diversity

Many techniques for implementing the variability of core-code assets have been identified and proposed from different research works or shared industrial experiences on SPL engineering [*e.g.*, Apel et al., 2013; Bachmann and Clements, 2005; Coplien, 1999; Gacek and Anastasopoules, 2001; Muthig and Patzke, 2003; Patzke and Muthig, 2002; Svahnberg et al., 2005]. During our work, we analysed each of these studies and the shared industrial experiences that we have encountered in the literature. Instead of following a systematic literature review, we have analysed one-by-one the existing approaches that we found concerning the variability implementation in core-code assets. Mostly, we found the relevant works by looking into the previously referenced works within each approach on variability implementation techniques, and then we checked the other studies that reference them too.

During their study, we observed that the diversity in variability implementations can be organized according to three orthogonal dimensions: (1) the characteristic properties of variable parts that each technique supports (*e.g.*, the binding time), (2) the quality criteria of variable parts or techniques (*e.g.*, shaping the code in terms of features), and (3) the classification of variability implementation techniques (*e.g.*, traditional, emerging). The details on each of these dimensions are given in the following.

3.1.1 Characteristic Properties of Variable Parts

A lot of knowledge is gathered about features of software products in an SPL during domain analysis. Usually, just a part of that knowledge is modeled in the feature model (FM) and the rest remains implicit or is kept in informal ways (*e.g.*, the binding time of features). Specifically, from Section 2.1 and Figure 2.1, an FM represents only (1) the parent-child hierarchy of features (*specialization/generalization*, and/or *consist-of*), (2) the logical relations between features (*mandatory*, *optional*, *or*, *alternative*), and (3) possibly some cross-tree constraints (*requires*, *mutual exclusion*) that are expressed in propositional logic [Ch. 3 Capilla et al., 2013; Kang et al., 1990].

For example, from Figure 2.1 (Section 2.1), (1) feature `Directed` specialize/generalize feature `EdgeType`, (2) `Directed` with `Undirected` are alternative features, and (3) feature `MST` requires ($\Rightarrow$) `Undirected` and ($\wedge$) `Weighted` features.

However, variation points (*vp*-s) with variants in core-code assets are characterized by a richer set of characteristic properties, which are additional to the documented knowledge of features in the FM. These *vp*-s with variants and their properties are important to be captured during both foward and reverse engineering processes. For example, the logical relation between variants and their binding time are important to reconstruct the FM, or to be documented for resolving variability.

Remark. *Capturing a variation point (vp) or variant means to abstract a variable asset element in a core-code asset or a whole variable core asset* (i.e., *the place where the variability happens), and thus representing it by a variation point or variant concept.*

For example, in Listing 1.1 (Section 1.1.2) two *vp*-s and four variants can be captured. The `callback` function in lines 26-27 is the *vp* `vp_edgetype` and the parameter `WEIGHTED` in line 2 is captured as another *vp* `vp_weight`. Then, the methods in lines 6-13 and 14-25 are captured as variants `v_directed` and `v_undirected`, respectively, for the first *vp*. Similarly, the `true` and `false` values in line 2 are captured as variants `v_weighted` and `v_unweighted`, respectively, for the second *vp*.

In the following, we gather the important characteristic properties of *vp*-s with variants that should be considered when choosing a variability implementation technique or during the capturing of variability in core-code assets. These properties are diverse as they depend from the variability implementation techniques that are used to implement a domain.

Logical relation. As *vp*-s with variants at the implementation level are refinements of features in an FM at the domain level (see their definition in Section 2.2.1), the logical relations of *vp*-s and of their variants are similar to the possible relations

Table 3.1: Logical relations of *vp*-s and of variants in a *vp*

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

between features in an FM (*cf.* Figure 2.1), and they are shown in Table 3.1. A single variability implementation technique can offer at least one of these logical relations, for example, the inheritance can be used for implementing the alternative variants, overriding for implementing the multi-coexisting variants, or aggregation for optional variants.

Concretly, in Listing 1.1 (Section 1.1.2) we capture the alternative logical relation between variants `v_directed` and `v_undirected`, which is realized by the strategy pattern.

Variation points with variants can have dependencies in core-code assets [Bühne et al., 2003]. They are similar to the dependencies between features in an FM (*i.e.*, mutual exclusion, requires, which are shown in Table 3.1). In addition, when a feature or a set of features modify or influence another feature in defining the overall system behaviour [Zave, 1999], it is reflected in one of the *vp*-s with variants dependencies.

Binding time. Within a domain, *vp*-s may require being resolved at different development phases. Thus, 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. This should be supported by the chosen variability implementation technique. A *vp* can be resolved early during the development cycle (*e.g.*, when the decision for a variant is made at compile time), or later during the development cycle (*e.g.*, dynamically, at runtime) [Alves et al., 2009; Bosch, 2000]. Binding units are important for identifying which *vp*-s should be bound together (*e.g.*, when several *vp*-s participate in implementing some major functionality in a system and they should be resolved together as a unit) [Ch. 4 Capilla et al., 2013; Lee and Kang, 2004; Lee and Muthig, 2008]. The binding time here is the time when the variability should

Table 3.2: Binding times of *vp*-s with variants (adapted from Bosch and Capilla [2012]; Capilla et al. [2013])

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, i.e., the decision for a variant in a variation point is made early/statically.*
Dynamic binding (D)	(D) runtime (start-up) (D) pure runtime (operational mode)	*The variability is resolved later during the development cycle, i.e., the decision for a variant in a variation point is made as late as possible/dynamically.*

be resolved and should not be confused with the time when it will be introduced, which is differentiated by Svahnberg et al. [2005]. As for the common kinds of binding times, a taxonomy is given by Capilla et al. [2013] and Bosch and Capilla [2012]. They are also shown in Table 3.2.

For example, in Listing 1.1 (Section 1.1.2) we capture that the `vp_edgetype` is bound during the runtime to one of its variants, for instance to `v_directed`.

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 dynamic binding [Rosenmüller et al., 2011]. 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 areusually combined. However, 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.

Defaults. Some variability is not subject to frequent variations among the majority of software products in an SPL. In such cases, one of the variants on a *vp* can be set as its *default variant* [Coplien, 1999, p. 94].

For example, in Listing 1.1 the `v_unweighted` is captured as a default variant of the `vp_weight`. It is realized by setting the argument `w: Int = 1` (line 27), as a default value.

Granularity. A *vp* or variant in core-code assets can have different granularities depending on the size of variability and the used technique [p. 59 Apel et al., 2013; Kästner et al., 2008a] (*cf.* Table 3.3). As abstractions, a *vp* or variant can represents a coarse-grained element that is going to vary (*e.g.*, a file, a package, a class, an

Table 3.3: Granularity of *vp*-s with variants

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

interface), a medium-grained element (*e.g.*, a method, a field inside a class), or a fine-grained element (*e.g.*, an expression, a statement, a block of code).

For example, in Listing 1.1 we should be able to capture the `v_directed` in method level (realized by method `adddirectededge()` in lines 6-13); or, the `vp_weight` as a parameter (realized by `WEIGHT` parameter in line 2).

Evolution. Depending on whether the specified variability in the FM is meant to be evolved with new features, *vp*-s can be *open* or *closed* (*i.e.*, to be extended with new variants in the future or not, respectively).

For example, we capture a *vp* as *closed* when it is implemented as an `enum` type in Java, and *open* when it is implemented simply as an abstract class.

Abstract features. In addition to these characteristic properties for *vp*-s with variants, some of the features in the FM are abstract [Thum et al., 2011].

For example, in Figure 2.1 (Section 2.1) the conceptual feature `GraphProductLine` is an abstract feature, which is colored with the lighter blue.

The abstract features are introduced to structure the variable features in the FM but do not require any implementation. This property becomes important during the traceability of variability, as there are no abstract *vp*-s or abstract variants in core-code assets where the abstract features can be mapped.

3.1.2 Quality Criteria

Variability implementation techniques are supported by different constructs among potentially various programming languages, thus providing different qualities for the resulting implemented variability. A dominant quality criterion is

the ability to shape the code (*i.e.*, its variability) in terms of features as cohesive reusable units so, to handle more easily the variability among distinct abstraction levels. Several other quality criteria are introduced in the literature [Apel et al., 2013; Fritsch et al., 2002; Gacek and Anastasopoules, 2001; Patzke et al., 2011]. Some that are widely studied are shown in Table 3.4. We also consider them as the most important ones for evaluating techniques and capturing the *vp*-s with variants.

Table 3.4: Quality criteria of variability implementation techniques

Quality	**Description**
Preplanning effort	*The required preplanning effort to introduce and use a variability implementation technique.*
Visibility of *vp*	*Variation points in code can be explicit, implicit* (e.g., *in cloning technique [Patzke et al., 2011]), or ambiguous* (i.e., *in traditional techniques when the same mechanism is used for implementing the variability and the functionality of software).*
Information hiding	*It enables the modular reasoning of variability in implementation by separating a system into modules [Apel et al., 2013, Ch. 3]. In such case, each module has an internal part, which is encapsulated or hidden from other modules, and an external part, which represents the interface or the contract with the rest of the system.*
Uniformity	*Some techniques that are used for variability at the implementation level can also be applied to realize the variability to the other not code artifacts among the other abstraction levels.*
Separation of Concerns (SoC)	*While the main concerns in an SPL development are features, SoC is related here to the ability to shape the code in terms of (modular) features*
Traceability	*The ability to trace features in their development lifecycle, specifically with the implemented artifacts, which is also described in Section 2.3.1.*
Scalability	*The ability of a technique to support new user requirements or some extension/evolution of variability* (i.e., *being able to add new variants to a vp over time without changing their implementation technique).*

For example, from Listing 1.1, using the *strategy pattern* requires more preplanning effort than using the technique of *parameters*.

A single technique supports differently each of these quality criteria. Overall, it is not possible to meet all of the quality criteria as they have conflicting properties, for example, by choosing a technique that offers little preplanning effort, such as

the technique of parameters, it makes difficult the traceability of variability at the finest-grained level (*i.e.*, at the parameter level). Similarly, by chosing any design pattern to implement some variability, the information hiding and traceability is improved but it requires more preplanning effort.

3.1.3 Classifications of Techniques

Due to the evolving diversity of variability implementation techniques, researchers have grouped them differently and mentioned dissimilar subsets of them [*e.g.*, Apel et al., 2013; Gacek and Anastasopoules, 2001; Kästner, 2010; Kästner and Apel, 2008; Muthig and Patzke, 2003; Patzke and Muthig, 2002; Svahnberg et al., 2005]. In general, implicitly or explicitly, all variability implementation techniques are found to be classified based on three orthogonal subdimensions: (1) traditional or emerging, (2) language-based or tool-based, and (3) annotative or compositional.

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

Traditional techniques. They have emerged and evolved indipendently and before the emergence of the SPL paradigm. Alternatively known as classical techniques, they encompass methods that are used for single system development but, nevertheless, provide the necessary mechanisms to be good candidates for SPL engineering. Example of these techniques are inheritance, overloading, generic types, design patterns. Consequently, in all these techniques, the concept of feature does not have a first-class representation in implementation.

Emerging techniques. On the contrary, these techniques have emerged as the SPL engineering field advances. Here, the concept of feature is a first-class citizen at the code level. In our study, all these techniques come from academia, such as frames [Bassett, 1996], feature modules [Apel et al., 2013], delta modules [Schaefer et al., 2010]. They are mainly integrated by language extensions through specific mechanisms and do not require heavy tooling. For example, the *Jak* language is an extension of Java language where each feature is realized by using the language construct of *layer*, which represents a feature module [Batory et al., 2004]. In the literature, these techniques are also called *advanced techniques* [Apel et al., 2013].

Some of the techniques only depend on language mechanisms, whereas others rely on extra tool support. Based on Apel et al. [2013], we introduce the following subdimension of classification.

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

Tool-based techniques. In this case, specialized tools are used to identify and resolve variability among the software assets. Although they are conceptually independent of any given language and orthogonal to its constructs, currently they are only supported by specific programming environments. For example, such technique is frames [Bassett, 1996], where a specific frame processor is needed to automatically execute the frame commands that are used to denote the variability in code level. Currently, such frame processors are developed by Bassett [1996], the XVCL language by Jarzabek et al. [2003]; Swe et al. [2002], and the plain frame processor (fp) by Patzke and Muthig [2003].

The third subdimension is about, the now common distinction, on the way that variability is represented and resolved at code level [Kästner, 2010; Kästner and Apel, 2008].

Annotative techniques. The specified features are realized in core-code assets as a whole and the variants are annotated by a technique in order to include or exclude them during variability resolution. Different variability implementation techniques have different means for realizing annotations, for example, preprocessor directives in C [Tartler et al., 2012], or simple tagging approach [Heymans et al., 2012], but this last one falls into the tool-based approaches and not techniques. When an annotative technique is used to remove the unneeded variants from core-code assets, it supports *negative variability*.

Compositional techniques. Variable features aim at having a cohesive or modular representation at code level, for example, in form of components, plug-ins, classes, packages, modules, aspects, deltas, or subjects. The intention is that a final product is derived by simply attaching or composing any of these modular units or variants in the base assets as part of core-code assets that are included in all software products. Thanks to this ability to compose the needed variants, these techniques are known to support *positive variability* (*e.g.*, the technique behind feature modules). In the last subdimension, negative or positive variability may sometimes be orthogonal to this classification. For example, the delta programming as compositional technique suggests to add or remove the delta modules from the base core-code assets.

Another very notable set of techniques is the one that follows a *generative approach* [Czarnecki, 2005; Czarnecki and Eisenecker, 2000]. These techniques have the ability to generate software products from the generic assets (*i.e.*, core-code assets). Respectively, they may use a description in a higher abstraction language level, such as a Domain Specific Language (DSL), for deriving software products [Weiss et al., 1999]. We see these techniques as orthogonal to our issue of evaluating and choosing a variability implementation technique or capturing the variability in core-code assets.

3.2 Catalog Building Method

In this section we describe both the building method and the resulting catalog of variability implementation techniques regarding the characteristic properties of *vp*-s with variants that should be realized or can be captured in core-code assets.

3.2.1 Covered Techniques

To have an up-to-date catalog of implementation techniques, we collected the identified variability implementation techniques so far. They are diverse and spread among different research works and shared experiences, forming a significant set. It must be noted that we did not follow a systematic process in our literature review. We took the existing classifications and related references as starting points for finding the relevant research publications. Then, we decided to consider only those techniques that are used in a closed-world SPL engineering process [Kästner et al., 2011] (*i.e.*, when a technique is used for implementing a set of features with a closed view within a domain). Different programming languages support subsets of them (see a given mapping of techniques with the programming languages by Gacek and Anastasopoules [2001]), and even for a single technique, different mechanisms or language constructs can be used. We excluded techniques like components and frameworks as they already use the considered closed-world techniques to realize their variability.

3.2.2 Evaluation Process

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

First process. We used four small case studies from different domains to experiment with several techniques and *vp*-s characteristic properties. Those case studies are: *Arcade Game Maker PL* [AGM, 2009], *Microwave Oven PL* [Gomaa, 2005], *Expressions PL* [Lopez-Herrejon et al., 2005], and *Graph PL* [Apel et al., 2013; Lopez-Herrejon and Batory, 2001] [1]. Their domains are quite well understood and used by the SPL community. A summary of their development is given in Appendix B. For their implementation, we used the Scala language, which supports both the object-oriented and functional paradigms and has rich constructs for modularization of features. This enables us to cover more choices of techniques with only one implementation language.

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

Second process. We built an informed opinion for each technique and some of their classification criteria from the existing research works, some of them are summarized in Section 3.2.4. These works mainly evaluate a technique using Java and C++ language constructs or extensions of these languages (*e.g.*, AspectJ). We thus examined manually the realized evaluations and the obtained results by the others. Similar to us, most of them use three evaluation levels for a criteria: good support, possible support, and no support. After the examination step, we compared the results from these works, especially when they were considering the same criteria for a same technique. Whenever two different works do not agree on a value for a specific criterion, we did a more depth analysis in the literature (*e.g.*, analysing more detailed approaches, auch as Ramos Alves [2007]) or, when was possible, we used one of our case studies to evaluate it. As a result, each technique was systematically evaluated by the mentioned criteria in the three classification dimensions (given in Section 3.1).

3.2.3 Resulting Catalog

The evaluated techniques are shown organized in the proposed catalog in Table 3.5 [2]. Techniques are gathered in two main groups, those that provide *ad-hoc reuse* and *methodological reuse*. We did not want to exclude from our classification the most applied techniques such as *copy-and-paste* or *cloning*. These techniques have well-known drawbacks and may not scale due to their *ad hoc* form of reuse. However, this is strongly related to the maturity levels of an SPL [Antkiewicz et al., 2014; Bosch, 2000] and practical for building quick solutions [Patzke, 2010], which can be refactored later. The other techniques encourage the methodological reuse as the main ingredient for a sustainable SPL.

The catalog contains two legends. The first legend, Legend A, is used to explain the evaluation results and the second one, Legend B, is used to explain our evaluation method using previous works. Therefore, some evaluation values to a criteria are associated with a reference work, using numbers, to show when a result is influenced by an existing work and by which one. For example, the technique of Frames for the criterion of Binding Time (see the colored intersection in Table 3.5) is evaluated as offering static binding of variants. This result is supported by three references, [Gacek and Anastasopoules, 2001; Patzke and Muthig, 2003; Patzke et al., 2011], refereed by numbers 2, 5, and 7, respectively. In Table 3.5 only some of the main works that we used are shown, as they cover a considerable set of techniques. These references can be used also to show how the evaluated techniques and the used criteria are distributed among the previous research works.

In this way, we address the Challenge A1 to comprehend the diversity in variability implementations of core-code assets, and to compare the majority of the existing variability implementation techniques by the same set of properties.

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

Table 3.5: Catalog of variability implementation techniques

	Feature types			Binding time		Granularity			Preplanning effort	Visibility of *vp-s*	Information hiding	Uniformity	Sep. of concerns	Traceability	Scalability	Language paradigm	Annotative	Compositional	Language-based	Tool-based	Traditional	Emerging
Legend A: ●: good support / belong ◐: possible support (difficult) ○: no support (not often applicable) / does not belong ✱: high; ◖ : average; ◔ : low **E**: explicit; **A**: ambiguous	*Optional*	*Or*	*Alternative*	*Static* (S)	*Dynamic* (D)	**Defaults**	**Open for evolution**	*Coarse*	*Medium*	*Fine*												
AD-HOC REUSE																						
Cloning / Patching	●	●	●	● 7	○ 7	○ 7	● 7	● 7	●	●	◔	A/E 7	○ 7	○ 7	○ 7	Not specific 7	●	●	●	○	●	○
Conditional Execution (Parameters)	● 7	○ 7	● 7	◐ 1,7	● 1,7	○ 7	○ 7	○ 1,7	◐ 1,7	● 1,7	◔ 1	A 7	◐ 1	◐ 1,7	○ 1,7	Not specific 7	● 1	○ 1	● 1	○ 1	● 1	○ 1
METHODOLOGICAL REUSE																						
Preprocessor directives	● 2,3,4,7	◐ 2,4	● 1,2,3,4,7	● 1,2,7	○ 1,2,7	◐ 6,7	○ 7	● 1,3,4,6,7	● 1,3,4,6,7	● 1,3,4,6,7	◔ 1,7	A/E 7	○ 1	○ 2,7	◐ 1,2	◐ 2,7	Not specific 7	● 1,2,4,6	○ 1,4,6	○ 1,3,4,6	● 1,3,4,6	● 1,3	○ 1,3
Argument defaulting	● 4	○ 4	○ 4	●	○	● 6	○	○	●	○	◖ 1	A 7	◐ 1	○ 2,7	○ 1,2	◐ 2,7	Not specific 3,4,6	○ 3,4,6	●	○	◐ 3	○ 3
Overriding	◐ 4	●	○	○	●	●	●	○	●	○	◔	A 7	○ 1	● 7	● 1,7	● 1,7	O. Oriented 7	○	●	○	●	○
Aggregation / Delegation	● 2	◐ 2	◐ 2	● 2	● 2	●	◐	●	●	○	◖	E 6	◐ 1	○ 2	◐ 2	◐ 2	O. Oriented 2	● 2	● 2	○	●	○
Inheritance	● 2	● 2	◐ 2	● 2,6	● 2,6	●	●	● 6	● 6	○ 6	◖	A 6	○ 1	● 2	● 2	● 2	O. Oriented 6	◐ 2,6	● 6	○ 6	●	○
Reflections	● 2	● 2	● 2	◐ 2	● 2	●	◐	●	●	○	✱	E 6	○	● 2	○ 2	● 2	O. Oriented 6	○	●	○	●	○
Aspects	◐ 2,4	● 2,4	◐ 2,4	● 1,2,7	◐ 1,2,7	● 7	● 7	○ 1,7	● 1,7	○ 1,7	◖ 1,7	A 1,7	○ 1	◐ 1	● 1,3,7	● 1,7	Aspect Ori. 1,3	◐ 2	● 1,2,3	● 1,3,7	○ 1,3,7	● 1,3
Polymorphism																						
Coercion (Casting)	◐ 4	◐ 4	◐ 4	●	○	● 4	●	●	○	○	✱	A	●	○	●	●	O. Oriented 3,4	○	●	○	● 3	○ 3
Overloading	○ 2,4	○ 2,4	● 2,4	● 2,6	○ 2,6	●	●	○ 6	● 6	○ 6	◖	A	●	○	◐ 2	◐ 2	Procedural 3,4	● 2,6	● 2,6	○	● 3	○ 3
Subtype polymorphism	◐ 4,7	◐ 3,4	● 3,4,7	◐	● 3,4,7	● 6	● 7	○ 7	● 7	○ 7	✱	A 7	○	● 7	●	●	O. Oriented 3,4	○	● 7	● 7	○ 7	○ 3
Parametric polymorphism (generics)	○ 3,4,7	● 3,4	● 3,4,7	● 3,4,6,7	○ 3,4,6,7	○/● 7		● 6	● 6,7	○ 6,7	✱	E 6,7	○	◐	●	◐	Generic prog. 3,4	● 6	● 7	● 4,7	○ 4,7	● 3
Design patterns																						
Strategy pattern	● 1	● 1	● 1	◐ 6	● 1,6	● 1	●	●	● 1	○	✱ 1	A 1	○ 1	● 1	● 1	●	O. Oriented 1	○ 1	● 1	○ 1	● 1	○ 1
Decorator pattern	● 1	● 1	○	◐	● 1	●	●	●	●	○	✱ 1	A 1	○ 1	● 1	● 1	●	O. Oriented 1	○	●	○	●	○
Observer pattern	●	●	●	◐	● 1	●	●	●	●	○	✱ 1	A 1	○ 1	● 1	● 1	●	O. Oriented 1	● 1	● 1	○ 1	● 1	○ 1
Template method pattern	● 1	◐ 1	● 1	◐ 6	● 1,6	● 1	●	●	● 6	○	✱ 1	A 1	○ 1	● 1	● 1	●	O. Oriented 1	○ 1	● 1	○ 1	● 1	○ 1
Visitor pattern	◐	●	◐	◐	●	●	◐	●	●	○	✱	A	○	●	●	●	O.Oriented	○	●	○	●	○
Emerging techniques																						
Frames	● 2,5	● 2,5	● 2,5	● 2,5,7	○ 2,5,7	● 5,7	○/● 7	● 5,7	● 5,7	● 5,7	◖ 2,7	E 5,7	◐ 1	● 2,7	◐ 2	◐ 2,7	Not specific 5,7	● 2,5	○ 2,5	● 5,7	○ 5,7	●
Feature Modules	◐ 4	● 1,3,4	◐ 4	● 1	◐ 1	●	●	● 1	● 1	○ 1	◖ 1	A 1	○ 1	● 1,3	● 1	● 1	Feature Ori. 1	○ 1	● 1,3	○ 1,3	○ 1,3	● 1,3
Delta Modules	◐	●	◐	● 1	○	●	●	● 1	● 1	○ 1	◖	A	○	◐	●	●	Delta Ori. 1	● 1	● 1	○ 1	○ 1	● 1

Legend B: **1** → Apel et al. [2013]; **2** → Gacek and Anastasopoules [2001]; **3** → Muthig and Patzke [2003];
4 → Patzke and Muthig [2002]; **5** → Patzke and Muthig [2003]; **6** → Coplien [1999]; **7** → Patzke et al. [2011]

3.2.4 Related Work

An important issue for variability implementation is the ability to evaluate and choose a technique that will fulfill best some given variability requirements. Several evaluation schemas of various implementation techniques are currently available. They mainly came from the academia in form of *frameworks*, *taxonomies*, *studies*, and *catalogs*. We used them for building our catalog in Table 3.5; and, the most influential ones are described in the following in no particular order.

Taxonomies. Svahnberg et al. [2005] group and organize the techniques by two dimensions of similarity. According to the size of software entities in which the techniques can be applied (components, frameworks, and lines of code), and their latest binding time. Examples of concrete techniques categorized by this taxonomy have been given, but for the majority of the other techniques it is up to the user to evaluate and categorize them.

Catalogs. Patzke and Muthig [2002] gather several techniques and evaluate them by three types of variability: optional, alternative, and multi-coexisting. They also give a model that captures how and which technique is appropriate to be introduced [Patzke, 2010] depending on the maturity level of the SPL [Antkiewicz et al., 2014; Bosch, 2000]. Concrete examples on each technique and other details have been complemented later in an enriched set of techniques and evaluation criteria [Patzke et al., 2011]. Quite similarly, Muthig and Patzke [2003] evaluate a slightly different set of techniques and criteria. Gacek and Anastasopoules [2001] discuss also another rich set of techniques and criteria. They are organized in a catalog and evaluated by the proposed criteria. A second catalog gives details on the language support for each technique.

Studies. Coplien addresses many issues for realizing the commonalities and variabilities [1999], although with no direct emphasis on SPL engineering. He introduces several techniques from different programming paradigms for variability implementation, and enlightens how these techniques factorize the commonality and accommodate the variability on it [Coplien et al., 1998].

Apel et al. [2013] recently cover the majority of the identified variability implementation techniques and study a set of criteria for each of them. Although a deeper discussion about techniques is given, they are not organized in a catalog nor compared, while only a subset of previously used evaluation criteria is considered.

Some of the variability implementation techniques are evaluated separately, such as Aspects [Anastasopoulos and Muthig, 2004] or Frames [Patzke and Muthig, 2003]. It is important to note that in almost all these evaluation schemes, techniques are evaluated in isolation. A notable exception is the combined evaluation of aspects and other techniques by Apel et al. [2008], which analyses the synergy between aspects and feature modules to implement crosscutting features.

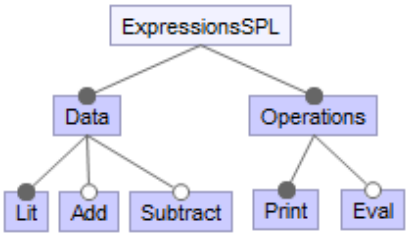


Figure 3.1: The FM of Expressions PL

Furthermore, the combination of annotative and compositional techniques was also investigated by Kästner and Apel [2008].

Frameworks. The work of Fritsch et al. [2002] is of a different nature. They propose an evaluation framework for techniques. Several evaluation criteria are determined, but it is up to the developers to evaluate the techniques by these criteria and to create their own catalog.

As the SPL engineering field is making progress, various techniques are regularly identified. New ones have emerged and have not been taken into account with enough details in the previous comparisons or catalogs. Considering all the work mentioned above, we observe that they cover different subsets of techniques and use different evaluation methodologies and criteria. Merging all techniques and criteria is not straightforward and cannot be made easily by SPL architects or developers. Making a decision over one or several variability implementation techniques is then a cumbersome task. Therefore, we created a richer catalog where all techniques are evaluated by the same set of criteria (see Table 3.5), and, still, we kept a mapping of these criteria and techniques to the existing works (see Legend B in Table 3.5).

3.3 Choosing a Technique

3.3.1 Illustration

For illustrating the evaluation steps in choosing a technique, by using the catalog and the unexpressed facets on it, we use the Expressions PL case study [Lopez-Herrejon et al., 2005; Loverdos and Syropoulos, 2010, Ch. 12]. Its feature model is shown in Figure 3.1, whereas the development details for its three versions are given in Appendix B.

We use a proactive approach for implementing the variability of Expression PL. Then, we decide for choosing those techniques that offer a methodological reuse (*cf.* Table 3.5). From these techniques, we rule out the ones that are not directly supported by any of the Scala language constructs, such as frames, feature

Table 3.6: Two implemented versions of the Expressions PL

	version 1		**version 2**	
Variation Point	Operation	Data	Operation	Data
Variants	*Print, Eval*	*Lit, Add, Sub*	*Print, Eval*	*Lit, Add, Sub*
Technique	Subtype polymorphism		Visitor Pattern	
Lang. mechanism	Bounds, Traits	Traits	Classes	Case classes
Open/Closed	Open	Open	Open	Closed
Binding time	Compile time	Static config.	Runtime	Compile time
Granularity	Traits	Traits, methods	Class	Method
Visibility of VPs	Explicit	Explicit	Implicit	Explicit
	(Abstract type)	(Trait)	(-)	(Trait)
Comment:	*Operation-centric decomposition*		*Data-centric decomposition*	

modules, or deltas. We then take into consideration the logical relations between features in the FM to choose a technique. As a result, several techniques are left to be chosen, and we decide for any of them in an arbitrary way. Concretely, in Table 3.6 are shown the different chosen techniques to implement the case study as two different versions. These two implementations are given in Appendix B.2.

In version 1 (*cf.* Table 3.6), from the remaining options of techniques, we choose the subtype polymorphism (*cf.* Listing B.1). Thus, one *vp* has static binding and the other dynamic binding, the optional logical relation between features is supported, the *vp*-s are left open for adding more variants in the future, and the code is shaped in terms of features by using the trait mechanism in Scala language. Whereas, in version 2 we choose a single technique (*i.e.*, the visitor pattern with case classes in Scala) for implementing the two *vp*-s (*cf.* Listing B.2). We decide for using the visitor pattern because the lines of code are fewer compared with any other implementation as it is supported directly by a Scala language construct (with the case classes). Concretely, the version 1 has 73 lines of code, whereas the version 2 has 34 lines of code. In Table 3.6 are shown the other properties of *vp*-s with variants. By using this technique, the code is not shaped in terms of features, thus making difficult the management or traceability of variability.

3.3.2 Discussion

The methodological development and reuse of core-code assets can be achieved by evaluating and choosing a technique in a systematic way that fits best to implement some variability. The given catalog in Table 3.5 supports such systematic evaluation and choice of techniques, but it is insufficient.

Concretely, although we used the catalog for realizing the two versions [3] in Table 3.6, we still took some decisions that were unexpressed in the catalog. Specifically, we decided for using the Scala language, which looks as an ad hoc decision but was conform with our personal knowledge and preferences. Therefore, we

[3] One more version is given in the repository link, in Appendix B.

noticed that to choose a technique, there are few more facets that make difficult a systematic choice of techniques, which should be considered and are unexpressed in the catalog. We discuss them in the following.

Evaluation is informal. Evaluating whether we are selecting the right technique is, not surprisingly, difficult as it is largely informal. First, a feature model itself does not say much, obviously, about how its features are going to be implemented. Even some of the characteristic properties of variable parts that are important for the end users, such as the binding time, are unspecified in the FM. Similarly, the other characteristic properties and quality criteria are also unspecified and the developers have to balance and make trade-offs on them [Apel and Kästner, 2009]. Secondly, variability implementation techniques came from different development paradigms (*e.g.*, object-oriented, functional) that are supported by different languages, which may use different language constructs even for the same paradigm. Overall, there is a variety of domains and strategies within an organization, which may prefer to support some of the characteristic properties of variable parts or the quality criteria over the others.

Evaluation from different perspectives. An aspect of techniques, which is implicit in the catalog, is that they can be evaluated from different perspectives. Specifically, a technique can be evaluated (1) whether it addresses the characteristic properties of variable parts that are relevant for the end users or for the resolution of variability (*e.g.*, by the logical relation between variants, their binding time), (2) whether it simplifies the development of variability (*e.g.*, by the preplanning effort, information hiding), or (3) whether it improves the management of variability (*e.g.*, by shaping the core-code assets in terms of features – SoC, or easing the traceability of variability). All techniques overlap regarding this aspect and any of such perspectives can have an impact in evaluating and choosing a technique. Although implicit, these perspectives can be used to begin with the evaluation of a technique by using our catalog for implementing some variability.

Non-isolated evaluation. Another aspect is that a single technique may not cover well all requirements within an SPL domain. Therefore, a given domain may employ several techniques and paradigms (*e.g.*, object-oriented, functional) [Coplien, 1999], although a smaller set of them is suggested in order to simplify the management of variability [Bachmann and Clements, 2005]. This is evident when traditional techniques are used, such as inheritance, overloading, design patterns. The emerging techniques, such as feature modules or delta modules, are meant to be used as a single technique for development of the whole SPL. Similarly, preprocessors in C, which have a wide usage in the realistic SPLs, are used as a single technique. Actually, these techniques annotate or modularize the variability of core-code assets in terms of features, which are implemented at first by a programming paradigm (*e.g.*, object-oriented, functional). In other words,

when techniques are used together they have an effect on each other regarding the implementation of variability. A combined usage of paradigms and techniques means that a systematic choice of a technique depends also on the other chosen techniques, which implement together the variability of some strongly related sub-domains.

For example, in Listing 1.1, we could not use the strategy pattern to implement the variants `v_directed` with `v_undirected` and then also the variants `v_weighted` with `v_unweighted`, although both are in an alternative relation.

Dependency on the adaptation model of the SPL. As a last but not the least aspect, the development of core-code assets is basically driven by the adaptation model of the SPL, which can be a proactive, extractive, or reactive adaptation [Krueger, 2002a] (see Section 1.1.1). When it is not a proactive way then, the choice of a technique and usage of the catalog depends from the existing techniques used for implementing the variant software application(s). Consequently, depending on the adaptation model some techniques may be more suitable to be chosen over the others.

3.4 Summary

A diverse and growing set of variability implementation techniques is available in the SPL engineering field. Understanding and choosing an appropriate technique is not a trivial task for architects and developers. Besides, the variation points with variants in core-code assets become diverse, meaning that the realisation and the capture of variability at the implementation level is not uniform. Different subsets of techniques have been identified and classified while new techniques emerged.

In this chapter, we studied the elements of diversity for capturing the implemented variability of core-code assets in terms of variation points with variants, in form of their characteristic properties or comparison criteria for their implementation techniques. Then, we studied the majority of the variability implementation techniques and provided a unified set of comparison criteria for them. We organized these criteria in a catalog that covers an enriched set of techniques, which are compared with a same set of criteria. Finally, we provide an illustrative example to show the diversity of the characteristic properties for variation points with variants even within a small case study. We also illustrate the usage of our catalog and discuss four of the unexpressed facets of techniques in this catalog, which should be considered during the evaluation and choice of a variability implementation technique.

CHAPTER 4

Technical Variability Models

This chapter shares material with the following paper:

Tërnava, Xh. and Collet, P. (2017). Tracing imperfectly modular variability in software product line implementation. In *International Conference on Software Reuse*, pages 112–120. Springer

---

In Section 2.1, we have shown the necessity to differentiate and model separately the variability in problem and solution spaces. Specifically, one of our main challenges (*cf.* Challenge A) is to model the variability of core-code assets, as solution space artifacts. Toward this, we undertake two main steps: (1) capturing the variability of core-code assets and (2) modeling it in variability models. For the first step, we reflect on whether we should use the concepts of features or variation points (*vp*-s) with *variants* to abstract the variability at the implementation level (which analysis is given in Section 2.2.2). Therefore, we first study their differences as concepts and their terminology, and then their differences during the capturing of variability in a forward or reverse engineering process. In particular, we are focused in the common situation in which a combined set of traditional techniques are used together for variability implementation, thus tackling the Challenge A2. Finally, we propose a framework for capturing and modeling the variability of core-code assets in a fragmented way, thus addressing the Challenges A3 and A4.

4.1 Capturing the Variability of Core-Code Assets

4.1.1 Imperfectly Modular Variability

In realistic SPL settings, variability is implemented by using a combined set of traditional techniques, such as inheritance, overloading, generic types, design patterns. These techniques offer a form of imperfectly modular variability at the implementation level. To illustrate what imperfectly modular variability means, let us consider the implementation of a set of features in the JavaGeom [Legland, 2017], as a realistic feature-rich system. Its features are depicted in the FM on Figure 4.1, whereas the details for the development of the whole case study are given in Appendix B.

Specifically, from Figure 4.1, `StraightCurve2D` is a mandatory feature with three shown alternative features: `Line2D`, `Segment2D`, and `Ray2D`. Excerpts of their respective implementations are given in Listings 4.1 to 4.4. Focusing on the implementation techniques, the abstract class `AbstractLine2D` is a *vp* and its three variants, `StraightLine2D`, `LineSegment2D` and `Ray2D`, are created by generalizing/specializing its implementation.

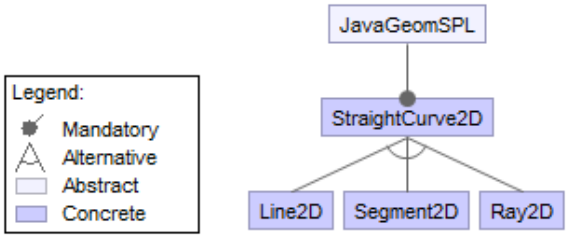


Figure 4.1: Four features of *JavaGeom* product line

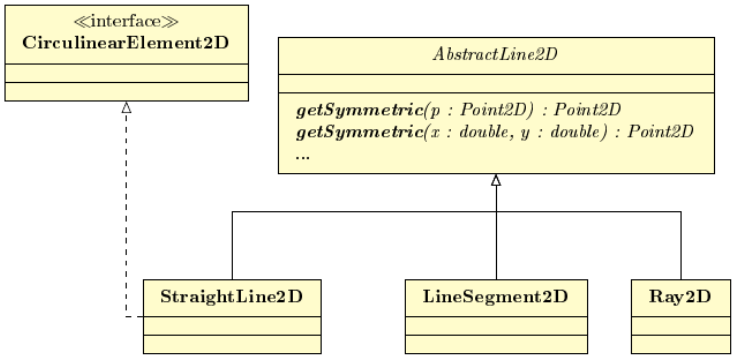


Figure 4.2: A detailed design excerpt of *JavaGeom* product line from the implementations in Listings 4.1 to 4.4

In a detailed design, their implementation technique looks as in Figure 4.2. Features in the FM (*cf.* Figure 4.1) seem to have a direct and perfect modular mapping in implementation (*cf.* Figure 4.2) (*i.e.*, «`StraightCurve2D` implemented by `AbstractLine2D`», «`Line2D` implemented by `StraightLine2D`», «`Segment2D` implemented by `LineSegment2D`», and «`Ray2D` implemented by `Ray2D`»). But, actually, this perfect form of modularity hardly exists. The *vp* `AbstractLine2D` and its *variants* have a lot of internal variability, for example, in `AbstractLine2D` the method `getSymmetric()` is another *vp* with two alternative variants implemented using the technique of overloading (*cf.* Figure 4.2 and Listing 4.1 in lines 5-7 and 9-12). The variants of `getSymmetric()` can be considered as an implementation in finer grained modules. It represents some technical and nested variability, which could be specified or not in the FM, but still needs to be documented, traced, and managed (*e.g.*, to be resolved).

Imperfect modularity comes also from the fact that a feature is a domain concept and its refinement in core 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.

For example, the feature `Line2D` uses several *vp*-s, such as `AbstractLine2D`, `Cloneable` and `CirculinearElement2D`, the variant `StraightLine2D` (*cf.* Figure 4.2 and Listing 4.3), plus their technical *vp*-s mentioned in the previous paragraph.

```java
/* File AbstractLine2D.java */
public abstract class AbstractLine2D extends AbstractSmoothCurve2D
                implements SmoothOrientedCurve2D, LinearElement2D {
    /**Return the symmetric of point p relative to this straight line.*/
    public Point2D getSymmetric(Point2D p) {
        return getSymmetric(p.x(), p.y());
    }
    /**Return the symmetric of point with coordinate (x, y) relative to
    this straight line.*/
    public Point2D getSymmetric(double x, double y) {
    double t = 2 * positionOnLine(x, y);
    return new Point2D(2 * x0 + t * dx - x, 2 * y0 + t * dy - y);
  }
  /*...*/
}
```

Listing 4.1: An excerpt from the realization of feature StraighCurve2D, for *JavaGeom* in Figure 4.1, as a variation point AbstractLine2D

```java
/* File StraightLine2D.java */
public class StraightLine2D extends AbstractLine2D
  implements SmoothContour2D, Cloneable, CircleLine2D {
  /*...*/
}
```

Listing 4.2: An excerpt from the realization of feature Line2D, for *JavaGeom* in Figure 4.1, as a variant StraightLine2D of AbstractLine2D

```java
/* File LineSegment2D.java */
public class LineSegment2D extends AbstractLine2D
     implements Cloneable, CirculinearElement2D {
  /*...*/
}
```

Listing 4.3: An excerpt from the realization of feature Segment2D, for *JavaGeom* in Figure 4.1, as a variant LineSegment2D of AbstractLine2D

```java
/* File Ray2D.java */
public class Ray2D extends AbstractLine2D
           implements Cloneable {
  /*...*/
}
```

Listing 4.4: An excerpt from the realization of feature Ray2D, for *JavaGeom* in Figure 4.1, as a variant Ray2D of AbstractLine2D

While it would be preferable to use a single variability implementation technique to implement an SPL, none of the existing techniques is yet "a preferred one" [Apel et al., 2013], which could cover all types and properties of variability that may appear in different domains. Although, to facilitate the management of

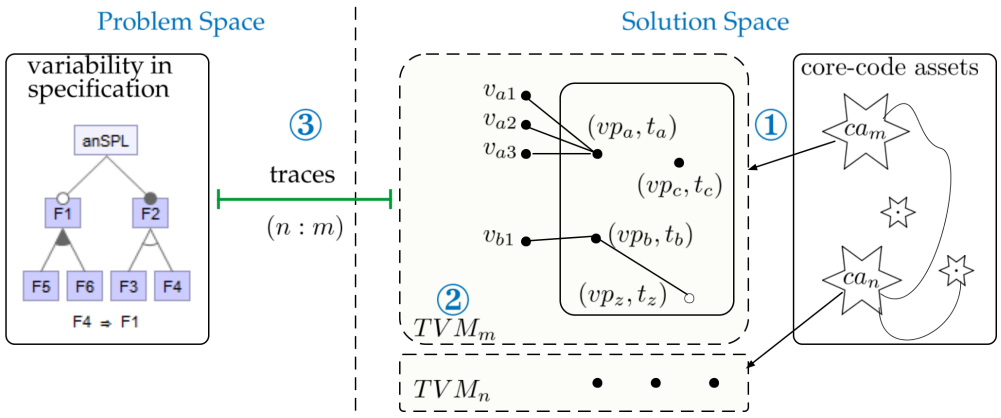


Figure 4.3: A three step framework for variability management of core-code assets ($\mathcal{TVM}_m$ *stands for Technical Variability Model (*cf. *Section 4.2.2) of the core-code asset* ca_m*, with vp-s* $\{vp_a, vp_b, ...\}$ *and their respective variants* $\{v_{a1}, v_{a2}, ...\}$ *that are realized by different techniques* $\{t_a, t_b, ...\}$*. Whereas,* $\{f_1, f_2, ...\}$ *are features in the FM.)*

variability, a smaller set of techniques is preferable for implementing an SPL [Bachmann and Clements, 2005]. This implies that an SPL architect has to deal with a variety of *vp*-s with variants, that is, implemented by different language constructs, or with different binding times. In our illustrative example (*cf.* Figure 4.2 and Listings 4.1 to 4.4) the traditional technique of inheritance and overloading are used, which induce a class and method granularity of variability with runtime and compile time binding, respectively. In these techniques, the concept of feature does 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.

In this chapter, we address the ability to capture and model this imperfectly modular variability at the implementation level, in a forward engineering process, which we define as follows:

Definition 1. *An imperfectly modular variability in implementation occurs when some variability is implemented in a methodological way, with several variability implementation techniques used in combination, the code being not necessarily shaped in terms of features, but still structured with the variability traceability in mind.* ■

Specifically, we distinguish three main steps for managing the imperfectly modular variability of core-code assets. Therefore, we provide a three step framework, which is depicted in Figure 4.3, for

1. Capturing the imperfectly modular variability at the implementation level in terms of *vp*-s with variants, as abstract concepts;

2. Modeling (documenting) this variability in terms of *vp*-s with variants, while keeping the consistency with their implementation in core-code assets, and;

3. Establishing the trace links between the specified and implemented variabilities (*i.e.*, we extend the framework for variability traceability).

In this chapter we address the first two steps of capturing and modeling the variability of core-code assets, whereas in the next chapter we address the ability to trace this variability with the specified domain features in a feature model.

4.1.2 A Framework for Capturing the Variability

Toward addressing one part of Challenge A2, we provide the following framework for capturing the variability of core-code assets when it is realized by using a combined set of traditional variability implementation techniques (*e.g.*, inheritance, overriding).

The variabilities in core-code assets that have a form of imperfect modularity usually consist of a common part and a variable part. It may happen that a whole core-code asset is also a variable asset, for example, a source file, a package, a class. The variable part consists of technique with a mechanism for creating the variants, a way for resolving the variants, and the variants themselves. We abstract them by using the concepts of *vp*-s with variants and their dependencies. The variable part may have several *vp*-s and their variants, which may have dependencies with the *vp*-s and variants in the same or in the other assets.

Let vp_x be a specific variation point and v_x one of its variants. We assume that the vp_x is implemented by a single traditional technique t_x. The set $\mathcal{T}$ of possible techniques for vp_x is then made explicit in our framework:

```
T = {Inheritance, Generic Type, Overriding, Strategy
     pattern, Template pattern,...}
```

A *vp* is not a by-product of a traditional implementation technique [Bosch et al., 2001; Sinnema et al., 2004a]. Therefore, we use the second and the fourth meaning of a *vp* (see its all five meanings in Section 2.2.2), depending on the implementation technique, for tagging the variability of core-code assets.

Remark. *Tagging a variation point or variant means to map the variation point or variant concept to a concrete variable asset element in core-code assets; respectively, to maintain their consistency.*

For example, we use the fourth meaning when t_x = `Inheritance` through tagging as a vp_x the whole superclass instead of tagging the methods inside it or its interface that is going to vary in subclasses as variants. When t_x = `Generic Type` we use the second meaning by tagging as a vp_x the parameter itself.

For example, the superclass `AbstractLine2D` is a *vp* (*cf.* Figure 4.2), and we abstract/tag it as `vp_AbsLine2D`. Similarly, we abstract its variants, `StraightLine2D`, `LineSegment2D`, and `Ray2D`, as `v_Line2D`, `v_Segment2D`, and `v_Ray2D`, respectively. A similar example is given on page 30, for the Graph PL case study.

Characteristic properties of *vp*-s with variants. Depending on the used implementation technique, the nature of a core asset element that represents a *vp* or one of its variants varies. We gathered their variety as characteristic properties of *vp*-s with variants, which are given in the previous chapter (see Section 3.1.1). We give here these properties of *vp*-s with variants in a formal way, as part of our framework.

The set of logical relations [1] ($\mathcal{LG}$) between variants in vp_x that are commonly faced in practice are similar with the relations between features in an FM, as are given in Table 3.1 (Section 3.1.1). Then,

$\mathcal{LG}$ = {Mandatory, Optional, Alternative, Multi−Coexisting}

A single technique, t_x, can offer at least one of these logical relations between variants, for example, the `Inheritance` can be used for implementing the alternative variants (as in *JavaGeom*, Figure 4.1), the `Overriding` for implementing the multi-coexisting variants, or the `Aggregation` for optional variants. Our catalog of variability implementation techniques can then be used as a guidance for chosing a technique (see Table 3.5).

The possible static and dynamic binding times ($\mathcal{BT}$) of a vp_x (see Table 3.2 in Section 3.1.1) are:

$\mathcal{BT}$ = {Compilation, Assembly, Programming time,
Configuration, Deploy, StartUp, Runtime}

Depending on whether the vp_x is meant to be evolved ($\mathcal{EV}$) in the future with new variants, it can be (see Evolution on page 33):

$\mathcal{EV}$ = {Open, Close}

For example, from the JavaGeom PL (*cf.* Figure 4.2), the `vp_AbsLine2D` has a `Class level` granularity (*cf.* Table 3.3 in Section 3.1.1), which can be resolved at `Runtime` to one of its `Alternative` variants (`v_Line2D`, `v_Segment2D`, or `v_Ray2D`) and it is `Open` for adding new variants in the future.

Other characteristic properties of a *vp* can be formalized and also added, for example, whether a variant is added, removed, or replaced by another variant. This property matters during the process of product derivation, which is a possible usage of our framework.

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

Let the set of all *vp*-s in core-code assets be:

$$\mathcal{VP} = \{vp_a, vp_b, vp_c, ..., vp_x\} \tag{4.1}$$

[1] In the literature they are also known as *types of vp*-s; but, it was more meaningful to use the term *Type of vp* for something else (see Table 4.2).

The set of all realized variants for the vp_x is:

$$\mathcal{V} = \{v_{x1}, v_{x2}, v_{x3}, ..., v_{xn}\}, \text{where } n \in \mathbb{N} \tag{4.2}$$

The set of all core-code assets, with variability or variable themselves:

$$\mathcal{CA} = \{ca_m, ca_n, ca_o, ..., ca_x\} \tag{4.3}$$

The variable elements in a core-code asset, for example, for ca_m, which represent some *vp*-s or its variants, are:

$$ca_m = \{ea_{m1}, ea_{m2}, ea_{m3}, ..., ea_{mn}\}, \text{where } n \in \mathbb{N} \tag{4.4}$$

The abstractions of each *vp* and variant from sets (4.1) and (4.2) are associated with their concrete implementation in core-code assets, for example, with a varying file, class, or method, which we named as in sets (4.3) and (4.4). Analysing some of the traditional variability implementation techniques, we determine four possible associations for *vp*-s and two for variants. They are described in the following and in Table 4.1.

- (vp_x, ea_{xn}), where n represents a specific element of the ca_x in (4.4). This is the case when vp_x represents only one element of an asset. For example, (`vp_AbsLine2D`, `AbstractLine2D`), represented by a class.

- $(vp_x, \varnothing)$, when the vp_x is implicit, hard to be tagged. For example, when an `if-else` statement is used [Svahnberg et al., 2005].

- (vp_x, ca_x), when the core-code asset ca_x itself is variable. For example, 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`, file).

- (v_{xy}, ea_{xn}), where y represents a specific variant of $\mathcal{V}$ in (4.2) and n represents a specific variable element of ca_x in (4.4). This is the case when the variant represents only one element of an asset.

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

Until now, we considered that a feature or a *vp* abstraction is associated with a single location in core-code asset, for example, (vp_x, ea_{xn}). In several techniques, this association is at least 1–to–m, that is, a feature or a *vp* can be present in several places in core-code assets.

If we analyse the second part of the *vp* definition, given in Section 2.2.1, it states that "*a variation point identifies ... more locations at which the variation will occur*" [Jacobson et al., 1997]. We found that only Muthig and Atkinson [2002] have considered explicitly this part of the definition after Jacobson et al. [1997] themselves.

Table 4.1: Tagging properties of variation points and variants

Properties	Description
Single location	*When a vp or variant is present in a single location*
Spread	*When the same vp or variant is present in several locations*
Implicit	*When a vp is implicit and cannot easily be tagged*

According to them, this means that a decision (an entity in form of a question that is used for resolving the realized variability) can have more than one *vp*, which is something else. However, Jacobson et al. [1997, p. 105] explain that, depending on the used technique, the same *vp* can be in more locations in core-code assets (*e.g.*, when the technique of parameterization is used). Therefore, we distinguish two more tagging properties:

- $(vp_x, \{ea_{x1}, ea_{x2}, ..., ea_{xn}\})$, when the vp_x is found in more than one place in core-code assets.
- $(v_{xy}, \{ea_{x1}, ea_{x2}, ..., ea_{xn}\})$, when the variant is found in more than one place in core-code assets.

For example, from Listing 1.1 (Section 1.1.2), the variant `Weighted` is implemented in two separate places: lines from 8–10 and 17–20. These properties can be recognized easily when features are implemented by using the technique of preprocessors in C. Just as an example, in the Berkley DB[2] the preprocessor directive that represent feature `DIAGNOSTIC` is present in 45 .c files or in 73 places or, feature `HAVE_QUEUE` is present in 19 .c files or in 25 places.

In a general case, the vp_x or v_{xy} can be associated with an element of any core-code asset from the $\mathcal{CA}$ in (4.3).

In this section, we presented the essential basis for capturing *vp*-s and variants at the implementation level. In some existing approaches, *vp*-s or variants are labeled depending on their resolution nature (*e.g.*, choice, substitution [CVL, 2012]), or depending on their location in assets (*e.g.*, kernel-param-vp, kernel-abstract-vp [Gomaa, 2004]). For now, we keep a more inclusive label simply as a *vp* concept, which can reflect in the future its resolution nature, location or evolution.

4.2 Modeling the Implemented Variability

After capturing the variability of a core-code asset ca_x, we now model its variability in terms of *vp*-s with variants as abstractions. Specifically, we model the relations between these abstractions by associating each *vp* with its realized variants

[2] `http://oracle.com/technology/products/berkeley-db`

Table 4.2: Types of variation points

Types	Description
Ordinary	*A vp is introduced and implemented* (i.e., *its variants are realized) by a specific technique.*
Unimplemented	*A vp is introduced but is without predefined variants* (i.e., *its variants are unknown during the domain engineering).*
Technical	*A vp is introduced and implemented only for supporting internally the implementation of another vp, which realizes some of the variability at the specification level.*
Optional	*The vp itself, not its variants, is optional* (i.e., *when it is included or excluded in a product, so are its variants).*
Nested *vp*	*When some variable part in a core code asset becomes the common part for some other variants.*

and its implementation technique (*i.e.*, the logical relation between the variants, their binding time, and the evolution property). Other properties from Tables 3.1 to 3.4 on Section 3.1.1 can also be abstracted and attached. In this way, we want to address the remained part of Challenge A2, for modeling the implemented variability of core-code assets, and Challenge A3 on keeping the consistency between the captured variability in terms of *vp*-s with variants and their concrete variability implementation techniques and core-code assets that they abstract.

4.2.1 Types of Variation Points

As a continuation of our framework, we observed that five types of *vp*-s can be distinguished for modeling the implemented variability, which are given in the set $\mathcal{X}$ below. A resolution for them is given in Table 4.2, and they are modeled as follows.

```
X = {vp, vp_unimplemented, vp_technical, vp_optional,
   vp_nested}
```

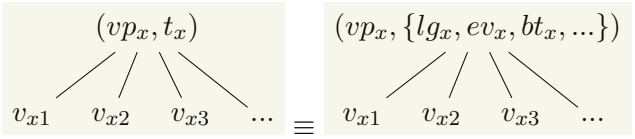
The implementation technique $t_x \in \mathcal{T}$ of the $vp_x \in \mathcal{VP}$, which relation we write as (vp_x, t_x), describes three main properties of the vp_x: the logical relation for its variants, the evolution, and the binding time. Then,

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

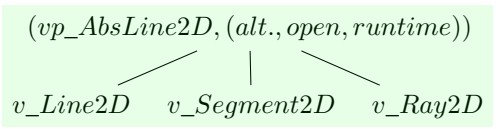
So, when the vp_x is an ordinary *vp* (*cf.* Table 4.2) we model the variability in a core-code asset as a set of its variants and the characteristic properties derived from the vp_x's implementation technique t_x. This leads to the following definition (*cf.* Equation (4.2)):

$$vp_x = \{\mathcal{V}, t_x\} = \{\{v_{x1}, v_{x2}, v_{x3}, ..., v_{xn}\}, t_x\}, \text{where } n \in \mathbb{N} \tag{4.5}$$

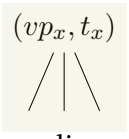
which we will graphically represent as:



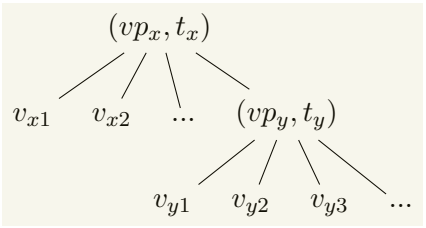
For example, from Figure 4.2,



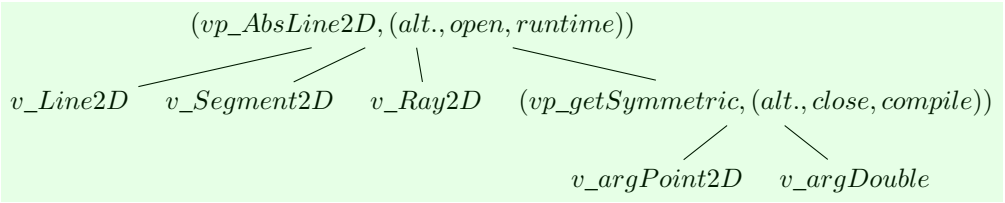
When the vp_x is an unimplemented *vp* (*cf.* Table 4.2), we model it as:

$vp_x = \{\{\varnothing\}, t_x\}$, respectively 

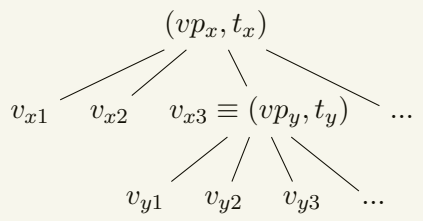
A technical *vp* is in principle an ordinary *vp* but it may be made visible or not to different stakeholders (*cf.* Figure 4.2). This depends on the purpose of variability management, for example, to resolve or to address the variability. For the sake of generalization, we consider that a technical *vp* can be also an unimplemented *vp*. A technical variation point vp_y of the vp_x is modeled as below.



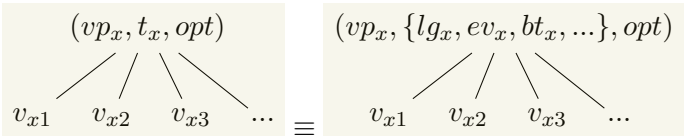
For example, let us consider that `vp_getSymmetric` is the abstraction of the *vp* `getSymmetric` (*cf.* Figure 4.2), and it is a technical *vp* of `vp_AbsLine2D`. It has two alternative variants, `v_argPoint2D` and `v_argDouble`, which are realized using the technique of overloading. This variability is then modeled as follows:



On the contrary, 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.



When vp_x is optional, the tuple (vp_x, t_x) becomes a triple (vp_x, t_x, opt). We use the acronym *opt* here in order to distinguish between the optional variants in a *vp* and the optionality of the *vp* itself. Also, to make a *vp* as optional it may not require a technique (*i.e.*, simply the *vp* is not included in a final product). Therefore, we should document it. An optional *vp* can be ordinary, unimplemented, and can have other nested or technical *vp*-s. This is modeled as:




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

Listing 4.5: The variability on file *GraphBasic.scala* (given also in Listing 1.1)

It is common that a variant can be part of several *vp*-s, such as the variant `v_Line2D` that use `vp_AbsLine2D` and `vp_CirculinearElem2D` (*cf.* Figure 4.2). This is more about modeling the variability from the perspective of a variant.

```
/* Search.scala*/
trait Search
class DFS extends Search {
  def detectCycle(gr: Graph, vr: Vertex): Boolean = false
  var elements: List[Vertex] = List()
  def dfs(g: Graph, v: Vertex, count: Int = 0): Unit = {
    v.wasVisited = true
    v.id = count
    v.printvertex; println(v.id)
    for(w <- g.getUnvisitedChild(v)) {
      if(!w.wasVisited) {
        elements = w :: elements
        dfs(g, w, count)
      }
    }
  }
}
class BFS extends Search {
  def bfs(g: Graph, v: Vertex): Unit = {
  var q: List[Vertex] = List()
    q = v :: q
    v.wasVisited = true
    while(!q.isEmpty) {
      val v: Vertex = q.last
      for(w <- g.getUnvisitedChild(v)) {
        if(!w.wasVisited) {
          q = w :: q
          w.wasVisited = true
          w.parent = v
        }
      }
    }
  }
}
```

Listing 4.6: The variability on file *Search.scala*

4.2.2 Fragmented Variability Modeling

For addressing Challenge A4, instead of modeling the whole implemented variability at once and in one place, we model it in a fragmented way. What we consider as a fragment aims at being flexible. Specifically, it is a core-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 to be modeled locally/separately.

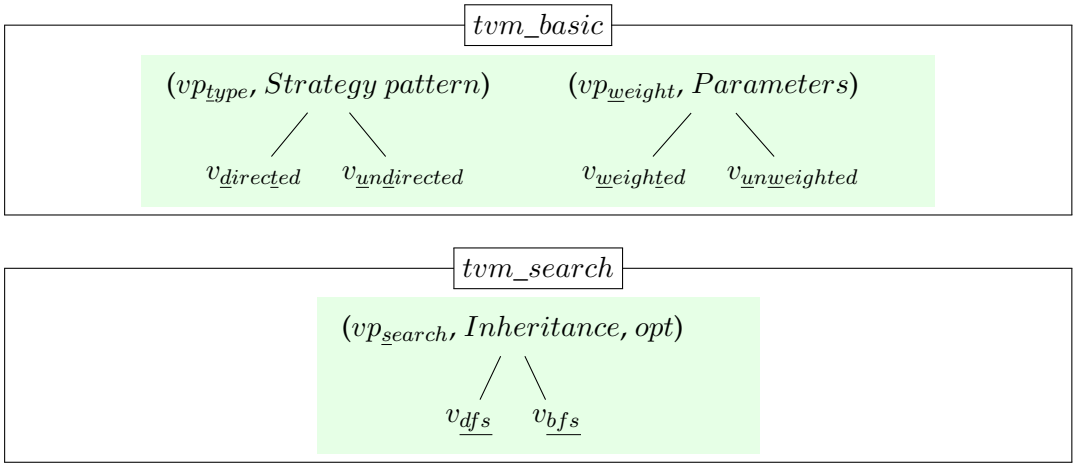


Figure 4.4: Documentation of the implemented variability in $\mathcal{TVM}s$ for the Graph SPL (*cf.* Figure 2.1)

For this reason, we designed special models, named as *Technical Variability Models* (TVM). They contain the abstractions of *vp*-s with variants, their tags for keeping the consistency with the variable elements in core-code assets (*cf.* Section 4.1.2), and describe the realized variability on specific core-code assets (*cf.* Section 4.2).

Specifically, the TVM is a power set (set of subsets) of all *vp*-s in a core-code asset. For example, let us suppose that the *vp*-s in (4.1) model the variability of the core asset ca_m in (4.3) and (4.4), then the $\mathcal{TVM}_m$ is:

$$\mathcal{TVM}_m = Pow(\mathcal{VP}_m) = \{\{\{v_{a1}, v_{a2}, v_{a3}, ..., v_{an}\}, t_a\}, \{\{v_{b1}, v_{b2}, v_{b3}, ..., v_{bn}\}, t_b\}, ..., \{v_{x1}, v_{x2}, v_{x3}, ..., v_{xn}\}, t_x\}\}, \text{ where } n \in \mathbb{N} \tag{4.6}$$

In a more illustrative way, in Figure 4.4 are shown two TVMs that document the implemented variability in files *GraphBasic.scala*, given in Listing 4.5, and *Search.scala*, given in Listing 4.6. These two TVMs, tvm_basic and tvm_search, model some of the implemented variability for the Graph PL (see Figure 2.1), and have two and one *vp*-s, respectively. One of the *vp*-s, vp_{search}, is optional, therefore the acronym *opt* for it in the tvm_search. In this case, the whole file *Seach.scala* is optional, that is, it can be included or not in a final software product. Both these TVMs are at file level.

As a result, the whole existing variability of core-code assets in $\mathcal{CA}$ (in (4.3)) is modeled in technical variability models (TVMs), which are created and maintained locally and closer to the core-code assets. In this way we address Challenge A4, for modeling the implemented variability in a fragmented way.

All these TVMs together constitute the *Main Technical Variability Model* (MTVM) that is created automatically from them. So,

$$\mathcal{MTVM} = \{\mathcal{TVM}_m, \mathcal{TVM}_n, \mathcal{TVM}_o, ..., \mathcal{TVM}_x\} \tag{4.7}$$

This MTVM contains the modeled variability of all core-code assets of an SPL. Unlike the organization of features in an FM as a tree structure, in MTVM the *vp*-s with variants reside in a forest-like structure. Moreover, the meaning of a *vp* is extended:

Definition 2. *A variation point is the location in core-code assets that represents a node for collecting, attaching to the core implementation, and configuring a set of variable units (variable elements of core-code assets) that are related in functionality. In particular, it is the place at which the variation occurs [Jacobson et al., 1997], and represents the used technique to realize the variability.* ■

4.2.3 Capturing and Modeling the Variability for Reverse Engineering

In the previous sections, we analysed the capturing and modeling of the implemented variability from a forward engineering process; that is, when the TVMs are created during the realization of the functionality in code.

During a reverse adaptation model of an SPL, the capturing and modeling of variability from code becomes important for two main reasons. First, when an existing software product, or a set of similar products, are required to be migrated to an SPL approach. Then, as variability evolves over time, new features are introduced or the existing variability is extended. When the implemented and the specified variability in domain level do not co-evolve, their mapping may deteriorate and inconsistencies appear. Thus, in both cases, a reverse approach is usually needed to reconstruct the FM (completely or partially), or an approximation of it, from the code.

Reconstructing the FM or TVMs from the implemented variability is not trivial, as the code may not be shaped in terms of features. According to a recent overview on the available approaches for detecting the concepts of *vp*-s with variants in source code [Lozano, 2011] (see Section 2.3.4), the role and characteristics of variability implementation techniques (analysed in Section 3.1.1) in the reverse engineering processes are generally not considered.

In this section, we show that our approach can be applied for capturing *vp*-s with variants, also the difference of *vp*-s with the capture of features, in a reverse engineering adaptation model. Moreover, we show the importance of capturing the variability implementation technique during reverse engineering, but we do not discuss the automation of a migration or evolution of an SPL. Therefore, the term of *capturing* is used in the meaning of extracting the variability information from code, as a first step of a reverse engineering process.

In the following, we illustrate and compare the capture of variability for the Graph PL (*cf.* Figure 2.1 in Section 2.1) in two implementation cases, when variability is implemented by using the traditional techniques of strategy pattern with parameters (*cf.* Listing 4.5) and then the emerging technique of feature modules (*cf.* Listing 4.7). In Listing 4.7 we have shown the main organization of feature modules (borrowed from Apel et al. [2013]), where each variable feature is realized as a physically separated module.

```
layer BasicGraph;
class Graph {
  Vector nodes = new Vector();
  Vector edges = new Vector();
  Edge add(Node n, Node m) {
    Edge e = new Edge(n, m);
    nodes.add(n);
    nodes.add(m);
    nodes.add(e);
    return e;
  }
 void print() {
    for(int i = 0; i < edges.size(); i++) {
        ((Edge)edges.get(i)).print();
        if(i < edges.size() - 1)
           System.out.print(" , ");
    }
  }
}
class Node {
  int id = 0;
  Node(int _id) { id = _id; }
  void print() {
    System.out.print(id);
  }
}
class Edge {
  Node a, b;
  Edge(Node _a, Node _b) { a = _a; b = _b; }
  void print() {
    System.out.print(" (");
    a.print();
    System.out.print(" , ");
    a.print();
    System.out.print(") ");
    }
}
```

```
layer Weighted;
refines class Graph {
  Edge add(Node n, Node m) {
    Edge e = Super.add(n, m);
    e.weight = new Weight();
    return e;
  }
  Edge add(Node n, Node m, Weight w) {
    Edge e = add(n, m);
    e.weight = w;
    return e;
```

```
12    }
13  }
14  refines class Edge {
15    Weight weight;
16    void print() {
17      Super.print();
18      weight.print();
19    }
20   }
21  class Weight {
22    void print() { /*...*/ }
23  }
```

```
1  layer Directed;
2  refines class Graph { /*...*/ } /*...*/
```

```
1  layer Undirected;
2  refines class Graph { /*...*/ } /*...*/
```

Listing 4.7: The Graph SPL using feature modules (borrowed from [Apel et al., 2013])

Capturing *vp*-s with variants. As in Section 3.1, we can capture the implemented variability in terms of *vp*-s with variants from Listing 4.5. The captured variability (*i.e.*, *vp*-s with variants) is given again in Tables 4.3 and 4.4. Our catalog of techniques can then help one to capture the right characteristic properties for each used technique. For example, the strategy pattern (*cf.* Table 3.5) can offer an alternative relation between variants, runtime binding, default variants, and the possibility to add new variants during the evolution. All these properties are consistent with the captured variability of `vp_edgetype`, and its variants, which are implemented by the strategy pattern (*cf.* Tables 4.3 and 4.4).

Table 4.3: Capturing the implemented variability of Graph PL in Listing 4.5

***VP*-s**	**Lines**	**Granularity**	**Binding time**	**Logical Rl.**	**Evolution**
`vp_edgetype`	26 – 27	method	runtime	alternative	Open
`vp_weight`	2	parameter	programming	alternative	Close

Capturing features. In Listing 4.7 the code is shaped in terms of features by using the technique of feature modules. Each feature module is realized by a *layer*, which is a language construct in the *Jak* language (an extension of the Java language for feature-oriented programming) [Batory et al., 2004]. In this implementation, each problem space feature (*cf.* Figure 2.1) is implemented by a feature module (*cf.* Listing 4.7), whereas their mapping is ensured by their common names. The module `BasicGraph` is the base module where can be added other feature modules as

Table 4.4: Capturing the implemented variability of Graph PL in Listing 1.1

Variants	Lines	Granularity	Default	*VP*-s
`v_directed`	6 – 13	method	No	`vp_edgetype`
`v_undirected`	14 – 25	method	No	`vp_edgetype`
`v_weighted`	2	value	No	`vp_weight`
`v_unweighted`	2	value	Yes	`vp_weight`

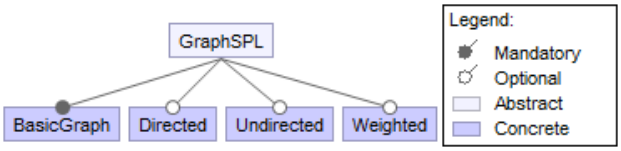


Figure 4.5: The reconstructed FM from features in Listing 4.7

variants during the product derivation. The only variability properties that can be captured in Listing 4.7 is the base module, which is mandatory, and 3 variants (`Directed`, `Undirected` and `Weighted`) that are optional with the same binding time (deployment), granularity (feature module), and no evolution or default concepts. Moreover, the feature `Unweighted` is made default by being part of the base module. In addition, in this implementation there is no concept of *vp*-s.

To reflect on the differences between capturing *vp*-s with variants and features in core-code assets, we show that they may stay together when variability has an imperfect form of modularity (defined in Section 4.1.1). Concretely, in Listing 4.5, except the captured *vp*-s with variants, we can also capture the specified features at the domain level. For example, feature `Directed` can be captured in lines 6 – 7, 11 – 13. Feature `Weighted` can be captured in lines 2, 6, 8 – 10, 14, 17 – 20. And, if we compare these lines of code with the lines of the captured *vp*-s with variants in Tables 4.3 and 4.4, it is obvious that capturing *vp*-s with variants and features is different. Whereas, in Listing 4.7 there is no concept of variation point, but only of variant features. Thus, by capturing features in core-code assets we capture only those lines of code that implement a problem oriented feature, while by capturing a *vp* with variants we capture the implementation technique that gives a rich set of properties to them (*cf.* Tables 4.3 and 4.4).

Reconstructing the feature model. As for reverse engineering, we decide to assess an implementation technique regarding their support to reconstruct an FM from the captured variability in core-code assets. This will also points out the differences between the captured variability in terms of features and *vp*-s with variants.

For example, from the captured variability in Tables 4.3 and 4.4 we can reconstruct an FM similar to the original FM on Figure 2.1. The only difference between these feature models is that feature `Unweighted` is realized as a default variant

`v_unweighted`. The successful reverse engineering of the FM from this captured variability is made possible mainly because the logical relations between *vp*-s with variants are known (*cf.* Table 3.1 in Section 3.1.1). The other captured properties can be used for other reasons (*e.g.*, the binding time during the product derivation).

Similarly, in Figure 4.5 is shown the reconstructed FM from the captured features and their properties in Listing 4.7. This reconstructed FM is hardly similar to the original FM in Figure 2.1 because the feature modules expose a single logical relation between them. Therefore, a feature oriented implementation is always associated with the original FM, which is used to configure and manage the feature modules.

These examples indicate that an implementation technique influences the capturing of variability and the reconstruction of a good approximation of the FM with the original one, in case of its deterioration over time. Furthermore, as a complement and comparison to the definition of variation point (*vp*) that is given in Definition 2, we state the following:

Definition 3. *A solution space feature is used to locate the parts (lines) of code in core-code assets, independently of the implementation technique, that belong to a single variable unit* (i.e., *a problem space feature), which is going to vary between products in a PL family.* ■

4.3 Summary

Managing the implemented variability is an important part of the development process in SPL engineering. While features at the specification level are merely concepts, at the implementation level a combination of traditional variability implementation techniques are actually used in many realistic SPLs, thus leading to a form of, what we call, an *imperfectly modular variability* at the implementation level.

In this chapter, we gave a framework for capturing and modeling this imperfectly modular variability. Specifically, we addressed the modeling of the implemented variability from its early step of capturing and then modeling it in a fragmented way, in technical variability models (TVMs). Then, we give an illustrative example for capturing *vp*-s with variants and features in a reverse engineering process, when variability is realized using the strategy pattern with parameters (in first implementation) and feature modules (in second implementation). We use these two implementations for highlighting the differences between capturing the implemented variability in terms of features and *vp*-s with variants, and then the importance of considering the variability implementation techniques during a reverse engineering process.

We believe that our framework is general enough and can be used for different management reasons of the implemented variability, such as for product derivation, consistency checking, or addressing the variability. In the following Chapter 5, we show an approach for using the TVMs in tracing the implemented *vp*-s with variants with the features in the FM at the specification level. Then, in Chapter 6, we use the TVMs and these trace links for checking the consistency of the

implemented variability against the specified features in the FM.

Part II:

Usage of Technical Variability Models

CHAPTER 5

Traceability with Technical Variability Models

This chapter shares material with the following paper:

Tërnava, Xh. and Collet, P. (2017). Tracing imperfectly modular variability in software product line implementation. In *International Conference on Software Reuse*, pages 112–120. Springer

---

In this chapter, we show the usage of technical variability models (TVMs) for supporting the variability management. In particular, we use them to trace the specified variability, in terms of features at the domain level, to the imperfectly modular variability, in terms of *vp*-s with variants, at the implementation level.

5.1 A Three Step Traceability Approach

In the previous Chapter 4, we addressed the first two steps of our three step framework, illustrated in Figure 4.3, for managing the variability of core-code assets. They resemble the capture and documentation of the *imperfectly modular variability* (*cf.* Definition 1) in *technical variability models* (TVMs). In this chapter, we address the last step of this approach. That is, the mapping between the modeled variability at the specification level (*i.e.*, features in an FM) and the TVMs at the implementation level (*i.e.*, *vp*-s with variants), by establishing the trace links between them, as is illustrated in Figure 5.1. In this way, we aim to address the Challenge B1 for variability traceability.

Variability traceability is usually organized around two types of trace links: *realization*, and *use* [Anquetil et al., 2010; Pohl et al., 2005] (*cf.* Section 2.3.1). The first 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. 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, from which it is elicited (see Figure 2.4 in Section 2.3). Usually, the *realization* trace link is implied by "variability traceability" and in the same meaning we use it also here.

Let us suppose that $f_x \in \mathcal{FM}$ is a variable feature at the specification level, then

$$\mathcal{FM} = \{f_1, f_2, f_3, ..., f_n\}, \text{ where } n \in \mathbb{N} \tag{5.1}$$

is the set of features in the feature model. Whereas, a single compound feature f_x with its variant features and their logical relation (*i.e.*, *logic*, which can have any of

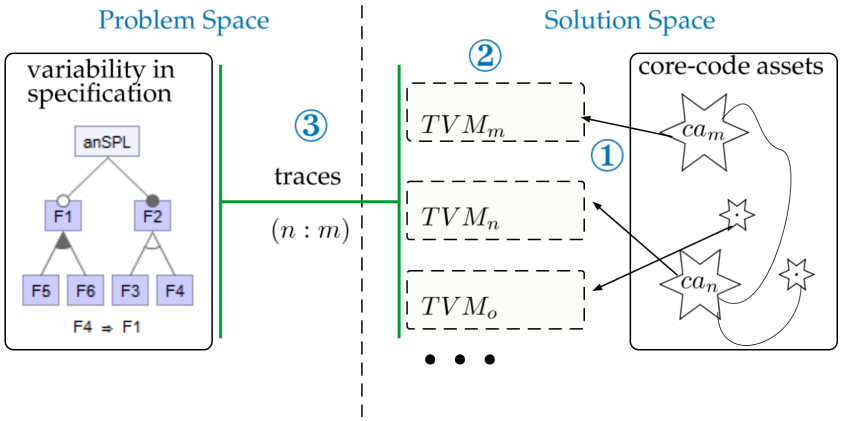
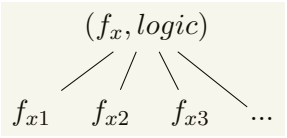


Figure 5.1: Proposed traceability approach through using the technical variability models (TVMs)

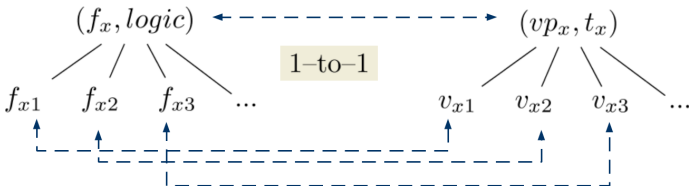
the values from Table 3.1 in Section 3.1.1) in the feature model, in an illustrative way, is given as in the following:



For mapping features to *vp*-s and variants in TVMs, we use a single bidirectional type of trace links `implementedBy` ($\longmapsto$) or `implements` ($\longmapsfrom$), which presents the *variability realization* trace link at the implementation level.

In general, this mapping of variability from the specification level to the implementation level is n–to–m (*cf.* step ③ in Figure 5.1). Specifically, from the analysed diversity of *vp*-s with variants in Chapter 4, the variability traceability may consist of (1) the ideal, 1–to–1 mapping, (2) the 1–to–m mapping, and/or (3) the n–to–1 mapping.

1–to–1 Mapping. In an ideal mapping, each specified feature at the domain level is realized by a single variable unit (*i.e.*, variable element in core-code assets) that is represented by a *vp* or variant concept at the implementation level. Such mapping looks as in the following:



In this case, we say that a compound feature f_x and its 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. For instance, for feature f_x we write:

$$(f_x \longmapsto vp_x) \text{ or } \texttt{implementedBy}(f_x, vp_x), \quad \text{respectively } \texttt{implements}(vp_x, f_x) \tag{5.2}$$

For example, by using the first meaning (*i.e.*, $\longmapsto$), from Figure 2.1 (Section 2.1) and Listing 4.5 for the GraphPL case study, then (`EdgeType` $\longmapsto$ `vp_edgetype`). Likewise, from Figure 4.1 and Figure 4.2 in JavaGeom case study, (`StraightCurve2D` $\longmapsto$ `vp_AbsLine2D`).

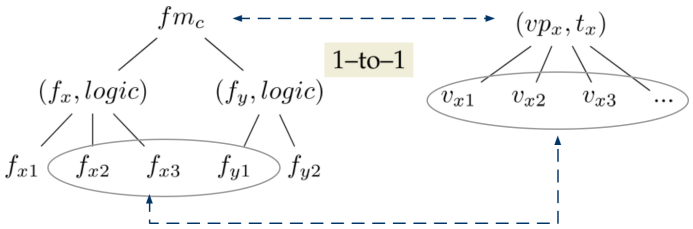
Similarly, a variant feature f_{xn} can be implemented by a single variant v_{xn} of vp_x (as in the illustration above), that is,

$$(f_{xn} \longmapsto v_{xn}) \text{ or } \texttt{implementedBy}(f_{xn}, v_{xn}), \quad \text{respectively } \texttt{implements}(v_{xn}, f_{xn}) \tag{5.3}$$

For example, from Figure 2.1 (Section 2.1) and Listing 4.5, (`Directed` $\longmapsto$ `v_directed`). Likewise, from Figure 4.1 and Figure 4.2, (`Line2D` $\longmapsto$ `v_Line2D`).

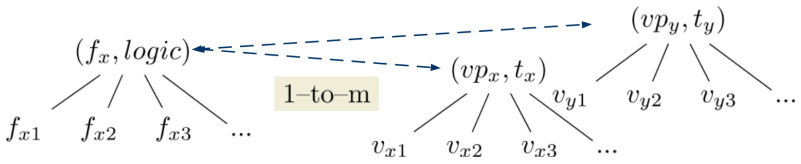
The vp_x and v_{xn} can be from different TVMs. Therefore, we may need to specify, for example, that $vp_x, v_{xn} \in \mathcal{TVM}_m$. Moreover, from Section 4.1.2, the *vp*-s with variants in a TVM can be from different core-code assets.

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 the following.



According to Capilla et al. [2013, Ch. 3], 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 the implementation of *vp*-s with variants.

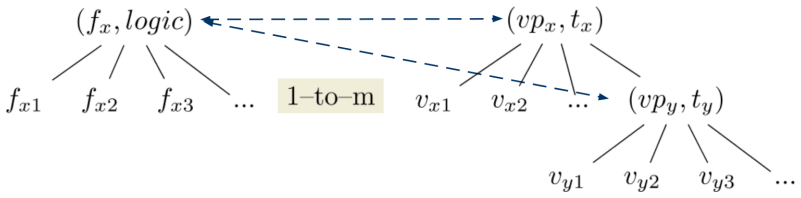
1–to–m Mapping. In this mapping, each feature at the domain level is realized by more than one *vp* or variant at the implementation level. For example, in the following is illustrated the case when a feature is mapped to two *vp*-s.



In this case, a feature f_x is implemented by several *vp*-s, which can be from the same core-code asset or not, then

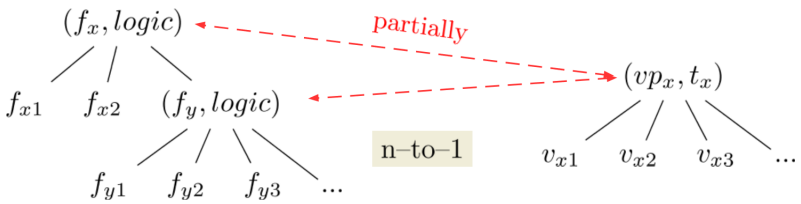
$$(f_x \longmapsto \{vp_x, vp_y, vp_z, ...\}) \text{ or } \texttt{implementedBy}\ (f_x, \{vp_x, vp_y, vp_z, ...\}) \tag{5.4}$$

In our observations, this case happens depending on the type of *vp*-s (given in 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, (`StraightCurve2D` $\longmapsto$ {`vp_AbsLine2D`, `vp_getSymmetric`}), from Figure 4.1 and Figure 4.2.

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.



We trace them as

$$(\{f_x, f_y, f_z, ...\} \longmapsto vp_x) \text{ or } \texttt{implements}\ (vp_x, \{f_x, f_y, f_z, ...\}) \tag{5.5}$$

The overall mapping between features and *vp*-s with variants (see step ③ in Figure 5.1) is expected to be a partial mapping. This is because, some features in FM can be abstract features (*i.e.*, do not require an implementation) [Thum et al., 2011], or they can be deferred to be implemented later, for example, when a variation point is unimplemented (*cf.* Table 4.2).

5.2 Establishing Trace Links

An automatic approach for establishing the trace links 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 Equation (5.2) with the example, we used the same names for *vp*-s and variants with features, except that we added the prefix `vp_` for a variation point and `v_` for a variant. Whereas, in the case of the multiple mapping (*i.e.*, 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 5.1), for establishing the trace links are required both variability models at the specification and implementation levels, that is, the FM and TVMs, respectively. Whereas, the established trace links are kept separately and orthogonal with the FM and the implemented variability in core-code assets that is modeled in TVMs.

5.3 Summary

This three step traceability approach (*cf.* Figure 5.1) can be used for managing the variability at the implementation level, from the early steps of capturing the imperfectly modular variability in core-code assets (*cf.* Sections 4.1.1 and 4.1.2), modeling their variability in TVMs (*cf.* Section 4.2), and finally tracing it with features in the FM (*cf.* Section 5.1).

It also can be extended and used for several management reasons during an SPL engineering. For example, these trace links can be used for product derivation, addressing the variability, consistency checking, evolution, etc. In the following Chapter 6, we use them for detecting the inconsistencies between the specified and implemented variabilities, so to continue with their respective implementation and application in Part III.

CHAPTER 6

Consistency Checking

This chapter shares material with the following paper:

Tërnava, Xh. and Collet, P. (2017a). Early consistency checking between specification and implementation variabilities. In *Preceedings of the 21st International Systems and Software Product Line Conference - Volume A*, SPLC '17, pages 29–38. ACM

Previously, in Chapter 4, we have described our framework for modeling the *imperfectly modular variabiliy* (*cf.* Chapter 3) in terms of variation points (*vp*-s) with variants in technical variability models (TVMs). Then, as part of a three step approach, we used these models to achieve variability traceability in core-code assets (*cf.* Chapter 5). In this chapter, we exploit the TVMs and the trace links to achieve an early consistency checking during the development process (*cf.* Challenge C3), between the specified and implemented variabilities (*cf.* Challenge C1), when several variability implementation techniques are used together (*cf.* Challenge C2). We present a tooled approach to check the consistency of variability while addressing the three Challenges in C. The proposed method takes the implemented variability that is modeled in terms of variation points with variants, in a forest-like structure, and uses slicing to partially check the resulting propositional formulas at both levels. As a result, it offers an early and automatic detection of inconsistencies when the mapping of variability between both levels is 1–to–1, and then with an extension to 1 – to – m mapping.

6.1 Assumptions and Issues of Consistency Checking

Within an SPL, it is important to be able to check whether the specified and implemented variabilities are consistent. For this reason, we consider that the variability of software products at the specification level is modeled in a feature model (FM), using the concept of features, for example, the FM for Graph PL in Figure 6.1. Whereas, at the implementation level variability is modeled in technical variability models (TVMs), in terms of variation points (*vp*-s) with variants, for example, the TVMs for the Graph PL in Figure 6.2. All TVMs of an SPL constitute the Main TVM (MTVM) at the implementation level. Therefore, the modeled variability in terms of *vp*-s with variants has a forest-like structure, unlike the tree structure of features in an FM. In this way, we aim in addressing the Challenge C2 for checking the variability consistency of core-code assets when a combined set of traditional variability implementation techniques are used.

Further, the specified features in an FM and their implementation as *vp*-s with variants that are modeled in TVMs may use different names and have an n–to–m

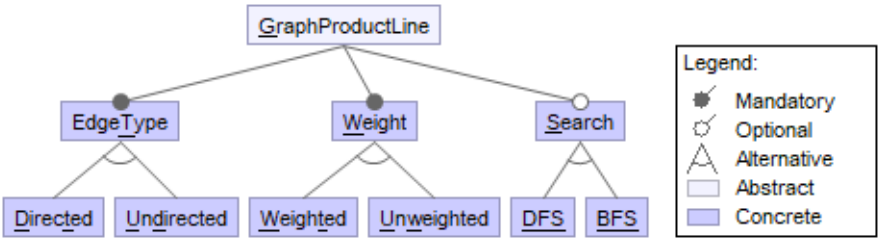


Figure 6.1: A simplified feature model of the Graph PL (*cf.* Figure 2.1)

mapping. Therefore, we achieve their mapping by using the trace links (*cf.* Chapter 5). Moreover, we assume that,

Assumption 1. *The mapping between features in a feature model (FM) and the variation points (vp-s) with variants in technical variability models (TVMs) is performed by bidirectional trace links ϕ_{TL} that are established before doing the consistency checking and are themselves consistent.*

What we mean by the consistency of trace links is that the *vp*-s with variants are traced correctly to the features that they implement or vice versa. Generally, the mapping between features and *vp*-s with variants is n – to – m. In our approach, we exclusively consider the single links (*i.e.*, 1 – to – 1) and multiple links (*i.e.*, 1 – to – m), as they are the two common forms of mapping. A *vp* may partially implement several features (*i.e.*, n – to – 1 mapping [Capilla et al., 2013, Ch. 3], see also Chapter 5), which case was not present in any of our targeted SPL implementations.

Following an inconsistency management approach [Spanoudakis and Zisman, 2001], we define a consistency rule, which represents what must be satisfied by the variabilities at specification and implementation levels.

Consistency Rule. *Within an SPL, where the specified domain variability and the implemented variability convey the same functionality, they also should represent the same set of software products.*

An inconsistency (*i.e.*, when the consistency rule is not hold) concerns a specific feature configuration for which it is impossible to derive a concrete software product from the existing core-code assets, despite the fact that the whole specified variability is implemented. We thus propose a method, based on propositional logic, for checking and detecting the places of such variability inconsistencies.

Conversion to propositional logic. Toward an automated reasoning (*cf.* Section 2.4), we convert the modeled variability at the specification and implementation levels (*i.e.*, the FM and the TVMs or the MTVM, respectively) into propositional logic. And then, we describe the consistency rule in a formal way. In this way, the whole issue of consistency checking is translated to analysing the propositional formulas ϕ_{FM} and ϕ_{MTVM} if the consistency rule between them hold.

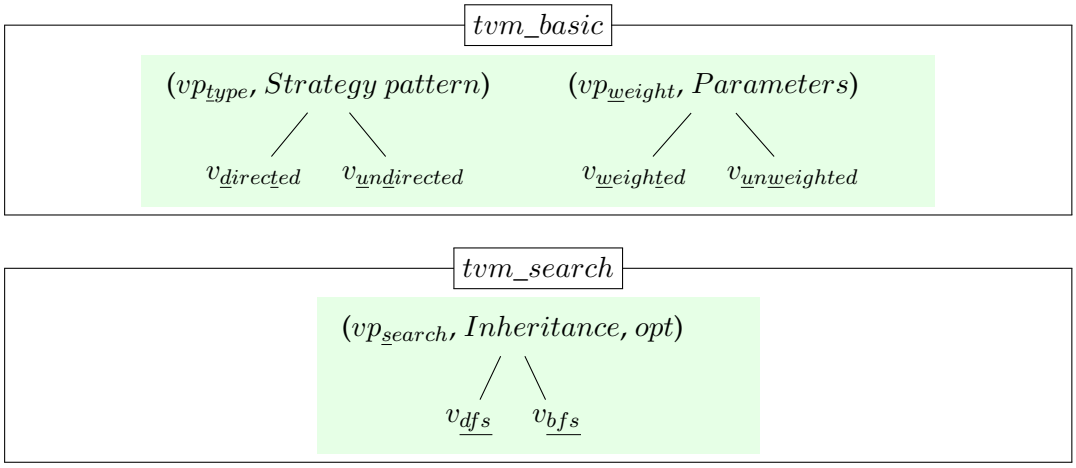


Figure 6.2: Documentation of the implemented variability from Listing 4.5 (Page 55) in TVMs for the Graph PL (*cf.* Figure 6.1)

For example, by using the underlined letters for the feature names in the simplified Graph PL in Figure 6.1 (*e.g.*, *dt* stands for `Directed`), as variables in the formula, and the translation rules in Table 2.1 (Page 23), the propositional formula for the Graph PL, ϕ_{FM_g} [1], in conjuctive normal form (CNF) is:

$$\begin{aligned}\phi_{FM_g} = \underline{g} \wedge \underline{(t \leftrightarrow g)} \wedge \underline{(w \leftrightarrow g)} \wedge \underline{(s \rightarrow g)} \wedge \\ (t \leftrightarrow (dt \vee ud)) \wedge (\neg dt \vee \neg ud) \wedge (w \leftrightarrow (wt \vee uw)) \wedge \\ (\neg wt \vee \neg uw) \wedge \underline{(s \leftrightarrow (dfs \vee bfs))} \wedge \underline{(\neg dfs \vee \neg bfs)}\end{aligned} \tag{Ex. 6.1}$$

Similarly, we apply the same conversion rules (*cf.* Table 2.1 in Page 23) to translate the TVMs to a propositional formula. In this case, the names of *vp*-s with variants are used as variables in the formula. Thus, each TVM is translated to a formula, ϕ_{TVM_x}, whereas the conjunction of all their formulas constitutes the Main TVM formula, ϕ_{MTVM}, for a whole SPL.

For example, for the features in Figure 6.1, their technical variability models (TVMs) at the implementation level (*cf.* Listing 4.5) are given in Figure 6.2. Then, the propositional formulas for these TVMs, the $\phi_{TVM_{basic}}$ and the $\phi_{TVM_{search}}$, are given in (*Ex. 6.2*) and (*Ex. 6.3*), respectively.

$$\begin{aligned}\phi_{TVM_{basic}} = vp_t \wedge vp_w \wedge \\ (vp_t \leftrightarrow (v_{dt} \vee v_{ud})) \wedge (\neg v_{dt} \vee \neg v_{ud}) \wedge \\ (vp_{wt} \leftrightarrow (v_{wt} \vee v_{uw})) \wedge (\neg v_{wt} \vee \neg v_{uw})\end{aligned} \tag{Ex. 6.2}$$

[1] The underlined parts of the formula will be explained in the following.

$$\phi_{TVM_{search}} = vp_{root} \wedge (vp_s \rightarrow vp_{root}) \wedge (vp_s \leftrightarrow (v_{dfs} \vee v_{bfs})) \wedge (\neg v_{dfs} \vee \neg v_{bfs}) \quad (Ex.\ 6.3)$$

A mandatory *vp* in a TVM is shown as a single positive literal in a propositional formula (*cf.* Table 2.1 in Page 23), for example, the vp_t or vp_w in (*Ex. 6.2*) , whereas an optional *vp* needs another parent feature to become optional. For this reason, we inserted a mandatory root *vp* that is the common root for all *vp*-s.

For instance, the vp_{root} in (*Ex. 6.3*), which is used to make optional the $vp_{\underline{s}earch}$ in Figure 6.2.

When the *vp* is mandatory, the vp_{root} does not make any difference in the formula, and thus we did not show it in (*Ex. 6.2*).

On the other hand, the bidirectional trace links (given already as a logical equivalence, $\leftrightarrow$, in Chapter 5) for all features and *vp*-s with variants are converted as a conjunction of each individual mapping.

In our example, ϕ_{FM_g} (*Ex. 6.1*) has 1 – to – 1 mapping to $\phi_{TVM_{basic}}$ (*Ex. 6.2*) and $\phi_{TVM_{search}}$ (*Ex. 6.3*) (*cf.* Table 6.1). Therefore, their single and bidirectional trace links ϕ_{TL_g} are:

$$\begin{aligned} \phi_{TL_g} = \underline{(g \leftrightarrow vp_{root})} \wedge \\ (t \leftrightarrow vp_t) \wedge (w \leftrightarrow vp_w) \wedge \underline{(s \leftrightarrow vp_s)} \wedge \\ (dt \leftrightarrow v_{dt}) \wedge (ud \leftrightarrow v_{ud}) \wedge (wt \leftrightarrow v_{wt}) \wedge \\ (uw \leftrightarrow v_{uw}) \wedge \underline{(dfs \leftrightarrow v_{dfs})} \wedge \underline{(bfs \leftrightarrow v_{bfs})} \end{aligned} \quad (Ex.\ 6.4)$$

The consistency rule then corresponds to the fact that the domain variability, ϕ_{FM}, and the implemented variability, ϕ_{MTVM}, must be semantically equivalent. In the propositional logic, the two formulas ϕ_{FM} and ϕ_{MTVM} are *semantically equivalent*, in symbols $\phi_{FM} \equiv \phi_{MTVM}$, if and only if they have the same set of interpretations [2] [Ben-Ari, 2012].

Let $[\![\phi_{FM}]\!]$ be the set of feature configurations for ϕ_{FM} (*i.e.*, the set of interpretations for the propositional formula that represents the FM), and $[\![\phi_{MTVM}]\!]$ be the set of *vp*-s with variants configurations for ϕ_{MTVM} (see an example from the given Graph PL in Table 6.1). Every feature configuration in $[\![\phi_{FM}]\!]$ should have a respective mapping to a *vp*-s with variants configuration in $[\![\phi_{MTVM}]\!]$, while considering that the mapping between features and *vp*-s with variants can be n–to–m. But, instead of comparing their configurations, we need a consistency checking approach at the formula level.

The ϕ_{FM} and ϕ_{MTVM} use different variability abstractions (features and *vp*-s with variants, respectively), which may use different names and have an n – to –

[2] Also known as the models of a propositional formula

Table 6.1: Features and *vp*-s with variants configurations in Graph PL

$[\![\phi_{FM_g}]\!]$	$[\![\phi_{MTVM_g}]\!] = [\![\phi_{TVM_{basic}}]\!] \cup [\![\phi_{TVM_{search}}]\!]$
$\{(g,t,w,dt,wt),$	$\{(vp_{root},vp_t,vp_w,v_{dt},v_{wt}),$
$(g,t,w,dt,uw),$	$(vp_{root},vp_t,vp_w,v_{dt},v_{uw}),$
$(g,t,w,ud,wt),$	$(vp_{root},vp_t,vp_w,v_{ud},v_{wt}),$
$(g,t,w,ud,uw),$	$(vp_{root},vp_t,vp_w,v_{ud},v_{uw}),$
$(g,t,w,dt,wt,s,dfs),$	$(vp_{root},vp_t,vp_w,v_{dt},v_{wt},v_s,v_{dfs}),$
$(g,t,w,dt,uw,s,dfs),$	$(vp_{root},vp_t,vp_w,v_{dt},v_{uw},v_s,v_{dfs}),$
$(g,t,w,ud,wt,s,dfs),$	$(vp_{root},vp_t,vp_w,v_{ud},v_{wt},v_s,v_{dfs}),$
$(g,t,w,ud,uw,s,dfs),$	$(vp_{root},vp_t,vp_w,v_{ud},v_{uw},v_s,v_{dfs}),$
$(g,t,w,dt,wt,s,bfs),$	$(vp_{root},vp_t,vp_w,v_{dt},v_{wt},v_s,v_{bfs}),$
$(g,t,w,dt,uw,s,bfs),$	$(vp_{root},vp_t,vp_w,v_{dt},v_{uw},v_s,v_{bfs}),$
$(g,t,w,ud,wt,s,bfs),$	$(vp_{root},vp_t,vp_w,v_{ud},v_{wt},v_s,v_{bfs}),$
$(g,t,w,ud,uw,s,bfs)\}$	$(vp_{root},vp_t,vp_w,v_{ud},v_{uw},v_s,v_{bfs})\}$

m mapping (*e.g.*, compare their names in the given example in Table 6.1, although their mapping is 1–to–1). Therefore, we could check their semantic equivalence only under the existence of their trace links. Consequently, the formal definition of the consistency rule becomes:

$$\begin{aligned} \phi_{FM} \wedge \phi_{TL} &\equiv \phi_{MTVM} \wedge \phi_{TL} \\ \text{or, } \phi_{FM} \wedge \phi_{TL} &\equiv (\phi_{TVM_1} \wedge \phi_{TVM_2} \wedge ... \wedge \phi_{TVM_n}) \wedge \phi_{TL} \end{aligned} \tag{6.1}$$

As the trace links are bidirectional (*i.e.*, features and *vp*-s with variants are mapped to each other as $f \leftrightarrow vp$), then (6.1) is valid thanks to the substitution theorem in propositional logic [Kleene, 2002]:

$$\phi_{FM(f)} \wedge (f \leftrightarrow vp) \equiv \phi_{MTVM(vp)} \wedge (f \leftrightarrow vp) \tag{6.2}$$

In other words, ϕ_{FM} and ϕ_{MTVM} are consistent if within the context of their trace links ϕ_{TL}, ϕ_{FM} and ϕ_{MTVM} represent the same variability.

According to the consistency rule in (6.1), ϕ_{FM_g} in (*Ex. 6.1*) represents the same software products with $\{\phi_{TVM_{basic}}, \phi_{TVM_{search}}\} \in \phi_{MTVM_g}$ in (*Ex. 6.2*) and (*Ex. 6.3*) when

$$\phi_{FM_g} \wedge \phi_{TL_g} \equiv (\phi_{TVM_{basic}} \wedge \phi_{TVM_{search}}) \wedge \phi_{TL_g} \tag{*Ex. 6.5*}$$

That is, within the context of trace links ϕ_{TL_g}, ϕ_{FM_g} is equivalent to $(\phi_{TVM_{basic}} \wedge \phi_{TVM_{search}})$. In details, their equivalence or conistency is shown in Table 6.1, too.

6.2 Proposed Method

For checking the consistency of the entire variability between both levels, by using the Equation (6.1), it is required that (1) the whole specified variability in an FM to be implemented and documented in an MTVM, and (2) all their trace links are established. This restricts the consistency checking to a complete system, which itself is likely to be represented by large variability models, harder to check, but also harder for tracing and fixing inconsistencies after all of them are shown at once [Vierhauser et al., 2010]. In addition, it is common that the implementation of some variability is deferred (*e.g.*, during the application engineering phase) and some partial checking is then highly desirable. Besides, even for illustrative SPLs with a small set of features, the propositional formula to compute Equation (6.1) becomes already quite large.

For example, in our Graph PL, to perform a total checking we need to find the logical equivalence ($\leftrightarrow$, and *cf.* (*Ex. 6.5*)) between the FM with trace links, which have 35 clauses in CNF (*cf.* (*Ex. 6.1*), (*Ex. 6.4*)), and the two TVMs and trace links, with 33 clauses in CNF (*cf.* (*Ex. 6.2*), (*Ex. 6.3*), (*Ex. 6.4*)).

Moreover, in realistic SPLs, checking for inconsistencies only within a single FM has still scalability issues [Benavides et al., 2010].

To overcome these problems, we propose a consistency checking method for detecting the variability inconsistencies earlier during the development process. Its main steps are based on slicing, substitution, and assertion properties, which are depicted in Figure 6.3. First, we will explain the method when single links are used, to extend it to multiple links just after.

6.2.1 Initial Checking

As a prerequisite, we check first if ϕ_{FM} and ϕ_{MTVM} individually are consistent. To do so, we use state of the art methods to check if each of them in isolation is valid (*i.e.*, satisfiable), as well as free of dead and false-optional (a.k.a common) features or *vp*-s with variants, respectively (*cf.* Section 2.4 and Figure 6.3). We also check whether the Assumption 1 about trace links holds; that is, they are established, bidirectional, and consistent.

A trace link is by default translated into a propositional formula as an equivalence (*i.e.*, $\leftrightarrow$, as in (*Ex. 6.4*)). Therefore, when some variability is selected to be checked, it is first checked whether it is traced. If ϕ_{FM} and ϕ_{MTVM} are free of such individual inconsistencies, we can proceed to the variability consistency checking between them.

6.2.2 Slicing

The originality of our method lies in the fact that we can select a single TVM, as the ϕ_{TVM_x} in Figure 6.3, or a subset of them from the MTVM in an SPL, so to check the consistency of their variability against the specified variability in an FM.

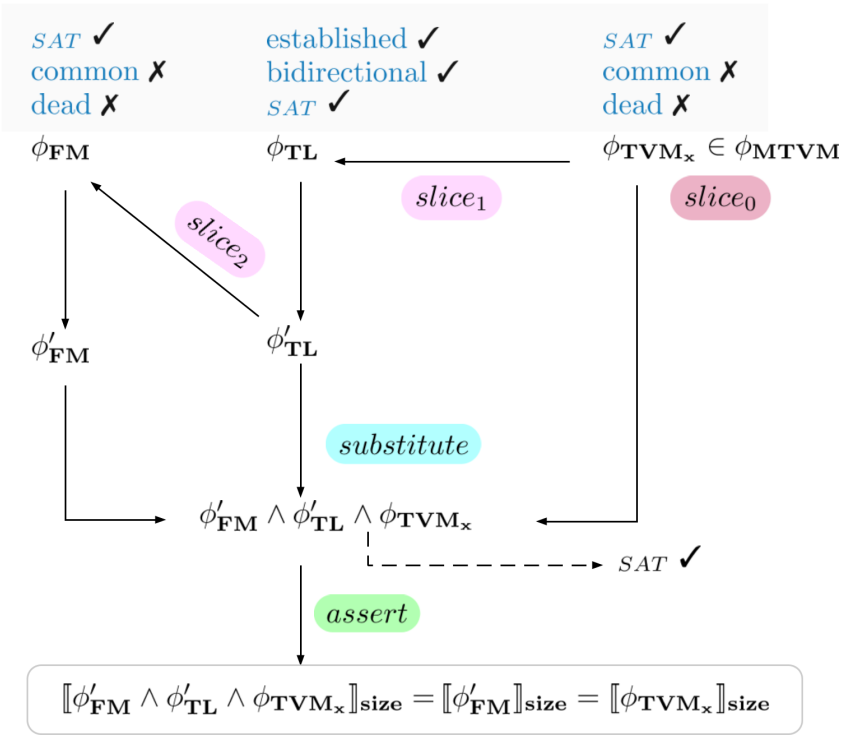


Figure 6.3: Proposed consistency checking method

According to [Acher, 2011; Acher et al., 2011], the purpose of slicing in SPL engineering is to produce a feature model that contains only a subset of relevant features for some stakeholders. This is made possible by defining the set of features that are of interest, known as the *slicing criterion*, and then computing a new feature model with only the relevant features, known as a *sliced feature model*.

In our case, the first slicing criterion is the selection of a TVM or subset of them among the other TVMs, whereas the slice is that selected TVM or subset of them that we want to check their variability consistency. This selection corresponds to the first slicing step, $slice_0$ in Figure 6.3, which is manual in our method. The advantage of selecting a single TVM instead of the whole MTVM is that, the *initial checking* has to be done only for the selected ϕ_{TVM_x} and the Assumption 1 about trace links must be met only for this TVM.

In the second step, $slice_1$ in Figure 6.3, we use the ϕ_{TVM_x} to simplify the formula for trace links ϕ_{TL} by selecting only those trace links that are relevant for the ϕ_{TVM_x}. As a result, the new sliced formula for trace links, ϕ'_{TL}, is generated (*cf.* Figure 6.3). Further, we slice the FM, during the $slice_2$, using the ϕ'_{TL} relevant trace links. The result is a new smaller formula, slice ϕ'_{FM}, which contains only the relevant features for the *vp*-s with variants in ϕ_{TVM_x}, against which they should be checked.

Slicing an FM is an operation that has recently drawn attention in the SPL community. In the literature, there are already some well defined and validated algo-

rithms [Acher et al., 2011; Krieter et al., 2016]. While we have experimented with them, we used a new slicing algorithm based on clause selection in a conjunctive normal form formula because of the trace links, as will be explained in the following.

For example, let us suppose that we want to check the consistency of $\phi_{TVM_{search}}$ in (*Ex. 6.3*) against its specification in ϕ_{FM_g} as in (*Ex. 6.1*), which is the step $slice_0$. By applying $\phi_{TVM_{search}}$ to slice ϕ_{TL_g} in (*Ex. 6.4*) we get the new formula ϕ'_{TL_g} (*Ex. 6.6*), during the step $slice_1$, which keeps only those clauses that contain the *vp*-s with variants that are in $\phi_{TVM_{search}}$. So, the generated ϕ'_{TL_g} contains only the underlined clauses in (*Ex. 6.4*). As we can see, by clause selection we keep the links to feature names, in this example, to the features g, s, dfs, and bfs; although, they are not the selected variables. Thus,

$$\phi'_{TL_g} = (g \leftrightarrow vp_{root}) \wedge (s \leftrightarrow vp_s) \wedge (dfs \leftrightarrow v_{dfs}) \wedge (bfs \leftrightarrow v_{bfs}) \quad \textit{(Ex. 6.6)}$$

where ϕ'_{TL_g} is a subformula of ϕ_{TL_g} if $\phi'_{TL_g} \in Sub(\phi_{TL_g})$. And, $Sub(\phi_{TL_g})$ is the set of subformulas for ϕ_{TL_g} [Ben-Ari, 2012].

Similarly, by applying the new formula, ϕ'_{TL_g}, to the ϕ_{FM_g} in (*Ex. 6.1*), we select only the relevant clauses for the features in these trace links (*i.e.*, only the underlined clauses of ϕ_{FM_g}). Thus, the slice ϕ'_{FM_g} is:

$$\phi'_{FM_g} = g \wedge (t \leftrightarrow g) \wedge (w \leftrightarrow g) \wedge (s \rightarrow g) \wedge (s \leftrightarrow (dfs \vee bfs)) \wedge (\neg dfs \vee \neg bfs) \quad \textit{(Ex. 6.7)}$$

where ϕ'_{FM_g} is a subformula of ϕ_{FM_g} if $\phi'_{FM_g} \in Sub(\phi_{FM_g})$.

It should be noted that the existing slicing algorithms cannot be applied to slice the trace links. Basically, the existing algorithms consist in eliminating the unselected variables in a propositional formula. But, as we need the bidirectional relationship of a selected variable (*i.e.*, a *vp* or variant) to another unselected variable (*i.e.*, a feature), we have to apply clause selection instead of variable elimination. This is the main reason why we propose this new slicing algorithm. Consequently, except for slicing trace links, slicing the FM itself can be done by previously proposed algorithms, which have shown good scalability on larger FMs [Acher et al., 2011]. Another reason is that our algorithm is based on selecting which features or *vp*-s with variants should remain in a sliced formula, instead of which of them should be eliminated. We expect that the selected set of variables to be eliminated from a formula of FM or MTVM can be larger than by only selecting which variables should remain in the sliced formula.

6.2.3 Substitution

After the slicing steps, the formal consistency rule given in (6.1) becomes:

$$\phi'_{FM} \wedge \phi'_{TL} \equiv \phi_{TVM_x} \wedge \phi'_{TL} \tag{6.3}$$

In essence, this consistency rule is applicable based on the substitution theorem [Kleene, 2002] for propositional formulas. *"The substitution theorem says in plain English that parts may be replaced by equivalent parts"* [Van Dalen, 2004], which correspond with the substitution of features with the traced *vp*-s or variants $(f \leftrightarrow vp)$. Therefore, checking the consistency between ϕ'_{FM} and ϕ_{TVM_x} by using the (6.3) is equivalent to checking the satisfiability and the interpretations of the following formula (*cf.* Figure 6.3):

$$\phi'_{FM} \wedge \phi'_{TL} \wedge \phi_{TVM_x} \tag{6.4}$$

Thus, we apply the substitution directly:

$$\phi'_{FM(f)} \wedge (f \leftrightarrow vp)' \wedge \phi_{TVM_x(vp)} \tag{6.5}$$

Concretely, for our Graph PL example, the formula for checking the consistency of $\phi_{TVM_{search}}$ becomes:

$$\begin{aligned} \phi'_{FM_g} \wedge \boldsymbol{\phi'_{TL_g}} \wedge \phi_{TVM_{search}} = g \wedge (t \leftrightarrow g) \wedge (w \leftrightarrow g) \wedge \\ (s \rightarrow g) \wedge (s \leftrightarrow (dfs \vee bfs)) \wedge (\neg dfs \vee \neg bfs) \wedge \\ \boldsymbol{(g \leftrightarrow vp_{root}) \wedge (s \leftrightarrow vp_s) \wedge (dfs \leftrightarrow v_{dfs}) \wedge (bfs \leftrightarrow v_{bfs})} \wedge \\ vp_{root} \wedge (vp_s \rightarrow vp_{root}) \wedge (vp_s \leftrightarrow (v_{dfs} \vee v_{bfs})) \wedge (\neg v_{dfs} \vee \neg v_{bfs}) \end{aligned} \tag{Ex. 6.8}$$

In boldface are shown the clauses, or bidirectional trace links, that make possible the substitution.

6.2.4 Assertion

By just checking if the resulting formula after slicing in (6.4) is satisfiable is insufficient to determine whether ϕ'_{FM} and ϕ_{TVM_x} are consistent or not. As previously stated, they are consistent when they have the same configurations. But, instead of comparing their configurations after they are generated, as in our example in Table 6.1, we propose to achieve this comparison while the formula in (6.4) is calculated.

Consistency detection. The trace links $(f \leftrightarrow vp)'$ in (6.5) indicate that (6.4) will generate only those interpretations (*i.e.*, configurations) which are similar between ϕ'_{FM} and ϕ_{TVM_x}. When ϕ'_{FM} is consistent with ϕ_{TVM_x} then they will have the same models with $(\phi'_{FM} \wedge \phi'_{TL} \wedge \phi_{TVM_x})$, by applying the substitution $(f \leftrightarrow vp)'$

in any side of the formula. As a result, we can simplify checking to only comparing their number of configurations. More exactly, we assert whether

$$[\![\phi'_{FM} \wedge \phi'_{TL} \wedge \phi_{TVM_x}]\!]_{size} = [\![\phi'_{FM}]\!]_{size} = [\![\phi_{TVM_x}]\!]_{size} \tag{6.6}$$

When this assertion is false then ϕ_{TVM_x} and ϕ'_{FM} are inconsistent. Let us illustrate how this works in our example.

Let us check the consistency of $\phi_{TVM_{search}}$ (*Ex. 6.3*) against the slice ϕ'_{FM_g} (*Ex. 6.7*). Figure 6.4 shows the sets of configurations for $[\![\phi'_{FM_g}]\!]$, $[\![\phi_{TVM_{search}}]\!]$, and $[\![\phi'_{FM_g} \wedge \phi'_{TL_g} \wedge \phi_{TVM_{search}}]\!]$. In this case, $\phi_{TVM_{search}}$ is consistent against the ϕ'_{FM_g} as the assertion (from (6.6)) is true (*i.e.*, each of the formulas have three configurations).

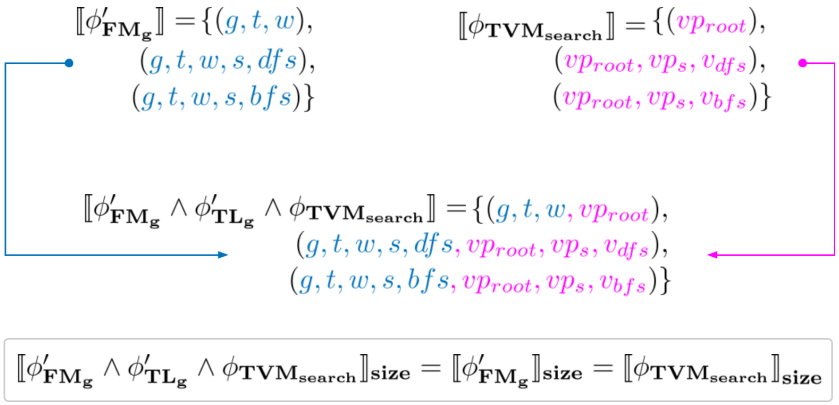


Figure 6.4: Consistency checking by asserting the number of configurations

As one can see, the bidirectional trace links ensure to generate from (6.4), respectively (*Ex. 6.8*), only those configurations that are similar between the ϕ_{TVM_x} and ϕ'_{FM}. Therefore, whenever the sliced formulas of ϕ'_{FM} and ϕ_{TVM_x} have the same number of configurations (*i.e.*, the same size of the set of configurations), then they are consistent. Moreover, under the Assumption 1 for trace links, it is not possible for ϕ'_{FM} with ϕ_{TVM_x} to have the same number of configurations and to be inconsistent.

Inconsistency detection.

As another example, let us suppose that the vp in $\phi_{TVM_{search}}$ is implemented as an Or logical relation between its variants (instead of their alternative logical relation), see Figure 6.5. According to the translation rules to the propositional logic (*cf.* Table 2.1), then

$$\begin{aligned}\phi_{TVM_{search}} &= vp_{root} \wedge \\ (vp_s \rightarrow vp_{root}) &\wedge (vp_s \leftrightarrow (v_{dfs} \vee v_{bfs}))\end{aligned} \tag{Ex. 6.9}$$

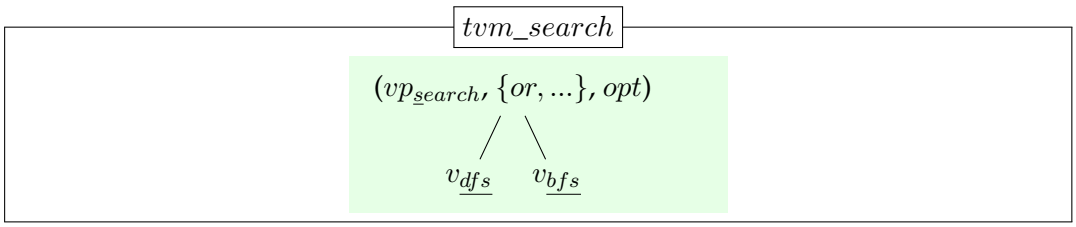


Figure 6.5: Implementation of vp_{search}, from Figure 4.4, as an Or logical relation between its variants.

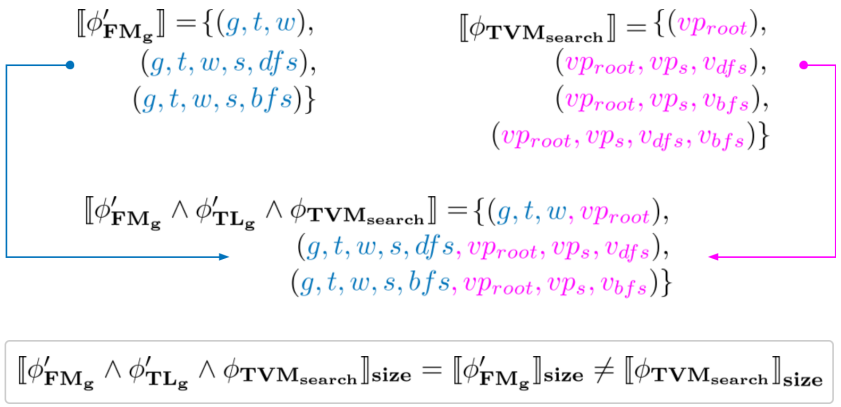


Figure 6.6: Example of inconsistency detection

Figure 6.6 shows the assertion step between $[\![\phi'_{FM_g}]\!]$ (from (*Ex. 6.7*)), the new formula $[\![\phi_{TVM_{search}}]\!]$ (from (*Ex. 6.9*)), and $[\![\phi'_{FM_g} \wedge \phi'_{TL_g} \wedge \phi_{TVM_{search}}]\!]$ (from (*Ex. 6.8*)). In this case, ϕ'_{FM_g} and $\phi_{TVM_{search}}$ are inconsistent as they have different sets of configurations, with 3 and 4 number of configurations, respectively. By comparing the configurations themselves in Figure 6.6, we see that for any feature configurations of ϕ'_{FM_g} we cannot obtain the software product as is the last configuration of *vp*-s with variants in $\phi_{TVM_{search}}$.

For checking the next TVMs, for instance the $\phi_{TVM_{basic}}$ as in Figure 4.4 , we repeat the same steps in our method except the initial checking for ϕ_{FM}, which is unnecessary. Instead of checking a single TVM, we can select for checking a set of TVMs until a total checking, in case that the whole specified variability is implemented. In this way, the form of variability consistency checking that we defined can be performed as soon as it is addressed, thus meeting the Challenge C3 on *early* consistency checking.

6.2.5 Handling 1 – M trace links

So far we considered that the mapping between features in the FM to the *vp*-s with variants in MTVM, respectively between their slices, is 1 – to – 1. When their

mapping is 1 – to – m, the assertion step in (6.6) becomes insufficient.

To illustrate the issues related to this mapping, let us suppose that the vp_{search} in $\phi_{TVM_{search}}$ (*cf.* Figure 4.4) has an implemented technical *vp*, $tvp_{\underline{n}one}$, that we have documented as in Figure 6.7 (see the five types of *vp*-s in Table 4.2, Section 4.2.1). Its functionality consists in coloring the graph with a green, $v_{gr\underline{e}en}$, or blue, $v_{\underline{b}lue}$, color instead of performing a search. This technical *vp* is not specified at the specification level (*i.e.*, at the FM in Figure 6.1), but it is implemented as an alternative variant with v_{dfs} and v_{bfs}. The propositional formula for this new TVM is (*cf.* Table 2.1):

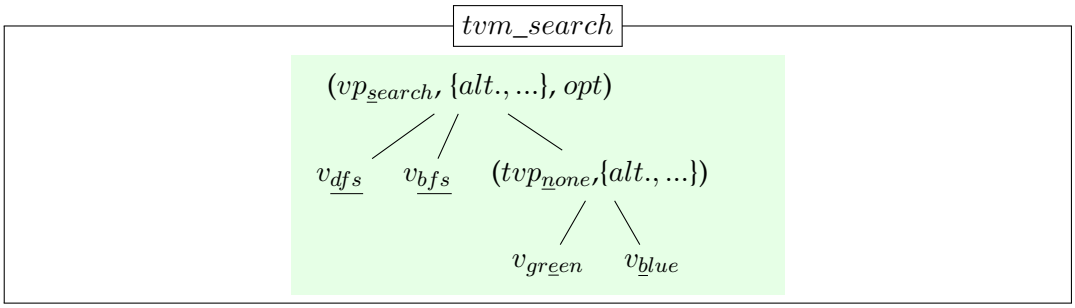


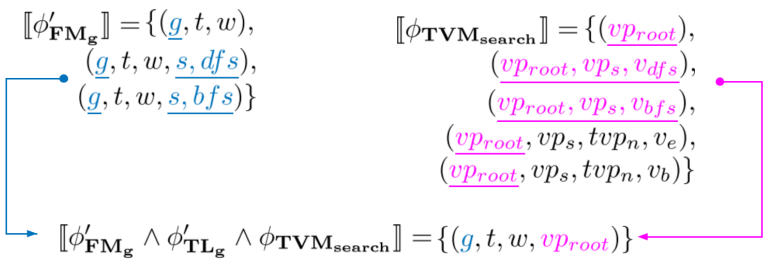
Figure 6.7: The `tvm_search` with a technical *vp* (*cf.* Figure 4.4 and Figure 6.1)

$$\begin{aligned}\phi_{TVM_{search}} = vp_{root} \wedge (vp_s \rightarrow vp_{root}) \wedge \\ (vp_s \leftrightarrow (v_{dfs} \vee v_{bfs} \vee tvp_n)) \wedge (\neg v_{dfs} \vee \neg v_{bfs}) \wedge \\ (\neg v_{dfs} \vee \neg tvp_n) \wedge (\neg v_{bfs} \vee \neg tvp_n) \wedge \\ (tvp_n \leftrightarrow (v_e \vee v_b)) \wedge (\neg v_e \vee \neg v_b)\end{aligned} \quad (Ex.\ 6.10)$$

Tracing the technical *vp*-s. It is common to consider that all *vp*-s (*e.g.*, ordinary, technical, nested) and their variants that implement a feature should be traced to that feature in FM, as in Equation (5.4) (Chapter 5).

Concretely, the technical variation point tvp_n with variants v_e and v_b, respectively, (*cf.* Figure 6.7) is traced to the feature `Search` in Figure 6.1, as this technical *vp* implements this feature. In this case, the slice of trace links $\phi'_{TL_g} \in Sub(\phi_{TL_g})$ corresponds to:

$$\begin{aligned}\phi'_{TL_g} = (g \leftrightarrow vp_{root}) \wedge \\ \boldsymbol{(s \leftrightarrow vp_s)} \wedge (dfs \leftrightarrow v_{dfs}) \wedge (bfs \leftrightarrow v_{bfs}) \wedge \\ \boldsymbol{(s \leftrightarrow tvp_n)} \wedge \boldsymbol{(s \leftrightarrow v_e)} \wedge \boldsymbol{(s \leftrightarrow v_b)}\end{aligned} \quad (Ex.\ 6.11)$$



$$[\![\phi'_{\mathbf{FM_g}} \wedge \phi'_{\mathbf{TL_g}} \wedge \phi_{\mathbf{TVM_{search}}}]\!_{\mathbf{size}} \neq [\![\phi'_{\mathbf{FM_g}}]\!_{\mathbf{size}} \neq [\![\phi_{\mathbf{TVM_{search}}}]\!_{\mathbf{size}}$$

Figure 6.8: False inconsistency detection for 1 – to – m links

For this new TVM (*cf.* (*Ex. 6.10*)) and its 1 – to – m trace links (see the links in boldface in (*Ex. 6.11*)), the assertion step is given in Figure 6.8.
Although the implementation of v_{dfs} and v_{bfs} are consistent to features `DFS` and `BFS` in Figure 6.1 (*i.e.*, both of them are in an alternative logical relation), the assertion step in our method shows that they are inconsistent.

The reason for such inconsistency is that, when we trace the technical *vp* to the same parent feature, we dismiss the logical relation between its variants, that is, the alternative relation between v_e and v_b in Figure 6.7 is lost and not considered.

In Figure 6.8, by comparing the underlined parts between $[\![\phi'_{FM_g}]\!]$ and $[\![\phi_{TVM_{search}}]\!]$, we noticed that the intersection between each feature configuration and *vp*-s with variants configuration is the set of configurations that shows their consistency.

This example reveals that, if for each configuration $c_f \in [\![\phi'_{FM}]\!]$ and $c_{vp} \in [\![\phi_{TVM_x}]\!]$ their intersection $c_f \cap c_{vp}$ is not empty then ϕ'_{FM} and ϕ_{TVM_x} are consistent. But, instead of comparing the configurations (*e.g.*, in Table 6.1 and Figure 6.8) we need a scalable solution at the formula level.

Slicing about technical *vp*-s. Among the possible solutions, to check the consistency of variability for 1–to–m trace links, the option to not trace the technical *vp*-s (*i.e.*, the software variability of core-code assets that is unspecified at the domain level) cannot be considered. Not tracing these technical *vp*-s, or roughly removing any of the *vp*-s with variants, may violate the dependencies of the other *vp*-s and variants, which have a direct mapping. Moreover, tracing all types of *vp*-s is important during the usage of trace links for product derivation, or simply to address the specified variability.

Therefore, a feasible solution is first to trace these technical *vp*-s and then to remove them by slicing the TVM_x for checking consistency. For such reason, it is easy to recognize when a *vp* is technical from the documentation of variability (*cf.* Section 4.2.1).

For example, in Figure 6.7 the technical variation point $tvp_{\underline{n}one}$ is preceded by a *'t'* letter.

We can remove this *vp* from the consideration during the consistency checking through slicing it, and still preserving the logical relations for the other *vp*-s and their dependencies.

For example, by using the approach of the existential quantifcation of a variable to slice a formula [Acher et al., 2011], we can remove the technical *vp* with its variants $\{tvp_n, v_e, v_b\}$ in (*Ex. 6.10*). The resulting slice, $\phi'_{TVM_{search}}$, will be the formula with 1–to–1 mapping, as in (*Ex. 6.3*).

In this way we bring back the 1 – to – 1 mapping and our consistency checking method is fully applicable. In this way, we addressed the Challenge C1 for checking the consistency of variabilies between the specification and implementation levels when their mapping is 1–to–m.

In our method we do not include checking for dead or false-optional features, respectively *vp*-s with variants, between both levels. To check for them would require a complete implementation of the specified features, and a total consistency checking of variabilities.

6.3 Related Work

Consistency checking of variability models and implementation is a topic of prime importance since the emergence of SPL engineering [Santos et al., 2015]. In the following we discuss works related to our approach according to different aspects.

Metzger et al. [2007] are among the first that propose to check consistency between the variability at specification and implementation levels. In their approach, the specified variability is modeled in an orthogonal variability model (OVM) [Pohl et al., 2005] in terms of *vp*-s with variants while the implemented variability is represented in an FM in terms of features. Although not explicitly, the mapping between *vp*-s with variants and features is 1 – to – 1. Basically, they check when cross-tree relations of features in FM or *vp*-s with variants in OVM may cause inconsistencies between each other. Unlike this approach, we advocate that the implemented variability is better captured by a forest-like structure, instead of a hierarchical tree structure that can be modeled by an FM. Further, in our approach the *vp*-s with variants are not merely abstractions, but they are consistent tags (*i.e.*, keep their corresponding association) to the existing variability in core-code assets that they represent.

Le et al. [2013] propose to check consistency of variability between features in an FM and preprocessor directives in C, their mapping being 1 – to – 1 and variability being implemented by only a single technique. Differently from us, they check the consistency of variability at once, considering that all features are implemented. Another major difference is that they propose to extract the variability

information from the core-code assets (*i.e.*, the preprocessor directives) and to rebuild the feature model which variability should be checked against the existing FM. In some points, the approach by Tartler et al. [2012] is similar to the one of Le et al. [2013]. The main difference is that they check variability consistency in a specific software system – the Linux kernel. Similarly, they check the total consistency of variability and identify the dead or false optional features modeled in KConfig language and their respective implementation as preprocessor directives. From our part, we target any SPL that use traditional techniques for addressing the variability in core-code assets.

Vierhauser et al. [2010] propose a tooled approach for checking the consistency between a variability model at specification level and other realization models, including the core-code assets. Unlike us, they define several consistency checking rules that are more about checking whether the variable units are addressed, and not if their logical relations are consistent with different models across the abstraction levels. As a result, they do not check whether the right mechanism or technique is used to realize the variability. As for their checking at code level, code artifacts are transformed into model elements and then checked. Another difference is that they propose an incremental checking, that is, whenever a developer makes a change it will be checked for consistency. Similarly, we propose to check the variability as earlier as it is implemented, but not after every single change. We do not check if a feature is simply addressed, but if the logical relation between a set of features are consistent with their implementation.

6.4 Summary

In this chapter, we proposed a method for checking the consistency of variabilities between the specification and implementation levels, as early as possible during the development process of an SPL. We handle the case when variability is implemented by different variability implementation techniques (*i.e.*, is *imperfectly modular*) and *vp*-s with variants can have an 1–to–1 and 1 – to – m mapping to the specified domain features in an FM.

Our consistency checking method is based on propositional logic. In particular, we achieve an early detection of inconsistencies during the development process by selecting just some of the modeled variability in terms of *vp*-s with variants in one or several TVMs within an SPL. Further, the method itself is based on three main steps, slice-substitute-assert. Whereas, for slicing the propositional formulas of FM and trace links we propose a new slicing algorithm based on clause selection. Thus, the method shows the usage of our designed TVMs and trace links, given in Chapters 4 and 5, respectively.

In the following Chapter 8, we present the implementation of this consistency checking method and its applicability in several case studies.

Part III:

Implementation and Validation

CHAPTER 7

Framework Implementation and Applications

Aiming at a complete solution to the challenges identified in this thesis, we implemented the proposed framework for capturing (*cf.* Section 4.1), modeling (*cf.* Section 4.2), and tracing (*cf.* Chapter 5) what we define as the imperfectly modular variability (*cf.* Section 4.1.1) at the implementation level. With such tooled support, we aim to show the implementation of our framework and its applicability in concrete PL domains. Specifically, we expect to be able to model their implemented variability in a fragmented way and to trace it, despite that several variability implementation techniques are used and that the variability in core-code assets is not shaped in terms of features. Therefore, in this chapter are presented the resulting tool support and its applications in four different case studies. These applications show a successful usage of our framework in small and medium-sized SPLs.

7.1 Technical Foundations

An Internal DSL in Scala. We implemented the proposed framework, for modeling (*cf.* Chapter 4) and tracing (*cf.* Chapter 5) the variability of core-code assets, as a textual *Domain Specific Language* (DSL) (*cf.* Section 2.5) in Scala. One of the design goals of the Scala language was the support for DSL development [Odersky et al., 2010], thus we choose it as it is well-known for its first-class support for designing expressive DSLs [Ghosh, 2010, Ch. 6]. The interoperability between Java and Scala enables one to use our DSL in almost all Java-based systems. Our framework could be implemented as another internal DSL also in other general-purpose languages, or an external DSL. For such DSL design, some of the available and handy guidance can be found in Fowler and Parsons [2010]; Ghosh [2010]; Voelter et al. [2013]; Voelter and Visser [2011].

Patterns supported by the DSL. In this part, we give the language constructs that are defined in our DSL. They are presented here as patterns for guiding the usage of the DSL, and the form of support that the DSL offers for modeling and tracing the implemented variability.

Specifically, Listings 7.1 to 7.3 show the patterns to use the DSL for defining (*i.e.*, capturing) *vp*-s with variants, modeling the implemented variability, and establishing the trace links with features at the specification level. The internal DSL just has to be imported into the current scope (*e.g.*, package, file, class) where variability needs to be separately documented into a technical variability model (TVM).

The TVM consists of two parts:

1. The captured abstractions of *vp*-s (< **vp_name** >) with variants (< **variant_name** >) that are associated with the variable elements in core-code assets (< **tag to the variabe asset** >), given in Listing 7.1; and,

2. The documented variability in a `fragment` construct, given in Listing 7.2.

To capture variability, our DSL relies on the reflection capabilities in Scala. This helps to keep the relationship and strong consistency between the concepts of *vp*-s with variants with their concrete implementation in core-code assets. The DSL supports the five types of *vp*-s we defined (*cf.* Table 4.2, Section 4.2.1), which are shown as < **vp_types** > in Listing 7.1. Thus, an ordinary *vp* is defined as `VP`, an optional *vp* is defined as `opt_VP`, a technical *vp* is defined as `tech_VP`, an unimplemented *vp* is defined as an ordinary *vp* (*i.e.*, as `VP`) but without variants, whereas a nested *vp* is defined as `nested_VP`. In addition, < **vp_name** > and < **variant_name** > are of *String* types.

```
import dsl._
import scala.reflect.runtime.universe._

<vp_name>       := <vp_type>(<tag to the variable asset>)
<variant_name>  := Variant(<tag to the variable asset>)

<tag to the variable asset>  := asset(<class | method | field>)
<vp_type>                    := <VP | opt_VP | tech_VP |nested_VP>
<vp_name> | <variant_name>   := String
```

Listing 7.1: Pattern for defining a variation point and variant

```
import dsl.fragment._

fragment(<class | file | package name>) {
        <vp_name> is <logical relation> with_variants
        (<list of variant_name(s)>) use
        <technique> with_binding
        <binding time> and_evolution <evolution>
}

<logical relation>  := <MND | OPT | ALT | MUL>
<technique>         := <Inheritance | Overriding | ...>
<binding time>      := <Compilation | Start up | Runtime | ...>
<evolution>         := < Open | Close>
```

Listing 7.2: Pattern for modeling some implemented variability in terms of the defined variation points with variants in Listing 7.1

In Listing 7.2, the < **logical relation** >, < **technique** >, < **binding time** > and < **evolution** > take their specified values that are given in our framework, in Section 4.1.2 (Page 49). Technically, we defined their possible values as extensions of

their respective sealed classes [1] in Scala, thus well defining their possible values. And, the DSL is made expressive by using the implicit conversions in Scala.

```
import dsl.traces._

traces {
        % for all variation points with variants %
        <vp_name> implements <feature_nameA>
        <variant_name> implements <feature_nameB>
}
```

Listing 7.3: Pattern for defining trace links

The implemented DSL uses another construct, `traces`, for establishing the trace links between the specified features and the documented variability in TVMs (*cf.* Listing 7.3). These traces are internally kept in a map structure. The DSL supports the creation of 1–to–m trace links, where a single feature, $<$ **feature_nameX** $>$, can be mapped to several *vp*-s, $<$ **vp_nameX1** $>$, $<$ **vp_nameX2** $>$, or variants. This is made possible by defining the map structure as $Map[feature, vp]$ and $Map[feature, variant]$, where feature is the key whereas the *vp* or variant is its value; and, a key in the map structure can have several values.

Further, the `traces` with the `implements` construct of the DSL (*cf.* Listing 7.3) are used to establish the trace links between the *vp* with variants and features, for example, between features in Figure 4.1 (Page 46) and the modeled variability for Listings 4.1 to 4.4 (Page 47), as shown in Listing 7.5.

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

Listing 7.4: An excerpt of variability capturing and documentation in JavaGeom

As illustrative example,

[1] A sealed class cannot have any new subclasses added except the ones that are defined in the same file [Odersky et al., 2010].

Listing 7.4 shows the usage of the DSL for documenting the implemented variabilities, from Listings 4.1 to 4.4 (Page 47), in a TVM. First, the DSL support is imported (line 1), then the abstractions for the implemented variability are created (*i.e.*, the *vp* with variants) by using the reflection in Scala (lines 4-7). Finally, the implemented variability is modeled in a `fragment` construct (lines 10-16) in terms of the defined *vp* with three variants. In addition, in Listing 7.4 is defined an ordinary *vp*, by using the `VP` construct of the DSL in line 4, with its three variants, by using the `Variant` construct of the DSL (lines 5-7).

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

Listing 7.5: An excerpt of variability traceability in JavaGeom

A particular case is the definition of a nested *vp*. In such case, a variant should be redefined as a nested *vp*, as illustrated in Listing 7.6.

```
1 import dsl._
2 import scala.reflect.runtime.universe._
3
4 val v_Line2D = Variant(asset(/*...*/))
5 val vp_newLine2D: nested_VP = v_Line2D.toNestedVP
```

Listing 7.6: Defining a nested variation point (a variant becoming a nested *vp*, as defined in Section 4.2.1)

Availability. The prototype implementation of our DSL is publicly available at https://github.com/ternava/variability-cchecking. It also contains the three case studies (Graph PL, Arcade Game Maker PL, and Microwave Oven PL) that we implemented in Scala, with examples on capturing, modeling, and tracing their variability. The feature model (FM) for each of these case studies is given at https://github.com/ternava/Expressions_SPL/wiki/Feature-Models. Whereas, a further detailed description of the DSL and its documentation is given at https://ternava.github.io/.

7.2 Applications

Our aim is to be able to use this DSL in concrete PL domains. First, for modeling the implemented variability in a fragmented way (*i.e.*, in technical variability models) when a combined set of traditional techniques are used. In such case, the

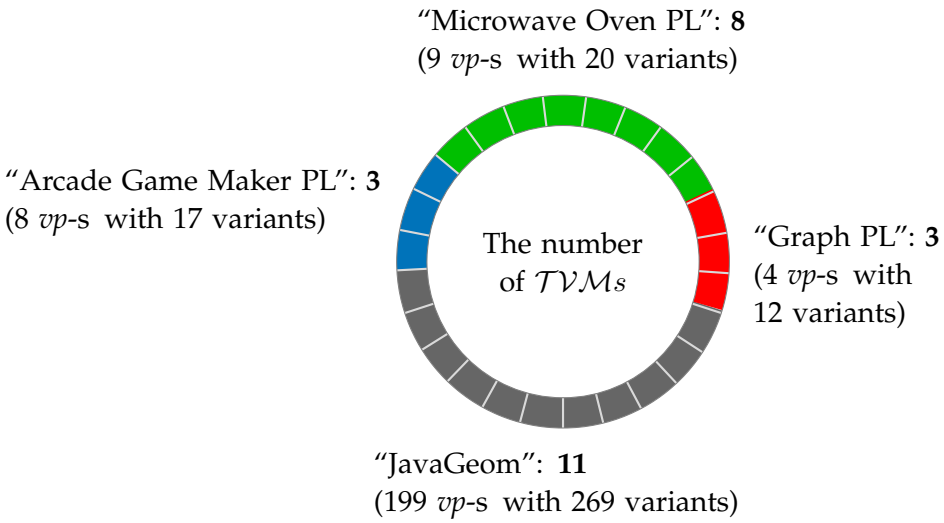


Figure 7.1: A comparison of TVMs (the colored boxes), and their *vp*-s with variants, between the four case studies

vp-s with variants are diverse and not a by-product of any of these techniques. Then, we want to show its applicability to trace these *vp*-s with variants with the features at the domain level, while their mapping is 1–to–1 and 1–to–m. Thus, we expect to be able to apply this DSL in different PLs, for modeling and tracing their variability of core-code assets.

Therefore, we used the DSL for modeling and tracing the implemented variability of four case studies: Graph PL [Lopez-Herrejon and Batory, 2001], Arcade Game Maker PL [AGM, 2009], Microwave Oven PL [Gomaa, 2005], and JavaGeom [Legland, 2017]. The domains of the first three case studies are quite well understood and used by the SPL community. The fourth case study, JavaGeom, is an open source library implemented in Java, which is a feature-rich system for creating geometric shapes. A summary of these case studies is given in Appendix B.

Initially, we applied the DSL to the open source library, JavaGeom. It has 142 Java source files (excluding tests) and 35,456 lines of code. Considering that the implementation of 3D geometric shapes is under implementation, we analysed only 122 files of 2D geometric shapes that constitute 92% of the code. As JavaGeom does not have an FM with the realized features, we created it from their textual description and by analysing the source code. It consists of 110 concrete features, which are rather close to the implementation.

Thus, we used the DSL for modeling the whole implemented variability for the JavaGeom library. It resulted in 11 TVMs, all of them being at the package level, as we intended to minimize the dependencies of *vp*-s across different TVMs.

As for the three other case studies, we implemented them in Scala and applied the DSL just after we were implementing their variability. In this way, we documented their variability in TVMs. The number of TVMs and their *vp*-s with

Table 7.1: Number of features (*F*-s), implemented features (Impl. *F*-s), *TVM*-s, and the captured *vp*-s with variants in each case study

Case Study	***F*-s**	**Impl. *F*-s**	***TVM*-s**	***vp*-s with variants**
Graph PL	19	16	3	4 with 12
Arcade Game Maker PL	27	26	3	8 with 17
Microwave Oven PL	26	23	8	9 with 20
JavaGeom	110	110	11	199 with 269

variants in each case study is given in Figure 7.1. For example, in Graph PL we modeled the variability in 3 TVMs, which contain 4 *vp*-s with 12 variants. While the TVMs in JavaGeom are at the package level, we created the TVMs in the three other case studies at the file level. A combination of TVMS at the package, file, or class level would have been also possible. Further, we save internally all modeled variability of TVMs directly into a map structure known as $power\_set$, which represents the whole variability of an SPL, that is, the MTVM. Thus, the MTVM in each case study is created automatically from their TVMs. With our DSL we were able to capture different variability implementation techniques and to model the variability of core-code assets in a fragmented way.

In Table 7.1 we show the number of features that are implemented and their respective number of *vp*-s with variants that we documented in each case study. This also shows that their mapping is not ideal (*i.e.*, 1–to–1 as in Listing 7.5).

For example, in Listing 7.7, the technical *vp* `vp_Temperature` with its three variants, `v_High`, `v_Medium`, and `v_Low`, are traced to the same feature `f_HeatingElement` in line 6 (*i.e.*, an 1–to–m trace link).

```
1  import dsl.traces._
2  import tvm_temperature._
3  import specification_level.moven_spl
4
5  traces {
6      vp_HeatingElement    implements f_HeatingElement
7      v_OneLevelHeating    implements f_OneLevelHeating
8      v_MultiLevelHeating  implements f_MultiLevelHeating
9      vp_Temperature       implements f_HeatingElement
10     v_High               implements f_HeatingElement
11     v_Medium             implements f_HeatingElement
12     v_Low                implements f_HeatingElement    //...
13 }
```

Listing 7.7: An example of multiple trace links, in Microwave Oven PL

Although it is not mandatory, in each case study we kept all the trace links in one file. To do so, we just need to import the specific TVM to trace its *vp*-s with

Table 7.2: Number of *vp*-s and their implementation techniques in each case study

Techniques of vp-s	**Graph PL**	**Arcade Game Maker PL**	**Microwave Oven PL**	**JavaGeom**
Inheritance	-	6	11	60
Overloading	-	-	-	121
Generic Type	-	-	-	18
Parameters	2	-	-	-
Template Pattern	1	-	-	-
Strategy Pattern	2	-	-	-
Include Package	2	-	-	-
Total	7	6	11	199

variants (*e.g.*, the `tvm_temperature._` in line 2, from Listing 7.7) and the specified features in the FM (*e.g.*, the `specification_level.moven_spl` in line 3).

In order to understand what variability implementation techniques are used in these case studies where we modeled the variability of their core-code assets, and how many of them and in what combination (*cf.* Challenge A2), we analysed the documented variability in four case studies regarding the used variability implementation techniques (*i.e.*, binding time, granularity etc.), and types of *vp*-s. The results are shown in Tables 7.2 and 7.3, which highlight some of the capabilities of our DSL, respectively of our framework, that we applied.

For example, in JavaGeom we documented up to three used techniques for implementing variability (*cf.* Table 7.2). We omit the technique of overriding as it is used constantly to realize the specialization in the inheritance hierarchy. More specifically, the inheritance is applied at class level (30%) with runtime binding. The overriding is used at method level (37%) or constructor level (24%) with compile time binding. Finally, generic types represent the variability of a parameter at class or method level (9%) with static binding. We marked all 199 *vp*-s with `Open` as their default evolution, and the majority of them has an alternative relation among the variants. These *vp*-s contains around 269 variants, excluding that 35 *vp*-s have 79 nested *vp*-s. As ordinary *vp*-s we documented 71, while the others were unimplemented or technical *vp*-s (*cf.* Table 7.3). Similarly, we captured and documented the variability for the other three case studies. In Table 7.2 we emphasized the "technique" of *Include Package* for the Graph PL because it is not actually an implementation technique (*cf.* Table 3.5). It is a way for making optional a core asset (*i.e.*, an optional *vp*).

These applications show the successful usage of our DSL for capturing and modeling different traditional techniques in core-code assets (*i.e.*, for modeling the imperfecly modular variability at the implementation level). For this reason, we successfully used the five types of *vp*-s with variants and traced them with the features at the domain level, under their 1–to–1 and 1–to–m mappings.

Table 7.3: The documented number of *vp*-s' types in each case study

Types of vp-s	**Graph PL**	**Arcade Game Maker PL**	**Microwave Oven PL**	**JavaGeom**
Ordinary	5	6	7	71
Unimplemented	-	-	-	7
Technical	-	-	-	121
Optional	2	-	4	-
Nested	-	-	-	79

7.2.1 Design Evaluation of TVMs

In this section, we give some of the advantages of TVMs, in the way that they are supported by the DSL. Specifically, we study how they make possible to keep the strong consistency between the *vp*-s with variants abstractions and their concrete realizations in core-code assets while keeping both the variability and development levels still separated (*cf.* Challenge A3), as well as modeling the implemented variability in a fragmented way (*cf.* Challenge A4). In addition, the usage of TVMs with the trace links, which is also supported by our DSL, is multiple.

Easy management. Our DSL supports the documentation of variability for an SPL in several TVMs. The *vp*-s with variants of a TVM can be part of their core-code assets, are not amalgamated with the functionality in code, but still they are close to the core-code assets. Thus, a TVM can be part of the same class or file, as in Figure 7.2, or kept separately in a class, a file (see Figures 7.3b and 7.3c), or a package level (see Figures 7.3a and 7.3c) on its own. A concrete example is given in Figure 7.3d. Moreover, if required, the whole variability can be modeled in a single TVM, in which case it becomes equivalent with the MTVM. On the other hand, we establish and keep all trace links for an SPL in one single file at the project level. By documenting the implemented variability in such flexible TVMs, we believe that the management of variability is facilitated, thus meeting Challenge A4.

Strong consistency with core-code assets. By using the reflection mechanisms in Scala, we keep the strong consistency of *vp*-s with variants, as concepts (*e.g.*, in Listing 7.4 lines 4-7), to their concrete realization in core-code assets (*i.e.*, Listings 4.1 to 4.4, Page 47). Moreover, by capturing the implementation technique, such as the logical relations, binding time, evolution properties of *vp*-s with variants (*e.g.*, in Listing 7.4 lines 12-15), we also keep a form of consistency between the modeled variability in TVMs and its implementation in core-code assets (*i.e.*, Listings 4.1 to 4.4, Page 47). Thus, the TVMs and the core-code assets are kept close but separated enough (*e.g.*, Figure 7.2), thus meeting the Challenge A3 on separating the variability and development dimensions. In this way, our DSL supports the management of the implemented variability from the earlier steps of capturing,

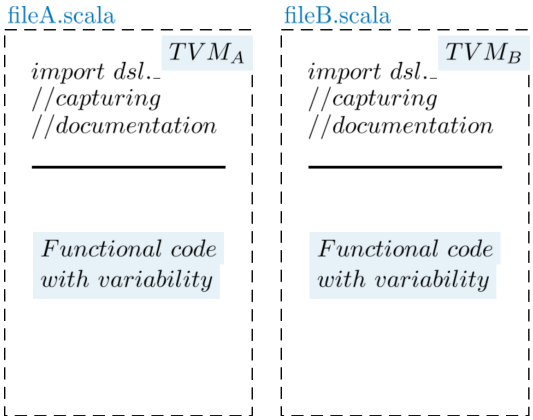


Figure 7.2: The TVMs closer but still separated with the functional code in files

documenting and then tracing it.

Multiple usage. Further, the usage of TVMs and trace links are multiple (*cf.* Challenge B). Capturing and documenting the different types of *vp*-s and their implementation techniques, as in Listing 7.4 and Tables 7.2 and 7.3, becomes important especially during the usage of technical variability models and trace links. Specifically, when trace links are used to resolve variability, knowing the binding time of variants in a *vp* is necessary. Then, the logical relation between variants in a *vp* is needed to check the consistency between the variability at the specification and implementation levels. In addition, knowing whether a *vp* is open or unimplemented is needed during variability evolution. All these indicate that considering the variability implementation techniques, as in our DSL, during the capturing and documentation of variability have a major impact in establishing and using the trace links, that is, in variability management. Especially when the variability is implemented by several traditional techniques in combination (*cf.* Challenge A2).

Main TVM to SPL ratio. For capturing and documenting the variability in TVMs we use our textual DSL. Obviously, it contribues to the overall lines of functional code in core-code assets. Therefore, we use the Main TVM to SPL ratio as a means for evaluating the design of TVMs, with our DSL, by measuring the percentage of lines of code in all TVMs toward the lines of code for realizing the functionality in core-code assets within an SPL.

For this reason, we measured this ratio in one of our PL case studies, for the Microwave Oven PL. This SPL has 914 lines of functional code, and 8 TVMs with 116 lines of code. Therefore, the Main TVM to SPL ratio, in this case, is around 11% and indicates that around 89% of the overall lines of code are used for implementing the functionality of Microwave Oven PL and the rest for documenting its variability. But, depending on what is calculated as a line of code in our DSL for the TVMs, this ratio may be smaller. For example, the 4 lines of code in Listing 7.4

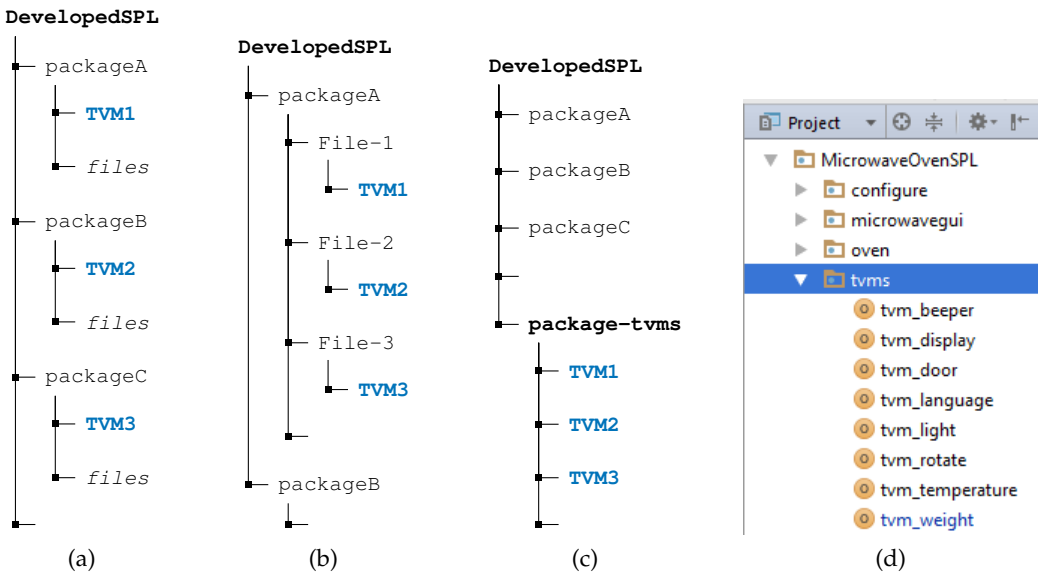


Figure 7.3: (a) The separated TVMs in package level; (b) The separated TVMs in file level; (c) The separated TVMs in package, file, or class level; (d) For example, the TVMs in separate files for Microwave Oven PL

in 12-15 lines represent a single line of code in our DSL, but we have separated in 4 lines for the sake of readability.

Overall, for a *vp* or variant to be captured is needed always a single line of code (*e.g.*, Listing 7.4, line 4, or 5), and also another line of code (or two lines, for the nested variation points) to model a *vp* with its variants within the `fragment` module (*e.g.*, Listing 7.4, lines 12-15). Moreover, we observed that this ratio is proportional to the number of *vp*-s with variants and the number of TVMs, but it does not depend on the lines of functional code within an SPL.

The DSL itself is a small language, (2) with few defined language constructs, (2) it uses the reflection in Scala, and (3) as an internal DSL relies on the host Scala language, also meaning that it can easily be extended. These imply that the DSL itself should also have a light learning curve.

7.2.2 Limitations

A limitation of our DSL, but not of the framework itself, is that we could not apply it for tracing the variability at the finest granularity level (*e.g.*, at the expression level), as using reflection for tagging the variability is not possible at that level. A solution is to refactor this finest-grained variability, although using reflection is not mandatory. Variability can also be captured by using a form of annotations, such as by Heymans et al. Heymans et al. [2012]. But, we used the reflection of a language instead of developing another tool for annotating and parsing the code for extracting "automatically" the variability information.

In our proposition, putting annotations is still a manual process, and they stay

amalgamated with code. Moreover, we capture the implementation technique, such as the logical relation, binding time, and evolution of *vp*-s with variants, instead of simply identifying and annotating the places in core-code assets where some variability happens. In the other hand, reflection helps us to maintain the strong consistency between the *vp*-s with variants and their implementation in core assets, and we keep the variability information separated from the core-code assets such that the TVMs can be part of the core asset (not amalgamated with its code) or separated into their own files. However, there is 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 7.4, lines 12-15).

Another difficulty was to understand which is exactly a *vp* and variant in core code assets, during the modeling of variability later as in the JavaGeom case study. It was sometimes hard to distinguish whether an inheritance hierarchy implements some product line variability (*i.e.*, which differentiates products in an SPL) or some internal functionality that should not be modeled or traced. Although, this is not a limitation that is particular to our framework or its DSL implementation. We emphasize it here because it became obvious to us while we were using the DSL in our case studies. We faced most of the difficulties in deciding which technique (*e.g.*, inheritance hierarchy) should be classified as a relevant *vp* with variants to be modeled in TVMs and traced, or whether we should document all of them. In the last case, the reverse engineering approaches, such as finding the feature locations during the migration of some software products in an SPL [Ziadi et al., 2014], could help, but within the context of our approach, to find and reverse engineer the *vp*-s with variants by capturing the implementation techniques. This is similar to approaches that consider that all C preprocessor directives in core-code assets are used for implementing some PL variability, whereas the others disagree [Zhang et al., 2013]. Specifically, Zhang et al. [2013] 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*". However, this was not an issue during a proactive SPL engineering process, when the variability is implemented in a methodological way and *vp*-s with variants are documented by the DSL while they are implemented, as in the three other case studies that we implemented.

This difficulty may restrict the usage of our framework in proactive SPL development. Still, we believe its usage could increase the awareness of developers for the variability traceability during the engineering phases, so that they aim at modularizing the variability whenever possible.

7.3 Summary

As a concrete solution of our given framework for capturing, modeling, and tracing the variability, we implemented an internal DSL in Scala language. To show

the applicability of this DSL and of the framework itself for variability management in realistic SPLs, we applied the DSL in three small SPLs and a medium SPL. The DSL has language constructs for capturing the five types of *vp*-s with variants when several traditional techniques are used in core-code assets, and two other constructs, for modeling their variability in a fragmented way - in TVMs, and tracing their *vp*-s with variants with the specified features in an FM.

Further, we present a design evaluation of TVMs, mostly regarding the way that they meet the given challenges A2 to A4 and B. Using this DSL, the addressed variability can be documented while it is implemented and in the same place (*e.g.*, package or file) by using the same development tool.

In addition, the DSL can be extended to raise the given limitations, also the future work for extending the framework itself, as presented in Section 9.3.

CHAPTER 8

An Implementation of the Consistency Checking Method

This chapter shares material with the following paper:

Tërnava, Xh. and Collet, P. (2017a). Early consistency checking between specification and implementation variabilities. In *Preceedings of the 21st International Systems and Software Product Line Conference - Volume A*, SPLC '17, pages 29–38. ACM

In this chapter, we present our implementation of the consistency checking method, given in Chapter 6, for checking the consistency of variabilities between the specification and implementation levels. It supports the early consistency checking, that is, during the development process and by selecting only some of the addressed variabilities at the implementation level. This is made possible by using our DSL for modeling the implemented variability of core-code assets in TVMs and tracing it to features at the specification level (*cf.* Chapter 7). We also show the applicability of this implemented method in four case studies.

8.1 Implementation

Figure 8.1 depicts our toolchain for checking the consistency of variability. We implemented the slice-substitute-assert method, given in Section 6.2, using the Scala language. In this way, we could use the TVMs and trace links directly, as our DSL is also defined in Scala (*cf.* Section 7.1).

Toolchain. Initially, we take as input the feature model (FM) in the propositional logic. We use FeatureIDE [Thüm et al., 2014] for converting the whole FM to propositional formula, ϕ_{FM}, although it is not integrated into our implementation. Then, the trace links (TLs) that are established by using our DSL, we convert them to the propositional logic by using the classic conversion rules given in Table 2.1. Further, the propositional logic formulas for FM and TLs are converted to conjunctive normal forme (CNF), by using the conversion algorithms by Gupta [2014]. In this way, with our tool, we encode their CNF formulas in DIMACS CNF format (*i.e.*, in $.dimacs$ or in $.cnf$ files) and we perform the initial checking (*cf.* Section 6.2.1), by using SAT techniques with the SAT4j solver [Berre, 2013; Le Berre and Parrain, 2010]. Specifically, we check whether each of the formulas is valid, free of false-optional and dead variables [Benavides et al., 2010]. A more complete set of the implemented operators for analysing the FMs can be found in several existing

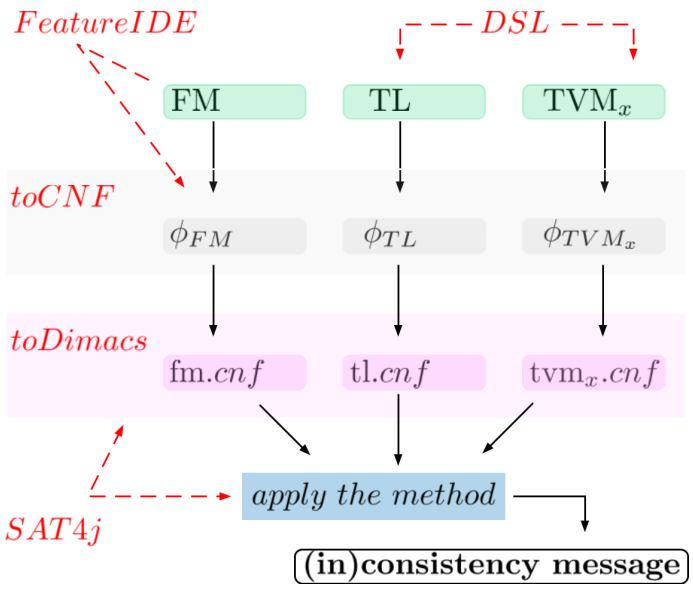


Figure 8.1: Prototype toolchain for consistency checking

implementations [Chechik et al., 2015; Henard, 2012; Holthusen et al., 2015]. Depending on the kind of analysis operation that has to be performed, a combination of different solvers is suggested [Benavides et al., 2007, 2006], but in our current implementation, we used only a SAT solver.

If the FM and TLs are valid we proceed to the conversion of a specific technical variability model, TVM$_x$, or several TVMs to CNF, which are selected to be checked. Then, we encode them to DIMACS CNF format and check their validity as well. What proceeds is the application of our method for checking the consistency of a selected TVM$_x$ against the given variability in FM. The slicing steps are implemented by using our proposed clause selection algorithm on CNF formulas (*cf.* Section 6.2.2), but other slicing algorithms could be used. The clause selection algorithm is sufficient for the considered SPLs (*cf.* Section 8.2). Moreover, we use SAT4j for generating the number of models for a propositional formula (*cf.* Section 6.2.3) and then conclude whether the TVM$_x$ is consistent with the FM or not (*cf.* Section 6.2.4).

In our implementation, the specified variability in terms of features in an FM is used as a reference model against which is checked the consistency of the implemented variability in terms of *vp*-s with variants in TVMs.

Availability. The prototype implementation of our consistency checking method is publicly available at `https://github.com/ternava/variability-cchecking`. It is configured for checking the consistency of the implemented variability for Microwave Oven PL, which can help as a guide for using our method in other case studies, too. Whereas, a further detailed description of our implemented method and its documentation is given at

https://ternava.github.io/.

8.2 Applications

We applied and evaluated our method by using it for checking the consistency of variability in the four case studies, which are given in Appendix B. In the previous chapter, we applied our DSL in these case studies for modeling and tracing their implemented variability (*cf.* Section 7.2).

We evaluated our method regarding the logical relations of features, respectively *vp*-s with variants that can be checked (*i.e.*, mandatory and optional *vp*-s, as well as alternative and Or logical relations between their variants). Specifically, we evaluated whether the method is capable to detect the inconsistencies successfully in all these possible logical relations. For example, in the Microwave Oven PL and JavaGeom we experimented with the technical and nested *vp*-s of any type, whereas in Graph PL with *vp*-s that are implemented as a refactoring form of the specified variability (*e.g.*, when some related optional features are implemented as a *vp* with variants in an Or logical relation). We expect that our method will be functional and applicable in many SPL settings, whereas we do not specifically discuss its scalability. Specifically, it should be able to detect successfully the inconsistencies that may happen on any kind of the logical relations between features in an FM and *vp*-s with variants in TVMs at the implementation level.

The implemented variability is documented in each case study, using our DSL [1]. In all case studies, a TVM has at least one *vp* with its variants. In the first three SPLs, we did not implement all features. First, it is common that some variability is implemented later. Secondly, the SPLs can be evolved during the time with new features. Despite that some features are unaddressed, we could check the consistency for only that part of the implemented variability showing the partial checking capability of our method.

8.2.1 Evaluation Process

We performed the evaluation process in two stages. First, we checked the consistency of variability by selecting TVMs one by one, and then we were selecting a subset of them. In each stage, we analysed first the *vp*-s with variants that have 1 – to – 1 mapping to features in the FM (*i.e.*, single trace links) and then those with 1 – to – m mapping (*i.e.*, multiple trace links).

For example, Figure 8.2 shows a screenshot of a setup for checking the consistency of one or several TVMs for Microwave Oven PL. Specifically, the FM and trace links are checked about their validity by selecting a specific PL. Currently, we first configure the solution with an SPL during the step ① in Figure 8.2, as is configured with the Microwave Oven PL. Then, during the step ②, the TVMs are imported in the current scope. Finally, one or several TVMs can be selected to

[1] In the previous chapter, Table 7.1 (Page 96) showns their respective number of TVMs and their *vp*-s with variants.

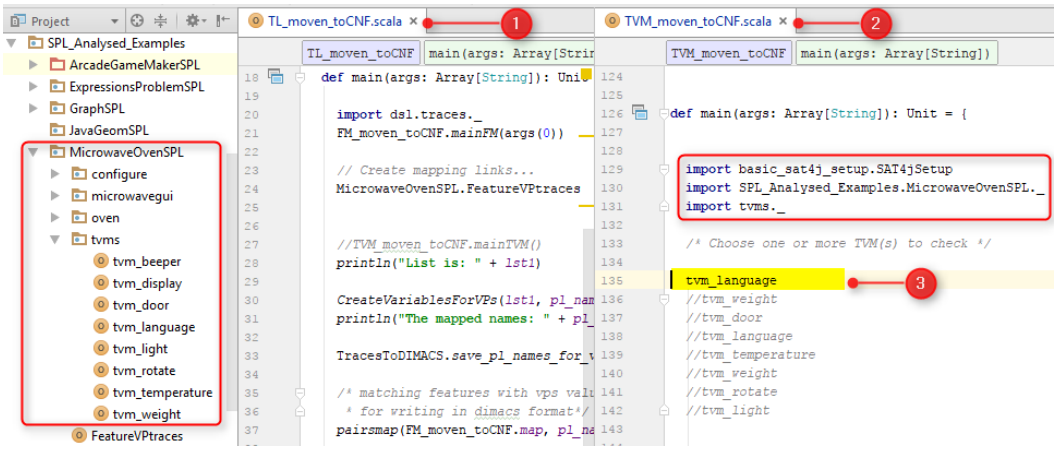


Figure 8.2: A setup for checking the consistency of one or several TVMs for Microwave Oven PL

check their consistency at once, as shown in step ③. Also, all TVMs can be selected for checking in case that the whole specified variability is addressed at the implementation level.

Inconsistencies due to trace links. The first inconsistencies that can be reported are about trace links (step ①, Figure 8.2). An inconsistency will be reported when a *vp* or variant from the selected TVM to be checked (1) does not have a mapping to the FM, and (2) when it has a wrong mapping. In the first case, when we select such TVM, or a subset of them, if the trace links are not well-established (*i.e.*, the Assumption 1 is not met, Section 6.1 on Page 74) an inconsistency is reported early, meaning that the consistency of the selected variability cannot continue and checked without establishing the trace links for all *vp*-s with variants in the selected TVMs.

Whereas, in the second case, the inconsistency will be reported at the end, showing an inconsistency between the checked TVMs with the sliced FM. This inconsistency is likely to require more effort to be detected and understood, as it is caused by wrong trace links. This happens when a variant is mapped to a feature that it does not implement. Whereas, such inconsistency goes undetected when a *vp* with variants is mapped mistakenly to some features that have completely similar logical relation between them in FM. Overall, the trace links can be established only between the TVMs and the FM. Otherwise, our DSL reports an error at compile time. For instance, it is not possible to trace a feature to another feature in the FM instead of tracing it to a *vp* or variant at the implementation level.

If trace links are well-established, the FM and TVM are checked about their self-consistencies, then the consistency checking between them is performed using our prototype implementation.

Table 8.1: Checking a TVM with one *vp* and its variants

a) The selected logical relations of variants in a vp to test. Where their mapping to features in FM is 1–to–1, and with the same logical relation

Features	*vp*-s with variants	Result
alternative	alternative variants	*consistent*
or	or variants	*consistent*
optional	optional *vp*	*consistent*
optional	or variants	*consistent*

b) The selected logical relations of variants in a vp to test. Where their mapping to features in FM is 1–to–1, and with the different logical relation

Features	*vp*-s with variants	Result
alternative	or variants	*inconsistent*
mandatory	optional *vp*	*inconsistent*
or	alternative variants	*inconsistent*

Table 8.2: Checking more than one TVM with one or more *vp*-s with variants

The selected logical relations of variants in a vp to test. Where, their mapping to features in FM is 1–to–1, and when only one of the vp-s has different logical relation

Features	*vp*-s with variants	Result
(alternative, or)	alternative variants	*inconsistent*
or	(or, alternative) variants	*inconsistent*
(optional, or)	mandatory *vp*, or variants	*inconsistent*

Single trace links. We selected several TVMs in each case study, with the aim to assess our implementation with different logical relations of *vp*-s with variants, shown in Table 8.1a. When the variability was implemented in the right way and the mapping of *vp*-s with variants to features in FM was 1–to–1, then we could check successfully their consistency. For example, in Figure 8.3 is shown the consistency checking for an alternative *vp*, `Language` [2], with five variants, `English`, `French`, `Italian`, `Spanish`, and `German` (step ①). This *vp* implements a compound feature with five alternative features in the FM [3]. Therefore, by selecting the TVM with this *vp* and its variants (step ②) one obtains a consistency message (step ③). Similarly, when an optional feature is implemented as an optional *vp*, their checking reports a success. Consistency was also reported when a *vp* was

[2]In this case study we did not used the prefix `vp_` for a *vp* and `v_` for a variant. For instance, instead of `vp_Language` we name it simply as `Language`.

[3]See its FM in https://github.com/ternava/expressions_spl/wiki/feature-models

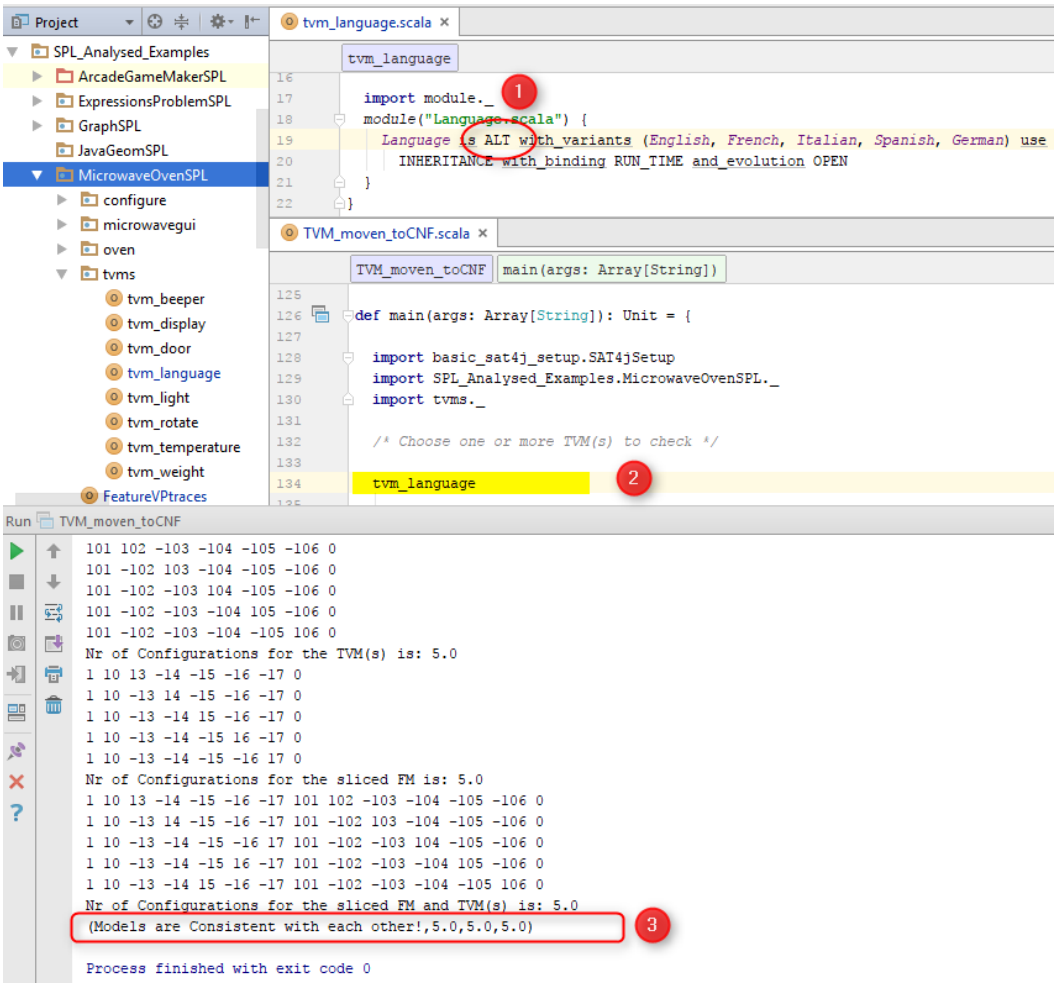


Figure 8.3: The detected consistency for a TVM in the Microwave Oven PL

a refactoring form of a group of features in FM, for example, when a group of Or related features is implemented as optional variants in a *vp*, see the last entry in Table 8.1a.

Similarly, we selected the TVMs that have different types of *vp*-s, but in cases where the variability is not properly implemented (see Table 8.1b), for example, when an alternative group of features is implemented as an Or group of variants in a *vp*, or a mandatory feature is implemented as an optional *vp*. Concretely, if the *vp*, `Language`, in Figure 8.3 ① is implemented by a technique that offers an Or logical relation between its variants (instead of an alternative logical relation) then an inconsistency is reported as in Figure 8.4. Their inconsistency is reported as a difference between their numbers of configurations.

Then, within a single case study, we selected different subsets of TVMs to check the consistency of their variability against the features in FM (*e.g.*, see the commented TVMs in Figure 8.2 in step ③). These TVMs have one or more *vp*-s with variants. When each individual TVM in the subset was consistent, we could check

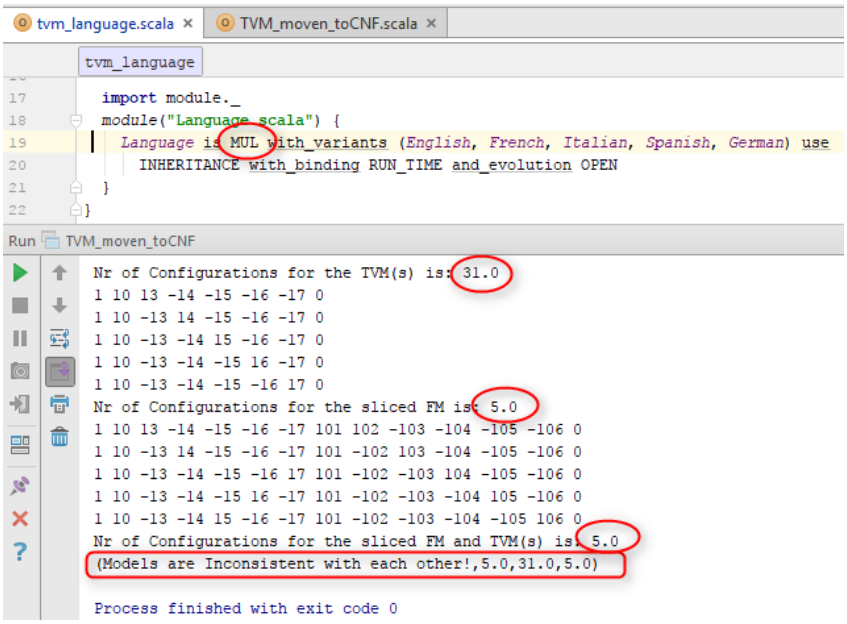


Figure 8.4: The detected inconsistency for a TVM in the Microwave Oven PL

that together they are consistent too. Further, we combined a consistent TVM, or a group of them, with at least one inconsistent TVM and we checked their variability together. Table 8.2 shows the types of logical relations that we selected for *vp*-s in TVMs. As was expected, we got an inconsistency, but now it is part of the all selected TVMs, or their respective slice FM.

All kinds of inconsistencies are reported as a difference between the number of *vp*-s with variants configurations in TVMs and the number of features configurations in the slice FM. Their configurations are also made available for any further comparison.

Multiple trace links. Our current implementation supports tracing a feature to more than one *vp* or variant, as illustrated by the example in Listing 7.7 (Page 96). This means that some *vp*-s are technical or nested, and do not have a direct mapping at the specification level. They are only modeled at the implementation level, and we must consider them during the consistency checking of variability.

We applied our consistency checking method when a technical *vp* and its variants are traced directly to the same specified feature in the FM (*cf.* Figure 6.7, Page 84). Under these 1 – to – m trace links, we did similar evaluations with different logical relations of *vp*-s with variants, as in the case of 1–to–1 trace links (*cf.* Tables 8.1 and 8.2). However, an inconsistency was always reported even when the other *vp*-s where consistent, as we were expecting. This indicates that our implementation for checking the consistency of variabilities when the mapping is 1–to–1 is insufficient for the 1–to–m mapping.

Therefore, for handling the 1–to–m mapping, in Section 6.2.5 (Page 85) we pro-

Table 8.3: Times for checking the consistency of TVMs compared to self-consistency checking of the FM, in the Microwave Oven SPL

Where FM′ is the sliced feature model, TVM the technical variability model, and CC the consistency checking of TVM against the FM′.

	tvm_{lang}			tvm_{temp}			tvm_{weight}		
Prop. Formula	FM′	TVM	CC	FM′	TVM	CC	FM′	TVM	CC
# Configurations	6	6	6	5	5	5	2	2	2
Checking time (average in *ms*)		28.10			27.81			30.26	

	$tvm_{\{lang,temp\}}$			$tvm_{\{lang,temp,weight\}}$		
Prop. Formula	FM′	TVM	CC	FM′	TVM	CC
# Configurations	30	30	30	60	60	60
Checking time (average in *ms*)		33.98			44.82	

posed to slice the selected TVM first, by removing the technical *vp*-s and keeping only *vp*-s with variants that have a single mapping to features in FM. This requires only an extension of our implemented method by one more slicing step, regarding the technical *vp*-s, between the $slice_0$ and $slice_1$ in Figure 6.3 (Page 79).

8.2.2 Execution Time

By selecting a single or a subset of TVMs at a time, the consistency checking is made possible early in the development process, as soon as the specified variability is implemented and documented. 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 Microwave Oven PL, as it has most of the *vp*'s types. The resulting data are given in Table 8.3. The measurements are performed on a PC Intel(R) Core(TM) i5-4300U CPU, 64-bit, with 2.50 GHz and 4 GB RAM, on Windows 10.

We selected for checking three TVMs from the Microwave Oven SPL, tvm_{lang}, tvm_{temp}, tvm_{weight}. They have 7, 7, and 3 *vp*-s with variants, respectively. We then checked them together. Execution times in each case was instantaneous, and are reported in Table 8.3. For example, for tvm_{lang} we measured the execution time for the sliced FM', the TVM itself, and the time for checking their consistency took around 28.10 millisecons. The execution time for checking the validity of the whole FM with 26 features and 720 configurations took around 130 milliseconds (it is not shown in the Table). As we expected, the execution times for checking the consistency of partial variability compared to only the validity of the FM is smaller. This indicates that the number of features and *vp*-s with variants that can

be selected and checked tend to be smaller, but we did not measure how much and for what number of *vp*-s with variants in TVMs that are checked.

The slicing time of trace links and FM contribute also to the consistency-checking time of a TVM. We did not measure it, as it is a property of the slicing algorithm and requires extra validation on its own.

8.2.3 Limitations

First, our DSL cannot be currently used to model features in an FM at specification level (*cf.* Figure 8.1). For this reason, we used another tool, FeatureIDE, to model the variability in an FM and to convert it to a propositional formula, but it is not integrated with our implementation. We thus plan to extend our DSL or to integrate it with another textual DSL that models the variability specifically at the specification level, such as FAMILIAR [Acher et al., 2013].

When features or *vp*-s with variants have dependencies, then slicing their propositional formulas with our proposed algorithm in Section 6.2.2, based on clause selection, is not tested. However, we tested it for slicing the bidirectional trace links, as the existing algorithms could not be used. On the other hand, after the initial checking, our consistency checking method is based on counting the number of features or *vp*-s with variants configurations. In a performance comparison of some logical solvers, such as SAT [4], BDD [5], and CSP [6], Benavides et al. [2010, 2007, 2006] suggest that for counting the number of configurations the BDD solver is more efficient than the SAT solver. However, we used the SAT solver for the initial checking and, in particular, because our slicing algorithm requires the propositional formula to be in a CNF, whereas the BDD solver takes a propositional formula not necessarily in CNF. Therefore, when other slicing algorithms are used, the method can use a BDD solver.

In the current implementation, slicing of TVMs according to technical *vp*-s for checking the consistency of variability for 1–to–m mapping (*cf.* Section 6.2.5) is not completelly implemented at the time of writting.

8.3 Summary

Our prototype implementation is a realization of our consistency checking method, given in Figure 6.3 (Section 6.2), and represents a usage of our framework for capturing, modeling and tracing the imperfectly modular variability at the implementation level. Further, it also shows the applicability of our implemented DSL, as a concrete solution for modeling and tracing the variability of core-code assets.

Towards evaluating the implemented method, we applied it in three small and a medium case studies. We evaluated it regarding three main aspects: (1) when

[4] Boolean SAT-isfiability problem
[5] Binary Decision Diagrams
[6] Constraint Satisfaction Problem

the mapping between features and *vp*-s with variants was 1–to–1 and 1–to–m, (2) by selecting one TVM at a time and then a combination of them within a single SPL, and also (3) by considering those TVMs that contain *vp*-s with variants in different logical relations, such as mandatory, optional, or, and alternative ones. In all cases the method was reporting the expected consistency or inconsistency, even considering the limitation on the current implementation for supporting the 1–to–m mapping. We also do a small comparison regarding the execution time for checking different numbers of the selected TVMs within an SPL, which result to be smaller than checking the satisfiability of only the whole FM.

In addition, we give some limitations of the currently implemented prototype of our consistency checking method. They are additional to the given limitations and future work for extending the framework and the method itself, as presented in Section 9.3

CHAPTER 9

Conclusion

In this chapter, we step back on the contributions of this dissertation. First, in Section 9.1 we discuss how the challenges determined in Section 1.2.1 have been addressed. Then, in Section 9.2 we discuss some limitations of this work. Finally, in Section 9.3 we discuss some possible perspectives.

9.1 On Challenges

A software product line (SPL) engineering approach is based on a methodological development and reuse of core software assets for a set of related software applications within a domain. At first, in a proactive approach, the domain of software products is scoped and the common and variable features between products are modeled in a variability model, usually in a feature model (FM). Then, these features are realized in different core assets, such as in code level, known as core-code assets. For realizing the variability in core-code assets, there are available different variability implementation techniques. When traditional techniques are used, such as inheritance, design patterns, the code basically is not shaped in terms of features at the domain level, thus hampering the handling of variability at the code level. In this dissertation, we addressed the three groups of the given challenges in Section 1.2.1, for modeling, tracing, and checking the consistency of the implemented variability in core-code assets during an SPL engineering. In this thesis, we addressed the following challenges:

Challenge A. Variability modeling at the implementation level

- A1. *Understanding the diversity of variability implementation techniques*
- A2. *Capturing and modeling the implemented variability when a combined set of traditional variability implementation techniques is used*
- A3. *Separating the development dimension and variability dimension while maintaining their corresponding association (a.k.a. consistency)*
- A4. *Modeling the implemented variability in a fragmented way*

Challenge B. Variability traceability

- B. *Supporting the variability traceability between the specification and implementation levels*

Challenge C. Consistency checking of variability

C1. *Checking the consistency of variability between the specification and implementation levels*

C2. *Checking the consistency of variability when a combined set of traditional variability implementation techniques is used*

C3. *Achieving an early detection of variability inconsistencies*

In Chapter 3 we studied the elements of diversity for the majority of variability implementation techniques and provided a unified set of comparison criteria. We organized these criteria in a catalog that covers an enriched set of techniques compared to a same set of criteria; thus, meeting Challenge A1. We built this catalog (1) by using existing catalogs, taxonomies, frameworks, and studies that we identify in the literature about the variability implementation techniques and also (2) by exercising with different techniques in three PL case studies implemented with the Scala language.

We believe the given catalog can be seen as a complete set of variability implementation techniques with respect to the major approaches in the state of the art. However, some very specific language techniques and those less popular in the industry or research works have not been included. Consequently, choosing a technique is not limited to selecting only among these techniques, but we expect the proposed catalog to guide SPL architects and developers in the majority of cases.

Although other criteria for evaluating techniques can also be added, we believe they would not change the current evaluated results to the existing criteria. To the best of our knowledge, the criteria themselves do not have a correlation within a technique, but they are used to make trade-offs between techniques. These trade-offs are not currently shown in our catalog, for instance, the one between binding time and open/closed properties [Bosch et al., 2001].

While features at the specification level are merely concepts, at the implementation level a combination of traditional variability implementation techniques are actually used in realistic SPLs, thus leading to a form of an imperfect modularity of the variability at the implementation level that should be handled. In Chapters 4 and 5 we provided a framework for capturing, modeling, and tracing this imperfectly modular variability of core-code assets, thereby meeting the Challenges A2 and B. The framework supports the capture of variability in core-code assets in terms of variation points (*vp*-s) with variants and its modeling in technical variability models (TVMs), thus meeting the Challenge A4 for modeling the variability in a fragmented way.

We did not use the decision models for variability modeling (*cf.* Section 2.1) because decisions are commonly used only as a means for resolving the *vp*-s with variants. Whereas, the *vp*-s with variants in TVMs and with their characteristic properties capture the variability implementation techniques and can be used for different purposes, such as for resolving the variability, evolving, checking the consistency, understanding the addressed variability in core-code assets. By keeping

separated the concepts of *vp*-s and variants with their realization in core-code assets, we achieved the objective of the Challenge A3 for separating the variability dimension from the development dimension in core-code assets, while maintaining their consistency. Moreover, we have met the Challenges A2 to B during the variability traceability of core-code assets from its early steps, by capturing and modeling the implemented variability in several TVMs, then we traced it up to variability at the specification level (*i.e.*, features in the FM), under their n–to–m mapping.

We expect that our framework can be realized and used in realistic SPLs where commonly a combined set of variability implementation techniques are used at the code level. Actually, we showed a form of its feasibility and applicability by implementing it as an internal domain specific language (DSL) in Scala, which realization is applied in four different SPL case studies (*cf.* Chapter 7).

Toward using the framework and meeting Challenge C, in Chapter 6 we proposed a method for checking automatically the consistency between specification and implementation variabilities expressed in the propositional logic. Concretely, the method supports the checking of whether the specified variability in terms of features at the domain level and their respective implementation in core-code assets represent the same number of software products. We handled the case when the imperfectly modular variability in core-code assets is modeled in TVMs, and *vp*-s with variants have an 1–to–1 and/or 1–to–m mapping to the specified domain features in the FM.

In this context, the consistency of variability could be checked successfully whenever features and *vp*-s with variants had a single mapping, whereas it became harder when their mapping was 1 – to – m, that is, with multiple trace links. The difficulty to check consistency under multiple links lies in the fact that it is ambiguous how to trace the technical *vp*-s with variants that do not have a single mapping to some features in the FM. We evaluated our method by tracing them at the same feature as their parent variation point. In this case, despite that some *vp*-s with variants that have single links were consistent, an inconsistency was reported. However, since multiple mapping links between these levels of variabilities can be reduced to single mapping links, it is possible to use the same proposed method for detecting the variability inconsistencies under the 1–to–m mapping.

In this way, by checking the consistency between the specification and implementation variabilities, while considering their multiple trace links, we met Challenges C1 and C2. Also, instead of doing a complete checking, we select a single TVM or a subset of them to detect their inconsistencies as early as possible during the development process, that is, immediately after some variability is addressed; thus, meeting Challenge C3.

Furthermore, in Chapter 8 we provided a prototype implementation of our consistency checking method. Then, we applied it in four SPL case studies to check the consistency of variabilities between specification and implementation levels as early as possible during the development process.

9.2 On the Scope and Limitations

Usage of the catalog. The existing catalogs, frameworks, and studies have evaluated their considered techniques by using different programming paradigms and languages, such as Java, C++, Smalltalk, and we used the Scala language. Therefore, the evaluation of techniques in our catalog is not uniform regarding the used paradigms or programming languages. Also, each variability implementation technique is evaluated in isolation, whereas in reality, a combined set of techniques seems to be necessary to meet the diverse requirements of variability and their dependencies in implementation. We consider that these two main issues should be taken into account during the usage of our given catalog.

Modeling of dependencies between the *vp*-s with variants. A weakness of our framework is that it does not include the modeling of dependencies between the *vp*-s with variants, which are different from the cross-tree constraints of features in the FM at the specification level. In addition, because we model the variability in several TVMs, then the *vp*-s with variants in one TVM can have dependencies to the *vp*-s with variants in another TVM. These dependencies between TVMs may also help us to define more exactly the proper number of TVMs that should be used within an SPL, whereas in our case studies we tried to minimize them.

The variability models in both abstraction levels are required. Our variability consistency checking method is based on checking the logical relations between features and *vp*-s with variants, thus checking whether the same number of the specified software products are also realized in core-code assets. Variability can also be checked with regards to binding times or evolution properties. Usually, these properties are modeled only at the realization level and checking for their consistency requires them to be specified at the specification level, too. Specifically, only in TVMs we documented explicitly these two properties of *vp*-s. In this context, there are two main limitations of our approach. First, the variability models in both abstraction levels are required. Then, for example, to check the variability regarding its binding time, it is required to be available or modeled the binding times for features and *vp*-s in each variability model. Besides, for an automatic checking, the binding time should be represented in the propositional formulas in some way.

Evaluating the proposed slicing algorithm. Another concern in our consistency checking method is about the slicing algorithm. Actually, we did not evaluate our proposed algorithm for slicing the propositional formulas (based on clause selection) when *vp*-s with variants have dependencies in core-code assets. This is because we used our framework, and it does not support the modeling of dependencies between the variability abstractions, yet. But, instead, some of the other existing slicing algorithms are mathematically proved that they do the right slicing for every kind of propositional formulas, which can also be used.

Alternative tool support for modeling the implemented variability. Instead of using our DSL, the implemented variability can be documented using other ways, such as a form of annotations [Heymans et al., 2012]. When annotations are used, they should not only annotate the place where a variable unit is implemented in core-code assets but also what is the logical relation between those units. This is related to our consistency rule. Specifically, we did not check only whether some variability is merely addressed. We checked whether the same software products that are specified can be derived from the core-code assets, within an SPL. Further, we supposed that the trace links are well-established. Otherwise, when trace links are the source of an inconsistency, we considered that it is not a variability inconsistency anymore. It is then an inconsistency regarding the addressing and mapping of variabilities, for instance, when we try to check the consistency of an unimplemented variability or the mapping is mistaken.

Considering that a single variability implementation technique is used. In our work, variability is implemented by using a combined set of traditional variability implementation techniques (*e.g.*, inheritance, design patterns). In these techniques *vp*-s are not explicit, such as by using preprocessors in C. If we take preprocessors, they also can offer all kinds of logical relations between variants in a *vp*. Although, they offer a single binding time for *vp*-s compared to the case when several traditional techniques are used. 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, and then applying our consistency checking method when only this implementation technique is used.

Moreover, in Chapters 7 and 8 we determined the limitations of our implementations, for the DSL and the prototype of our consistency checking method, respectively. They are particular limitations regarding only the current implementations. Specifically, we cannot use the DSL for modeling and tracing the variability at the finest granularity level. Also, there is a weak consistency between the used implementation language and the modeled properties of *vp*-s with variants. Further, using the DSL in a forward engineering approach is more straightforward than in a reverse engineering approach.

As for the limitations of the implemented consistency checking method, the current tool does not support the modeling of variability at the specification level (*i.e.*, features in a feature model). Besides, our slicing algorithm based on clause selection is not evaluated when *vp*-s with variants have dependencies and the consistency checking for 1–to–m mapping is not yet fully supported by our current implementation. Moreover, we could use in our method another logical solver, for example, a BDD solver that is suggested by Benavides et al. [2010, 2007] over the SAT solvers for counting the number of features or *vp* with variants configurations.

9.3 Future Work

Improving the catalog. Some ongoing work is currently tackling the evaluation and choice of variability implementation techniques in a more systematic way, by considering the *variable parts* in core-code assets as *centers* [1] in Alexander's meaning [Alexander, 2004]. We believe that the 15 fundamental properties for building beautiful and functional *centers* in the architecture can guide us also in evaluating and choosing different techniques in combination for realizing manageable *variable parts*, while realizing the functionality in core-code assets.

Extending the framework. Further, we plan to complement our framework for supporting the modeling of dependencies between *vp*-s with variants in core-code assets and among the TVMs. In the literature, there are already some frameworks about the dependencies of *vp*-s, such as from Bühne et al. [2003]; Sinnema et al. [2004a], which can be considered. By addressing these dependencies, we plan to extend our DSL to support the documentation of these dependencies, then to test our slicing algorithm in the consistency checking method for slicing the propositional formulas in the presence of *vp*-s with variants dependencies.

Resolving the detected inconsistencies. When an inconsistency is detected, it is supposed to be handled, that is, finding its location and resolving it. Finding its location is quite trivial as we select a single TVM or a set of them for checking their consistency against their sliced FM. When the location is known, the variability inconsistencies can be resolved by changing the implementation technique for the *vp*-s, changing the way how the variants are implemented or refactoring the specified features in the FM. In order to give help for resolving such inconsistencies at the implementation level, we plan to show the usage of our catalog for choosing the right variability implementation techniques.

Evaluating the scalability of our framework and consistency checking method. As we need both variability models, at the specification and implementation levels, we have not yet studied precisely the scalability of our framework and consistency checking method. However, the experimental results of our case studies (*cf.* Chapters 7 and 8) and the short execution time of TVMs are promising indicators. In addition, we plan to use existing slicing algorithms to slice the FM and compare their performances. Choosing the best slicing algorithm will improve the scalability of our implementation. Then, instead of using the SAT solver, we want to evaluate the consistency checking method when a BDD solver is used. Currently, we are tackling the implementation and analysis of the consistency checking method under 1 – to – m trace links when the technical *vp*-s are first removed by slicing the selected TVMs.

[1] Informally, centers are fundamental entities of a spatial structure. They mark something that we experience as memorable, remarkable, or draws our attention.

We expect these advances will complement our current framework and consistency checking method so that we can more largely evaluate their practicality and usefulness in order to obtain insights to guide SPL practitioners.

An upcoming usage of the framework. Moreover, the exploitation of our three step framework for product derivation is also envisaged. To do so, we are trying first to improve the current DSL for achieving a stronger consistency between the used variability implementation techniques in core-code assets and the characteristic properties of *vp*-s with variants, such as the binding time (*cf.* Listing 7.4, lines 12-15). Something similar with the consistency of *vp*-s with variants with core-code assets (*cf.* Listing 7.4, lines 4-7).

Actually, a concern about our framework is its use for an automatic resolution of *vp*-s with variants during a product derivation. Approaches that are based on using a form of annotations, actually annotate the places in core-code assets, such as some lines of code, a file, or a package, that represents a feature and has to be part of the final software product or not. For example, depending on whether an #if preprocessor directive in C is *true* or *false* then the whole enclosed code within that #if-#endif condition that represents a feature is included or erased in/from the final software product. This cannot happen with traditional techniques, such as inheritance, or overriding, for two main reasons. First, during a product derivation, when a feature from the specification level is not selected to be included in the final software product, it is not possible to remove an entire declaration, such as a package, a class, or method, from the core-code assets that is used to address that feature using the Java or Scala language. Secondly, in our framework, the *vp*-s with variants mark the places in code where the variability is concentrated (*e.g.*, a file, a class, a method) and we do not do a commitment which lines of code are exactly in or out a *vp* or variant concept (see *"An essential difference"* in Section 2.2.2 and Section 4.2.3). For such reasons, we plan to a semiautomatic product derivation approach.

Possible integration. Overall, while we are going to make these improvements, the framework itself can be used to complement several existing works. Our developed DSL can be integrated with another language for modeling the variability in terms of features in an FM, thus having a more inclusive support for managing the variability at the specification and implementation levels. In Section 8.2.3, we mentioned a possible integration with FAMILIAR textual DSL [Acher et al., 2013]. In such case, the developed analysis operators for feature models can be used also for analysing the technical variability models at the implementation level, such is integrated and used the FAMILIAR with the KCVL [2015].

Then, although FeatureIDE [Thüm et al., 2014] is meant to support only feature-oriented development [Apel et al., 2013], we believe that it can be extended, or a parallel and similar tool support in the existing IDEs is needed for managing the variability when a combined set of traditional variability implementation tech-

niques are used in core-code assets, which is the case in realistic SPL settings. Also, our catalog can be integrated in such tools and used to guide the developers in choosing the techniques in a methodological way. In this way, we aim at developing features of software products within an SPL as conceptual modules, although not perfectly modular, and not particularly as physically separated modules.

APPENDIX A

Definitions of Features and Variation Points with Variants

Feature's Definitions

Kang et al. [1990] – a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems

Jacobson et al. [1997] – a use case, part of a use case or a responsibility of a use case

Kang et al. [1998] – distinctively identifiable functional abstractions that must be implemented, tested, delivered, and maintained.

Zave [1999] – an increment of functionality, usually with a coherent purpose

Czarnecki and Eisenecker [2000] – anything users or client programs might want to control about a concept

Bosch [2000] – a logical unit of behavior that is specified by a set of functional and quality requirements

Czarnecki and Eisenecker [2000] – a property of a domain concept, which is relevant to some domain stakeholder and is used to discriminate between concept instances

Savolainen and Kuusela [2001] – a common language between many stakeholders. They communicate the high-level functional requirements from the marketing to the development

Riebisch [2003] – an aspect valuable to the customer

Batory et al. [2004] – a product characteristic that is used in distinguishing programs within a family of related programs

Gomaa [2005] – a functional requirement; a reusable product line requirement or characteristic. A requirement or characteristic that is provided by one or more members of the software product line

Berg et al. [2005] – product capabilities and characteristics that are important to the user

Pohl et al. [2005] – an abstract requirement. Features describe the functional as well as the quality characteristics of the system under consideration

Chen et al. [2005] – a product characteristic from user or customer views, which essentially consists of a cohesive set of individual requirements.

Batory [2006] – an elaboration or argumentation of an entity(s) that introduces a new service, capability or relationship.

Hotz et al. [2006] – a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems. An asset type that is used to model functional aspects of a product

Apel et al. [2007] – a structure that extends and modifies the structure of a given program in order to satisfy a stakeholder's requirement, to implement and encapsulate a design decision, and to offer a configuration option

Classen et al. [2008] – a triplet, f = (R, W, S), where R represents the requirements the feature satisfies, W the assumptions the feature takes about its environment and S its specification

Reiser [2008] – a characteristic or trait in the broadest sense that an individual product instance of a product line may or may not possess. A feature only describes what is variable, not how this variability is realized

John [2010] – a product requirement $R \subseteq D$ that is visible to a user of the product P (in the application domain D)

Variation Point's Definitions

Jacobson et al. [1997] – A variation point identifies one or more locations at which the variation will occur.

Czarnecki and Eisenecker [2000] – The nodes to which variable features are attached are referred to as variation points. More formally, a variation point is a feature (or concept) that has at least one direct variable subfeature (or feature).

Clauß and Jena [2001] – A variation point is modeled in the modeling element where the variation occurs because it is not an extra entity (in contrast to a class or component).

Van Gurp et al. [2001] – a variation point is defined as a delayed design decision

Becker [2003a] – The main concept that represents variability on the implementation level is the variation point. A variation point is a spot in a software asset where variation will occur, i.e. where a variability is realized, at least partially. Thus, a variation point can be considered as some kind of generic element in a software asset.

Sinnema et al. [2004b] – Variation points are places in a design or implementation that identify locations at which a choice can be made between zero or more variants.

Schmid and John [2004] – A variation point is "the «point» in a generic artifact where variation may occur.

Svahnberg et al. [2005] – Variation points are "places in the design or implementation that together provide the mechanisms necessary to make a feature variable.

Pohl et al. [2005] – A variation point is a representation of a variability subject within domain artefacts enriched by contextual information.

Bachmann and Clements [2005] – A variable part is the place in an asset that is allowed to vary." On contrary, "variation point is often used to describe variations in terms that refer to an asset's externally visible properties or functions rather than places in the asset's internal structure.

Hunt and McGregor [2006] – Variation point is the place in the product where we see the consequences of feature choices.

Rabiser et al. [2007] – A decision represents a user intervention needed for the selection of assets required for a concrete product during product derivation. Decisions are abstract representations of variation points in the asset model.

John et al. [2007] – A variation point describes the occurrence of variability in a development artifact.

Forster et al. [2008] – The locations at which a software artefact can be extended or configured for a particular context are so-called variation points.

Variant's Definitions

The way that a variation point is going to vary is expressed by its variants.

ISO/IEC 26550:2015 [2015] – one alternative that may be used to realize particular variation points (Note 1 to entry: One or more variants must correspond to each variation point. Each variant has to be associated with one or more variation points. Selection and binding of variants for a specific product determine the characteristics of the particular variability for the product.)

APPENDIX B

The Developed Case Studies

In the context of this dissertation, we study and developed several case studies, which we also use them for the illustration. In this appendix, we give some more details for each of these case studies. As a summary, four of them we have implemented and one of them is an open source library. In addition, we make the feature model and the source code available for all of these case studies: https://github.com/ternava/variability-cchecking.

B.1 Summary of Case Studies

Graph Product Line

Version:	March, 2017 (last change)
Features:	18 (16 developed)
vp-s with *variants*:	4 with 12
Developed by:	Xhevahire Tërnava
Developed as:	Software product line (forward approach)
Language:	Scala
Size:	323 lines of code, 3 files

Description. Graph product line is a family of graph applications, which is proposed as a case study for evaluating the product line methodologies [Lopez-Herrejon and Batory, 2001]. The domain of Graph PL consists of 2 mandatory features, 2 optional features, 3 groups of alternative features (each with 2 variants), and 1 group of 7 features in an Or logical relation. Also, it has 6 cross-tree constraints between these features. Table 7.2 shows the variability implementation techniques that we used to realize these features.

Arcade Game Maker Product Line

Version:	March, 2017 (last change)
Features:	28 (26 developed)
vp-s with *variants*:	8 with 17
Developed by:	Xhevahire Tërnava
Developed as:	Software product line (forward approach)
Language:	Scala
Size:	651 lines of code, 17 files

Description. Arcade Game Maker product line is a family of three arcade games: Brickles, Pong, and Bowling. It is introduced by the SEI [1] as a case study for experimenting and learning the development of software product lines [AGM, 2009]. Originally it has 19 features, with only a group of 3 alternative features, and 2 other optional features. We refined these in 27 features with 3 groups of alternative features (each contains from 2 variants), and 5 groups of features in an Or logical relation, and 1 optional feature. Both these feature models are given in https://github.com/ternava/expressions_spl/wiki/feature-models. Whereas, Table 7.2 shows the variability implementation techniques that we used for its development.

Microwave Oven Product Line

Version:	March, 2017 (last change)
Features:	26 (23 developed)
vp-s with *variants*:	9 with 20
Developed by:	Xhevahire Tërnava
Developed as:	Software product line (forward approach)
Language:	Scala
Size:	914 lines of code, 19 files

Description. Microwave Oven product line is one of the three case studies introduced by [Gomaa, 2005]. It has input buttons for selecting Cooking Time, Start, and Cancel, as well as a numeric keypad. It can display the cooking time left. Also, it has a microwave heating element, a door sensor, and a weight sensor. Cooking is possible only when the door is closed and when there is something in the oven. Its development is described step-by-step from requirements to detail design using the UML language. Even the feature modeling is given as a feature dependency diagram in UML. It has 7 mandatory features, 7 optional features, and 4 groups of features in an alternative logical relation. Table 7.2 shows the variability implementation techniques that we used for its development.

Expressions Product Line

Version:	March, 2017 (last change)
Features:	7 (7 developed)
vp-s with *variants*:	2 with 5
Developed by:	Xhevahire Tërnava
Developed as:	Software product line (forward approach)
Language:	Scala
Size (*version 1*):	73 lines of code, 1 file
Size (*version 2*):	34 lines of code, 1 file

[1] Software Engineering Institute: http://www.sei.cmu.edu/productlines/ppl/

Size (*version 3*):	103 lines of code, 1 file

Description. The Expressions product line is represented by Lopez-Herrejon et al. [2005], and it is actually the old "*Expressions Problem*" where one can add new Data (*e.g.*, Add, Subtract) and new Operations (*e.g.*, Print, Evaluate) over these data [Wadler, 1998]. It has several implementations in different programming languages. Our three implementations are based on the Scala capabilities to handle this problem [Lopez-Herrejon et al., 2005; Loverdos and Syropoulos, 2010; Zenger and Odersky, 2004]. It consists of 4 mandatory features and 3 optional features. Its feature model and the used techniques to implement two of its versions are also given on Page 42.

JavaGeom

Version:	2009 - 2017 (last change)
Features:	110 (110 developed)
vp-s with *variants*:	199 with 269
Developed by:	David Legland (open source)
Developed as:	A feature-rich system
Language:	Java
Size:	35 456 lines of code, 142 files

Description. JavaGeom is a feature-rich system in Java, which is developed by Legland [2017]. It is an open source geometry library for Java that is architected around well-identified features, although not presented as a product line. We selected JavaGeom as a relevant case for demonstrating the applicability of our tooled framework on a feature-rich system, being the size of medium-size SPLs and easily understandable. Some more details about its development and our usage are given in Section 7.2 (Page 94).

B.2 Implementation of Expressions PL - Versions 1 and 3

```
// Base interface
trait ExpAlg[E] {
  def lit(x: Int): E
}
// Adding subtraction
trait AddExpAlg[E] extends ExpAlg[E] {
  def add(e1: E, e2: E): E
}
// Adding adding
trait SubExpAlg[E] extends ExpAlg[E] {
  def sub(e1: E, e2: E): E
}
// Adding pretty printing
trait Echo {
  def print(): String
}
trait PrintExpAlg extends ExpAlg[Echo] {
  def lit(x: Int) = new Echo() {
    def print() = x.toString()
  }
}
// Updating evaluations
trait PrintAddExpAlg extends PrintExpAlg with AddExpAlg[Echo] {
  def add(e1: Echo, e2: Echo) = new Echo {
    def print() = e1.print() + " + " + e2.print()
  }
}
trait PrintSubExpAlg extends PrintExpAlg with SubExpAlg[Echo] {
  def sub(e1: Echo, e2: Echo) = new Echo() {
    def print() = e1.print() + " - " + e2.print()
  }
}
// The evaluation interface
trait Eval {
  def eval(): Int
}
// Updating evaluation
trait EvalExpAlg extends SubExpAlg[Eval] with AddExpAlg[Eval] {
  def lit(x: Int) = new Eval() {
    def eval() = x
  }
  def add(e1: Eval, e2: Eval) = new Eval() {
    def eval() = e1.eval() +  e2.eval()
  }
  def sub(e1: Eval, e2: Eval) = new Eval() {
    def eval() = e1.eval() - e2.eval()
  }
```

```
48 }
49 /* An alternative implementation
50 ** of pretty printing that directly computes a string */
51
52 trait PrintExpAlg2 extends SubExpAlg[String]{
53   def lit(x: Int) = x.toString()
54   def add(e1: String, e2: String) = e1 + " + " + e2
55   def sub(e1: String, e2: String) = e1 + " - " + e2
56 }
57 // Test
58 object ExpressionProblem {
59   def test(): Unit = {
60     class Core extends PrintSubExpAlg with PrintAddExpAlg
61
62     val pa = new Core
63     val ea = new EvalExpAlg() {}
64     val pa2 = new PrintExpAlg2() {}
65     val exp = pa.add(pa.lit(5), pa.lit(7)).print() + " = " + ea.
    add(ea.lit(5), ea.lit(7)).eval()
66     val exp2 = pa2.sub(pa2.lit(5), pa2.lit(7)) + " = " + ea.sub(
    ea.lit(5), ea.lit(7)).eval()
67     println(exp)
68     print(exp2)
69   }
70   def main(args: Array[String]) {
71     test
72   }
73 }
```

Listing B.1: Expressions PL implementation - version 1, in Scala language

```
1  sealed trait Expr
2  case class Lit(n: Int) extends Expr
3  case class Add(l: Expr, r: Expr) extends Expr
4  case class Sub(l: Expr, r: Expr) extends Expr
5  class Eval {
6    def eval(e: Expr): Int = e match {
7      case Lit(n) => n
8      case Add(l, r) => eval(l) + eval(r)
9      case Sub(l, r) => eval(l) - eval(r)
10   }
11 }
12 class Echo {
13   def echo(e: Expr): Unit = e match {
14     case Lit(n) => print(" " + n + " ")
15     case Add(l, r) => echo(l); print("+"); echo(r)
16     case Sub(l, r) => echo(l); print("-"); echo(r)
17   }
18 }
19 //Test
```

```
20 object ExpressionProblem {
21   def main(args: Array[String]) {
22     val eval = new Eval
23     val echo = new Echo
24
25     val expr1 =
26       Sub(Add(Lit(4), Sub(Lit(7), Lit(10))),
27         Add(Lit(4), Sub(Lit(7), Lit(10))))
28
29     val x = eval.eval(expr1)
30     val p = echo.echo(expr1)
31
32     p + "" + println(" = " + x)
33   }
34 }
```

Listing B.2: Expressions PL implementation - version 2, in Scala language

Bibliography

Acher, M. (2011). *Managing Multiple Feature Models: Foundations, Language and Applications*. PhD dissertation, Université de Nice-Sophia Antipolis. (Cited on page 79.)

Acher, M., Collet, P., Lahire, P., and France, R. B. (2011). Slicing feature models. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 424–427. IEEE Computer Society. (Cited on pages 79, 80 and 86.)

Acher, M., Collet, P., Lahire, P., and France, R. B. (2013). Familiar: A domain-specific language for large scale management of feature models. *Science of Computer Programming (SCP)*, 78(6):657–681. (Cited on pages 24, 111 and 119.)

AGM (2009). Arcade game maker pedagogical product line. `http://www.sei.cmu.edu/productlines/ppl/`. [Accessed 2016-11-02]. (Cited on pages 37, 95 and 126.)

Alexander, C. (2004). The phenomenon of life: The nature of order, book 1. (Cited on page 118.)

Alves, V., Schneider, D., Becker, M., Bencomo, N., and Grace, P. (2009). Comparitive study of variability management in software product lines and runtime adaptable systems. In *VaMoS*. Universität Duisburg-Essen. (Cited on page 31.)

American Heritage (1985). *The American Heritage Dictionar*. Houghton Mifflin, Boston, MA. (Cited on page 1.)

Anastasopoulos, M. and Muthig, D. (2004). An evaluation of aspect-oriented programming as a product line implementation technology. In *Software Reuse: Methods, Techniques, and Tools*, pages 141–156. Springer. (Cited on page 40.)

Anquetil, N., Grammel, B., Galvao Lourenco da Silva, I., Noppen, J., Shakil Khan, S., Arboleda, H., Rashid, A., and Garcia, A. (2008). Traceability for model driven, software product line engineering. (Cited on pages 18 and 19.)

Anquetil, N., Kulesza, U., Mitschke, R., Moreira, A., Royer, J.-C., Rummler, A., and Sousa, A. (2010). A model-driven traceability framework for software product lines. *Software & Systems Modeling*, 9(4):427–451. (Cited on pages xi, 18, 19, 20 and 67.)

Antkiewicz, M., Ji, W., Berger, T., Czarnecki, K., Schmorleiz, T., Lämmel, R., Stănciulescu, S., Wąsowski, A., and Schaefer, I. (2014). Flexible product line engineering with a virtual platform. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 532–535. ACM. (Cited on pages 38 and 40.)

Apel, S., Batory, D., Kästner, C., and Saake, G. (2013). *Feature-Oriented Software Product Lines*. Springer. (Cited on pages xv, 5, 6, 14, 15, 16, 17, 18, 21, 29, 32, 34, 35, 37, 39, 40, 47, 58, 60 and 119.)

Apel, S. and Kästner, C. (2009). An overview of feature-oriented software development. *Journal of Object Technology*, 8(5):49–84. (Cited on page 43.)

Apel, S., Leich, T., and Saake, G. (2008). Aspectual feature modules. *Software Engineering, IEEE Transactions on*, 34(2):162–180. (Cited on pages 21 and 40.)

Apel, S., Lengauer, C., Batory, D., Möller, B., and Kästner, C. (2007). An algebra for feature-oriented software development. *University of Passau, MIP-0706*. (Cited on page 122.)

Assunção, W. K. G. and Vergilio, S. R. (2014). Feature location for software product line migration: a mapping study. In *Proceedings of the 18th International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools-Volume 2*, pages 52–59. ACM. (Cited on page 21.)

Bachmann, F. and Clements, P. C. (2005). Variability in software product lines. Technical report, DTIC Document. (Cited on pages 5, 8, 14, 16, 17, 29, 43, 48 and 123.)

Bachmann, F., Goedicke, M., Leite, J., Nord, R., Pohl, K., Ramesh, B., and Vilbig, A. (2003). A meta-model for representing variability in product family development. In *International Workshop on Software Product-Family Engineering*, pages 66–80. Springer. (Cited on page 20.)

Barais, O., Baudry, B., Blouin, A., Combemale, B., Jézéquel, J.-M., and Vojtisek, D. (2013). A demonstration for building modular and efficient dsls. In *CIEL 2013 2ème Conférence en IngénieriE du Logiciel*. (Cited on pages 16 and 25.)

Bassett, P. G. (1996). *Framing software reuse: lessons from the real world*. Prentice-Hall, Inc. (Cited on pages 35 and 36.)

Batory, D. (2005). Feature models, grammars, and propositional formulas. In *International Conference on Software Product Lines*, pages 7–20. Springer. (Cited on page 22.)

Batory, D. (2006). Feature modularity for product-lines. *Tutorial at: OOPSLA*, 6. (Cited on page 122.)

Batory, D., Sarvela, J. N., and Rauschmayer, A. (2004). Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371. (Cited on pages 35, 60 and 121.)

Bayer, J. and Widen, T. (2001). Introducing traceability to product lines. In *International Workshop on Software Product-Family Engineering*, pages 409–416. Springer. (Cited on page 20.)

Becker, M. (2003a). Mapping variabilities onto product family assets. In *Proceedings of the International Colloquium of the Sonderforschungsbereich*, volume 501. Citeseer. (Cited on pages 12, 15, 20 and 122.)

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*, pages 19–27. (Cited on pages 12, 15, 16 and 20.)

Ben-Ari, M. (2012). Propositional logic: Formulas, models, tableaux. In *Mathematical Logic for Computer Science*, pages 7–47. Springer. (Cited on pages 76 and 80.)

Benavides, D., Martín-Arroyo, P. T., and Cortés, A. R. (2005). Automated reasoning on feature models. In *CAiSE*, volume 5, pages 491–503. Springer. (Cited on page 11.)

Benavides, D., Segura, S., and Ruiz-Cortés, A. (2010). Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636. (Cited on pages 22, 23, 78, 103, 111 and 117.)

Benavides, D., Segura, S., Trinidad, P., and Cortés, A. R. (2007). Fama: Tooling a framework for the automated analysis of feature models. *VaMoS*, 2007:01. (Cited on pages 104, 111 and 117.)

Benavides, D., Segura, S., Trinidad, P., and Ruiz-Cortés, A. (2006). A first step towards a framework for the automated analysis of feature models. *Proc. Managing Variability for Software Product Lines: Working With Variability Mechanisms*, pages 39–47. (Cited on pages 104 and 111.)

Berg, K., Bishop, J., and Muthig, D. (2005). Tracing software product line variability: from problem to solution space. In *Proceedings of the 2005 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries*, pages 182–191. South African Institute for Computer Scientists and Information Technologists. (Cited on pages 6, 18, 20, 21 and 121.)

Berre, L. (2013). SAT4j documentation and test examples. `http://www.sat4j.org/maven234/org.ow2.sat4j.core/xref-test/index.html`. [Accessed 2017-08-14]. (Cited on page 103.)

Beuche, D., Papajewski, H., and Schröder-Preikschat, W. (2004). Variability management with feature models. *Science of Computer Programming*, 53(3):333–352. (Cited on pages 15 and 16.)

Bosch, J. (2000). *Design and use of software architectures: adopting and evolving a product-line approach*. Pearson Education. (Cited on pages 31, 38, 40 and 121.)

Bosch, J. and Capilla, R. (2012). Dynamic variability in software-intensive embedded system families. *Computer*, 45(10):28–35. (Cited on pages xiii and 32.)

Bosch, J., Florijn, G., Greefhorst, D., Kuusela, J., Obbink, J. H., and Pohl, K. (2001). Variability issues in software product lines. In *Software Product-Family Engineering*, pages 13–21. Springer. (Cited on pages 3, 6, 49 and 114.)

Bühne, S., Halmans, G., and Pohl, K. (2003). Modelling dependencies between variation points in use case diagrams. In *REFSQ*, volume 3, pages 59–69. Citeseer. (Cited on pages 31 and 118.)

Capilla, R., Bosch, J., and Kang, K.-C. (2013). *Systems and Software Variability Management*. Springer. (Cited on pages xi, xiii, 7, 11, 12, 13, 15, 16, 17, 20, 30, 31, 32, 69 and 74.)

Castellani, E. (2003). On the meaning of symmetry breaking. *Symmetries in physics: Philosophical reflections*, pages 321–334. (Cited on page 16.)

Chechik, M., Di Sandro, A., Famelis, M., and Salay, R. (2012-2015). MMINT: A graphical tool for interactive model management. `http://www.programcreek.com/java-api-examples/index.php?source_dir=MMINT-master/examples/`. [Accessed 2017-08-14]. (Cited on page 104.)

Chen, K., Zhang, W., Zhao, H., and Mei, H. (2005). An approach to constructing feature models based on requirements clustering. In *13th IEEE International Conference on Requirements Engineering (RE'05)*, pages 31–40. IEEE. (Cited on page 122.)

Classen, A., Boucher, Q., and Heymans, P. (2011). A text-based approach to feature modelling: Syntax and semantics of tvl. *Science of Computer Programming*, 76(12):1130–1143. (Cited on page 24.)

Classen, A., Heymans, P., and Schobbens, P.-Y. (2008). What's in a feature: A requirements engineering perspective. In *International Conference on Fundamental Approaches to Software Engineering*, pages 16–30. Springer. (Cited on pages 14 and 122.)

Clauß, M. (2001). Generic modeling using uml extensions for variability. In *Workshop on Domain Specific Visual Languages at OOPSLA*, volume 2001. (Cited on page 19.)

Clauß, M. and Jena, I. (2001). Modeling variability with uml. In *GCSE 2001 Young Researchers Workshop*. Citeseer. (Cited on pages 18 and 122.)

Cleland-Huang, J., Gotel, O., and Zisman, A. (2012). *Software and systems traceability*, volume 2. Springer. (Cited on pages 18 and 19.)

Cleland-Huang, J., Gotel, O. C., Huffman Hayes, J., Mäder, P., and Zisman, A. (2014). Software traceability: trends and future directions. In *Proceedings of the on Future of Software Engineering*, pages 55–69. ACM. (Cited on page 19.)

Clements, P. and Northrop, L. (2002). *Software product lines*. Addison-Wesley,. (Cited on pages 1, 2 and 3.)

Coplien, J., Hoffman, D., and Weiss, D. (1998). Commonality and variability in software engineering. *Software, IEEE*, 15(6):37–45. (Cited on page 40.)

Coplien, J. O. (1999). *Multi-paradigm Design for C++*. Addison-Wesley. (Cited on pages 5, 13, 14, 21, 29, 32, 39, 40 and 43.)

Coplien, J. O. and Zhao, L. (2000). Symmetry breaking in software patterns. In *International Symposium on Generative and Component-Based Software Engineering*, pages 37–54. Springer. (Cited on page 16.)

Corporation, S. P. C. S. (1993). *Reuse-driven Software Processes Guidebook: SPC-92019-CMC, Version 02.00. 03*. Software Productivity Consortium Services Corporation. (Cited on page 12.)

CVL (2012). Common variability language. `http://www.omgwiki.org/variability/doku.php`. Accessed: 2016-26-09. (Cited on pages 16, 24 and 52.)

Czarnecki, K. (2005). Overview of generative software development. In *Unconventional Programming Paradigms*, pages 326–341. Springer. (Cited on pages xi, 2, 12 and 36.)

Czarnecki, K. and Antkiewicz, M. (2005). Mapping features to models: A template approach based on superimposed variants. In *International conference on generative programming and component engineering*, pages 422–437. Springer. (Cited on page 19.)

Czarnecki, K., Bednasch, T., Unger, P., and Eisenecker, U. W. (2002). Generative programming for embedded software: An industrial experience report. In *GPCE*, volume 2, pages 156–172. Springer. (Cited on page 11.)

Czarnecki, K. and Eisenecker, U. W. (2000). Generative programming. *Edited by G. Goos, J. Hartmanis, and J. van Leeuwen*, page 15. (Cited on pages 2, 11, 12, 15, 36, 121 and 122.)

Czarnecki, K., Grünbacher, P., Rabiser, R., Schmid, K., and 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*, pages 173–182. ACM. (Cited on pages 5, 12 and 13.)

Czarnecki, K., Helsen, S., and Eisenecker, U. (2004). Staged configuration using feature models. In *SPLC*, volume 3154, pages 266–283. Springer. (Cited on page 11.)

Czarnecki, K. and Wasowski, A. (2007). Feature diagrams and logics: There and back again. In *Software Product Line Conference, 2007. SPLC 2007. 11th International*, pages 23–34. IEEE. (Cited on page 22.)

de Oliveira Junior, E. A., Gimenes, I., Huzita, E. H. M., and Maldonado, J. C. (2005). A variability management process for software product lines. In *Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research*, pages 225–241. IBM Press. (Cited on page 20.)

Deelstra, S., Sinnema, M., and Bosch, J. (2004). Experiences in software product families: Problems and issues during product derivation. *Software Product Lines*, pages 120–122. (Cited on pages 19 and 22.)

Deelstra, S., Sinnema, M., and Bosch, J. (2005). Product derivation in software product families: a case study. *Journal of Systems and Software*, 74(2):173–194. (Cited on page 19.)

Dhungana, D. and Grünbacher, P. (2008). Understanding decision-oriented variability modelling. In *SPLC (2)*, pages 233–242. (Cited on page 12.)

Dolstra, E., Florijn, G., and Visser, E. (2003a). Capturing timeline variability with transparent configuration environments. (Cited on page 17.)

Dolstra, E., Florijn, G., and Visser, E. (2003b). Timeline variability: The variability of binding time of variation points. (Cited on page 17.)

El-Sharkawy, S., Dederichs, S., and Schmid, K. (2012). From feature models to decision models and back again an analysis based on formal transformations. In *Proceedings of the 16th International Software Product Line Conference-Volume 1*, pages 126–135. ACM. (Cited on page 12.)

Eriksson, M., Börstler, J., and Borg, K. (2005). The PLUSS approach–domain modeling with features, use cases and use case realizations. *Software Product Lines*, pages 33–44. (Cited on page 11.)

Forster, T., Muthig, D., and Pech, D. (2008). Understanding decision models-visualization and complexity reduction of software variability. In *VaMoS*, pages 111–119. (Cited on pages 12 and 123.)

Fowler, M. and Parsons, R. (2010). *Domain-specific language*. Pearson Education. (Cited on page 91.)

Fritsch, C., Lehn, A., Strohm, T., and Bosch, R. (2002). Evaluating variability implementation mechanisms. In *Proceedings of International Workshop on Product Line Engineering (PLEES)*, pages 59–64. (Cited on pages 14, 17, 34 and 41.)

Gacek, C. and Anastasopoules, M. (2001). Implementing product line variabilities. In *ACM SIGSOFT Software Engineering Notes*, volume 26, pages 109–117. ACM. (Cited on pages 5, 17, 29, 34, 35, 37, 38, 39 and 40.)

Ghosh, D. (2010). *DSLs in action*. Manning Publications Co. (Cited on pages 24 and 91.)

Gomaa, H. (2004). Designing software product lines with uml: From use cases to pattern-based software architectures. (Cited on page 52.)

Gomaa, H. (2005). *Designing software product lines with UML*. IEEE. (Cited on pages 19, 37, 95, 121 and 126.)

Gotel, O., Cleland-Huang, J., Hayes, J. H., Zisman, A., Egyed, A., Grünbacher, P., Dekhtyar, A., Antoniol, G., and Maletic, J. (2012). The grand challenge of traceability (v1. 0). In *Software and Systems Traceability*, pages 343–409. Springer. (Cited on page 19.)

Griss, M. L. (2000). Implementing product-line features by composing aspects. In *Software Product Lines*, pages 271–288. Springer. (Cited on page 15.)

Griss, M. L., Favaro, J., and d'Alessandro, M. (1998). Integrating feature modeling with the rseb. In *Software Reuse, 1998. Proceedings. Fifth International Conference on*, pages 76–85. IEEE. (Cited on page 11.)

Gupta, H. (2014). Implementation of algorithms from "artificial intelligence: A modern approach 3rd ed" in scala. https://github.com/himanshug/aima-scala. [Accessed 2017-08-14]. (Cited on page 103.)

Hein, A., Schlick, M., and Vinga-Martins, R. (2000). Applying feature models in industrial settings. *Software Product Lines. Experience and Research Directions*, pages 47–70. (Cited on page 11.)

Henard, C. (2012). Feature model automated analysis. http://www.programcreek.com/java-api-examples/index.php?source_dir=pledge-master/src/. [Accessed 2017-08-14]. (Cited on page 104.)

Heymans, P., Boucher, Q., Classen, A., Bourdoux, A., and Demonceau, L. (2012). A code tagging approach to software product line development. *International Journal on Software Tools for Technology Transfer*, 14(5):553–566. (Cited on pages 6, 16, 17, 18, 19, 21, 36, 100 and 117.)

Holthusen, S., Proksch, F., and Krueger, S. (2005-2015). Featureide: A framework for feature-oriented software development. https://github.com/SINTEF-9012/bvr/blob/master/de.ovgu.featureide.fm.core/src/de/ovgu/featureide/fm/core/FeatureModelAnalyzer.java. [Accessed 2017-08-14]. (Cited on page 104.)

Hotz, L., Wolter, K., Krebs, T., Deelstra, S., Sinnema, M., Nijhuis, J., and MacGregor, J. (2006). *Configuration in industrial product families*. Ios Press. (Cited on page 122.)

Hunt, J. M. and McGregor, J. D. (2006). 9 implementing a variation point: A pattern language. *Variability Management–Working with Variability Mechanisms*, page 83. (Cited on pages 6, 15 and 123.)

ISO/IEC 26550:2015 (2015). Software and systems engineering - reference model for product line engineering and management. `https://www.iso.org/standard/69529.html`. ISO/IEC JTC 1/SC 7. (Cited on pages 15 and 123.)

Jacobson, I., Griss, M., and Jonsson, P. (1997). *Software reuse: architecture, process and organization for business success*. ACM Press/Addison-Wesley Publishing Co. (Cited on pages 5, 14, 15, 16, 17, 51, 52, 58, 121 and 122.)

Jarzabek, S., Bassett, P., Zhang, H., and Zhang, W. (2003). Xvcl: Xml-based variant configuration language. In *Proceedings of the 25th International Conference on Software Engineering*, pages 810–811. IEEE Computer Society. (Cited on page 36.)

Jirapanthong, W. and Zisman, A. (2009). Xtraque: traceability for product line systems. *Software & Systems Modeling*, 8(1):117–144. (Cited on page 20.)

John, I. (2010). *Pattern-based documentation analysis for software product lines*. University of Kaiserslautern. (Cited on page 122.)

John, I., Lee, J., and Muthig, D. (2007). Separation of variability dimension and development dimension. In *VaMoS*, pages 45–49. (Cited on pages 6, 21 and 123.)

Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., and Peterson, A. S. (1990). Feature-oriented domain analysis (FODA) feasibility study. Technical report, DTIC Document. (Cited on pages 1, 3, 11, 12, 14, 30 and 121.)

Kang, K. C., Kim, S., Lee, J., Kim, K., Shin, E., and Huh, M. (1998). FORM: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5(1):143–168. (Cited on pages 1, 7, 11, 12, 14 and 121.)

Kästner, C. (2010). *Virtual Separation of Concerns: Toward Preprocessors 2.0*. PhD dissertation, Otto-von-Guericke-Universität Magdeburg. (Cited on pages 17, 35 and 36.)

Kästner, C. (2012). Virtual separation of concerns: Toward preprocessors 2.0. *it-Information Technology Methoden und innovative Anwendungen der Informatik und Informationstechnik*, 54(1):42–46. (Cited on pages 6 and 15.)

Kästner, C. and Apel, S. (2008). Integrating compositional and annotative approaches for product line engineering. In *Proc. GPCE Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering*, pages 35–40. (Cited on pages 35, 36 and 41.)

Kästner, C. and Apel, S. (2009). Virtual separation of concerns-a second chance for preprocessors. *Journal of Object Technology*, 8(6):59–78. (Cited on page 16.)

Kästner, C., Apel, S., and Kuhlemann, M. (2008a). Granularity in software product lines. In *Proceedings of the 30th international conference on Software engineering*, pages 311–320. ACM. (Cited on page 32.)

Kästner, C., Apel, S., and Ostermann, K. (2011). The road to feature modularity? In *Proceedings of the 15th International Software Product Line Conference, Volume 2*, page 5. ACM. (Cited on pages 15 and 37.)

Kästner, C., Ostermann, K., and Erdweg, S. (2012). A variability-aware module system. In *ACM SIGPLAN Notices*, volume 47, pages 773–792. ACM. (Cited on page 6.)

Kästner, C., Trujillo, S., and Apel, S. (2008b). Visualizing software product line variabilities in source code. In *SPLC (2)*, pages 303–312. (Cited on pages 18 and 21.)

KCVL (2015). The CVL implementation in Kermeta. `https://diverse-project.github.io/kcvl/`. (Cited on pages 16, 25 and 119.)

Kleene, S. C. (2002). *Mathematical logic*. Courier Corporation. (Cited on pages 77 and 81.)

Krieter, S., Schröter, R., Thüm, T., Fenske, W., and Saake, G. (2016). Comparing algorithms for efficient feature-model slicing. In *Proceedings of the 20th International Systems and Software Product Line Conference*, pages 60–64. ACM. (Cited on page 80.)

Krueger, C. (2002a). Easing the transition to software mass customization. In *Software Product-Family Engineering*, pages 282–293. Springer. (Cited on pages 1 and 44.)

Krueger, C. W. (2002b). Variation management for software production lines. In *Software product lines*, pages 37–48. Springer. (Cited on pages 3 and 18.)

Le, D. M., Lee, H., Kang, K. C., and Keun, L. (2013). Validating consistency between a feature model and its implementation. In *International Conference on Software Reuse*, pages 1–16. Springer. (Cited on pages 7, 8, 22, 86 and 87.)

Le Berre, D. and Parrain, A. (2010). The sat4j library, release 2.2, system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64. (Cited on page 103.)

Lee, J. and Kang, K. C. (2004). Feature binding analysis for product line component development. In *Software Product-Family Engineering*, pages 250–260. Springer. (Cited on page 31.)

Lee, J., Kang, K. C., Sawyer, P., and Lee, H. (2014). A holistic approach to feature modeling for product line requirements engineering. *Requirements Engineering*, 19(4):377–395. (Cited on pages 11 and 12.)

Lee, J. and Muthig, D. (2008). Feature-oriented analysis and specification of dynamic product reconfiguration. In *High Confidence Software Reuse in Large Systems*, pages 154–165. Springer. (Cited on page 31.)

Lee, K., Kang, K. C., Chae, W., and Choi, B. W. (2000). Feature-based approach to object-oriented engineering of applications for reuse. *Software-Practice and Experience*, 30(9):1025–1046. (Cited on page 17.)

Legland, D. (2009 - 2017). JavaGeom. `http://geom-java.sourceforge.net/index.html`. Accessed June 15, 2017. (Cited on pages 45, 95 and 127.)

Lopez-Herrejon, R. E. and Batory, D. (2001). A standard problem for evaluating product-line methodologies. In *Generative and Component-Based Software Engineering*, pages 10–24. Springer. (Cited on pages 3, 11, 37, 95 and 125.)

Lopez-Herrejon, R. E., Batory, D., and Cook, W. (2005). *Evaluating support for features in advanced modularization technologies*. Springer. (Cited on pages 37, 41 and 127.)

Lotufo, R., She, S., Berger, T., Czarnecki, K., and Wąsowski, A. (2010). Evolution of the linux kernel variability model. In *International Conference on Software Product Lines*, pages 136–150. Springer. (Cited on pages 7 and 8.)

Loverdos, C. K. and Syropoulos, A. (2010). *Steps in scala: an introduction to object-functional programming*. Cambridge University Press. (Cited on pages 41 and 127.)

Lozano, A. (2011). An overview of techniques for detecting software variability concepts in source code. In *International Conference on Conceptual Modeling*, pages 141–150. Springer. (Cited on pages 22 and 58.)

Mannion, M. (2002). Using first-order logic for product line model validation. *Software Product Lines*, pages 149–202. (Cited on page 22.)

Mendonca, M., Branco, M., and Cowan, D. (2009). Splot: software product lines online tools. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 761–762. ACM. (Cited on page 24.)

Metzger, A. and Heymans, P. (2007). Comparing feature diagram examples found in the research literature. *Technical report, Univ. of Duisburg-Essen*. (Cited on pages 7 and 12.)

Metzger, A. and Pohl, K. (2014). Software product line engineering and variability management: achievements and challenges. In *Proceedings of the on Future of Software Engineering*, pages 70–84. ACM. (Cited on page 17.)

Metzger, A., Pohl, K., Heymans, P., Schobbens, P.-Y., and Saval, G. (2007). Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. In *Requirements Engineering Conference, 2007. RE'07. 15th IEEE International*, pages 243–253. IEEE. (Cited on pages 7, 12, 15, 24 and 86.)

Mezini, M. and Ostermann, K. (2004). Variability management with feature-oriented programming and aspects. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 127–136. ACM. (Cited on page 21.)

Mohan, K. (2003). Tracing variability in software product families. `http://www.idea-group.com/`. (Cited on page 20.)

Muthig, D. and Atkinson, C. (2002). Model-driven product line architectures. In *International Conference on Software Product Lines*, pages 110–129. Springer. (Cited on pages 6, 12, 21 and 51.)

Muthig, D. and Patzke, T. (2003). Generic implementation of product line components. In *Objects, Components, Architectures, Services, and Applications for a Networked World*, pages 313–329. Springer. (Cited on pages 5, 17, 21, 29, 35, 39 and 40.)

Northrop, L., Clements, P., Bachmann, F., Bergey, J., Chastek, G., Cohen, S., Donohoe, P., Jones, L., Krut, R., Little, R., et al. (2007). A framework for software product line practice, version 5.0. *SEI.–2007–http://www. sei. cmu. edu/product-lines/index. html*. (Cited on pages 1 and 2.)

Odersky, M., Spoon, L., and Venners, B. (2007-2010). *Programming in scala*. Artima Inc. (Cited on pages 24, 91 and 93.)

Patzke, T. (2010). The impact of variability mechanisms on sustainable product line code evolution. In *Software Engineering*, pages 189–200. (Cited on pages 38 and 40.)

Patzke, T. and Muthig, D. (2002). Product line implementation technologies. programming language view. *Fraunhofer IESE Report, 057.02/E*. (Cited on pages 5, 29, 35, 39 and 40.)

Patzke, T. and Muthig, D. (2003). Product line implementation with frame technology: A case study. *Fraunhofer IESE Report, 018.03/E*, 18. (Cited on pages 36, 38, 39 and 40.)

Patzke, T. B., Fraunhofer, K., Rombach, D., Liggesmeyer, P., and Bomarius, F. (2011). *Sustainable Evolution of Product Line Infrastructure Code (PhD Theses in Experimental Software Engineering)*. Fraunhofer IRB Verlag. (Cited on pages 14, 34, 38, 39 and 40.)

Pohl, K., Böckle, G., and van der Linden, F. J. (2005). *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media. (Cited on pages xi, 1, 2, 6, 7, 12, 15, 16, 18, 20, 67, 86, 121 and 123.)

Rabiser, R., Grunbacher, P., and Dhungana, D. (2007). Supporting product derivation by adapting and augmenting variability models. In *Software Product Line Conference, 2007. SPLC 2007. 11th International*, pages 141–150. IEEE. (Cited on page 123.)

Ramos Alves, V. (2007). *Implementing software product line adoption strategies*. PhD dissertation, Universidade Federal de Pernambuco. (Cited on page 38.)

Reiser, M.-O. (2008). *Managing complex variability in automotive software product lines with subscoping and configuration links*. PhD dissertation, Elektrotechnik und Informatik der Technischen Universität Berlin. (Cited on page 122.)

Riebisch, M. (2003). Towards a more precise definition of feature models. *Modelling Variability for Object-Oriented Product Lines*, pages 64–76. (Cited on page 121.)

Riebisch, M., Böllert, K., Streitferdt, D., and Philippow, I. (2002). Extending feature diagrams with uml multiplicities. In *6th World Conference on Integrated Design & Process Technology (IDPT2002)*, volume 23. (Cited on page 11.)

Rosen, J. (1995). *Symmetry in science*. Springer. (Cited on page 16.)

Rosenmüller, M., Siegmund, N., Apel, S., and Saake, G. (2011). Flexible feature binding in software product lines. *Automated Software Engineering*, 18(2):163–197. (Cited on pages 17 and 32.)

Rubin, J. and Chechik, M. (2013). A survey of feature location techniques. In *Domain Engineering*, pages 29–58. Springer. (Cited on page 21.)

Santos, A. R., de Oliveira, R. P., and de Almeida, E. S. (2015). Strategies for consistency checking on software product lines: a mapping study. In *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, page 5. ACM. (Cited on pages 7 and 86.)

Savolainen, J. and Kuusela, J. (2001). Violatility analysis framework for product lines. In *ACM SIGSOFT Software Engineering Notes*, volume 26 – 3, pages 133–141. ACM. (Cited on page 121.)

Schaefer, I., Bettini, L., Bono, V., Damiani, F., and Tanzarella, N. (2010). Delta-oriented programming of software product lines. In *Software Product Lines: Going Beyond*, pages 77–91. Springer. (Cited on pages 18, 21 and 35.)

Schmid, K. and John, I. (2003). Generic variability management and its application to product line modelling. *Software Variability Management*, page 13. (Cited on pages 6 and 20.)

Schmid, K. and John, I. (2004). A customizable approach to full lifecycle variability management. *Science of Computer Programming*, 53(3):259–284. (Cited on pages 5, 12, 15, 16, 18, 20 and 123.)

Schmid, K., Rabiser, R., and Grünbacher, P. (2011). A comparison of decision modeling approaches in product lines. In *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*, pages 119–126. ACM. (Cited on page 12.)

Sinnema, M., Deelstra, S., Nijhuis, J., and Bosch, J. (2004a). Covamof: A framework for modeling variability in software product families. In *Software product lines*, pages 197–213. Springer. (Cited on pages 6, 49 and 118.)

Sinnema, M., Deelstra, S., Nijhuis, J., and Bosch, J. (2004b). Managing variability in software product families. In *Proceedings of the 2nd groningen workshop on software variability management*. (Cited on pages 20 and 122.)

Spanoudakis, G. and Zisman, A. (2001). Inconsistency management in software engineering: Survey and open research issues. *Handbook of software engineering and knowledge engineering*, 1:329–380. (Cited on page 74.)

Svahnberg, M., Van Gurp, J., and Bosch, J. (2005). A taxonomy of variability realization techniques. *Software: Practice and Experience*, 35(8):705–754. (Cited on pages xi, 5, 17, 20, 21, 29, 32, 35, 40, 51 and 123.)

Swe, S. M., Zhang, H., and Jarzabek, S. (2002). Xvcl: a tutorial. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 341–349. ACM. (Cited on page 36.)

Tartler, R., Sincero, J., Dietrich, C., Schröder-Preikschat, W., and Lohmann, D. (2012). Revealing and repairing configuration inconsistencies in large-scale system software. *International Journal on Software Tools for Technology Transfer*, 14(5):531–551. (Cited on pages 6, 7, 8, 18, 22, 36 and 87.)

Tërnava, Xh. and Collet, P. (2017a). Early consistency checking between specification and implementation variabilities. In *Preceedings of the 21st International Systems and Software Product Line Conference - Volume A*, SPLC '17, pages 29–38. ACM. (Not cited.)

Tërnava, Xh. and Collet, P. (2017b). On the diversity of capturing variability at the implementation level. In *Proceedings of the 21st International Systems and Software Product Line Conference - Volume B*, SPLC '17, pages 81–88. ACM. (Not cited.)

Tërnava, Xh. and Collet, P. (2017). Tracing imperfectly modular variability in software product line implementation. In *International Conference on Software Reuse*, pages 112–120. Springer. (Not cited.)

Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G., and Leich, T. (2014). Featureide: An extensible framework for feature-oriented software development. *Science of Computer Programming*, 79:70–85. (Cited on pages 16, 22, 24, 103 and 119.)

Thum, T., Kastner, C., Erdweg, S., and Siegmund, N. (2011). Abstract features in feature modeling. In *Software Product Line Conference (SPLC), 2011 15th International*, pages 191–200. IEEE. (Cited on pages 33 and 70.)

Turner, C. R., Fuggetta, A., Lavazza, L., and Wolf, A. L. (1999). A conceptual basis for feature engineering. *Journal of Systems and Software*, 49(1):3–15. (Cited on pages 12 and 13.)

Van Dalen, D. (2004). *Logic and structure*. Springer. (Cited on page 81.)

Van Deursen, A., Klint, P., and Visser, J. (2000). Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36. (Cited on page 24.)

Van Gurp, J., Bosch, J., and Svahnberg, M. (2001). On the notion of variability in software product lines. In *Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on*, pages 45–54. IEEE. (Cited on pages 11, 15 and 122.)

Vierhauser, M., Grünbacher, P., Egyed, A., Rabiser, R., and Heider, W. (2010). Flexible and scalable consistency checking on product line variability models. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 63–72. ACM. (Cited on pages 7, 8, 78 and 87.)

Voelter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L., Visser, E., and Wachsmuth, G. (2013). *DSL engineering*. `http://dslbook.org`. (Cited on pages 24 and 91.)

Voelter, M. and Visser, E. (2011). Product line engineering using domain-specific languages. In *Software Product Line Conference (SPLC), 2011 15th International*, pages 70–79. IEEE. (Cited on pages 12, 24 and 91.)

Volter, M. (2011). Dsls for product lines: Approaches, tools, experiences. In *Software Product Line Conference (SPLC), 2011 15th International*, pages 353–353. IEEE. (Cited on page 24.)

Wadler, P. (1998). The expression problem. *Java-genericity mailing list*. (Cited on page 127.)

Weiss, D. M. et al. (1999). *Software product-line engineering: a family-based software development process*. Addison-Wesley Professional; Har/Cdr edition. (Cited on pages 1, 24 and 36.)

Winkler, S. and Pilgrim, J. (2010). A survey of traceability in requirements engineering and model-driven development. *Software and Systems Modeling (SoSyM)*, 9(4):529–565. (Cited on page 18.)

Xue, Y., Xing, Z., and Jarzabek, S. (2012). Feature location in a collection of product variants. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pages 145–154. IEEE. (Cited on page 21.)

Zave, P. (1999). Faq sheet on feature interaction. *Link: http://www. research. att. com/~ pamela/faq. html*. (Cited on pages 31 and 121.)

Zenger, M. and Odersky, M. (2004). Independently extensible solutions to the expression problem. Technical report. (Cited on page 127.)

Zhang, B., Becker, M., Patzke, T., Sierszecki, K., and 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*, pages 168–177. ACM. (Cited on page 101.)

Zhao, L. and Coplien, J. (2003). Understanding symmetry in object-oriented languages. *Journal of Object Technology*, 2(5):123–134. (Cited on page 16.)

Zhao, L. and Coplien, J. O. (2002). Symmetry in class and type hierarchy. In *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded applications*, pages 181–189. Australian Computer Society, Inc. (Cited on page 16.)

Ziadi, T., Frias, L., da Silva, M. A. A., and Ziane, M. (2012). Feature identification from the source code of product variants. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, pages 417–422. IEEE. (Cited on page 21.)

Ziadi, T., Hélouët, L., and Jézéquel, J.-M. (2003). Towards a uml profile for software product lines. *PFE*, 3:129–139. (Cited on pages 18 and 19.)

Ziadi, T., Henard, C., Papadakis, M., Ziane, M., and Le Traon, Y. (2014). Towards a language-independent approach for reverse-engineering of software product lines. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pages 1064–1071. ACM. (Cited on pages 22 and 101.)

Ziadi, T. and Jézéquel, J.-M. (2006). Product line engineering with the uml: deriving products. (Cited on page 18.)