Chapter 2
Variability Implementation and UML-Based Software Product Lines

Ana Paula Allian, Elisa Yumi Nakagawa, Jabier Martinez, Wesley Klewerton Guez Assunção, and Edson OliveiraJr

Abstract Variability makes it possible to easily change and adapt software systems for specific contexts in a preplanned manner. It has been considered in several research topics, including self-adaptive systems, large-scale enterprise systems, and system-of-systems, and was mainly consolidated by the Software Product Line (SPL) engineering. SPL manages a common platform for developing a family of products with reduced time to market, better quality, and lower cost. Variability in the SPL must be clearly identified, modeled, evaluated, and instantiated. Despite the advances in this field, managing the variability of systems is still challenging for building software-intensive product families. One difficulty is that the software architecture, the cornerstone of any design process, is usually defined with notations and languages lacking accurate forms to describe the variability concerns of software systems. Hence, in this chapter, we analyze approaches used for describing software variability in SPL, paying special attention to the architecture.

A. P. Allian (✉) · E. Y. Nakagawa
Department of Computer Systems, University of São Paulo, São Carlos, Brazil
e-mail: ana.allian@usp.br; elisa@icmc.usp.br

J. Martinez
Tecnalia, Basque Research and Technology Alliance, Derio, Spain
e-mail: jabier.martinez@tecnalia.com

W. K. G. Assunção
ISSE, Johannes Kepler University Linz, Linz, Austria

OPUS, Pontifical Catholic University of Rio de Janeiro, Rio de Janeiro, Brazil
e-mail: wesley.assuncao@jku.at

E. OliveiraJr
Informatics Department, State University of Maringá, Maringá, Paraná, Brazil
e-mail: edson@din.uem.br

© Springer Nature Switzerland AG 2023

E. OliveiraJr (ed), *UML-Based Software Product Line Engineering with SMarty*,
https://doi.org/10.1007/978-3-031-18556-4_2

2.1 Introduction

Variability is a mechanism that allows a system, software asset, or development environment to be configured, customized, or changed for use in a specific domain in a preplanned manner [11]. This mechanism enables the mass customization of software products, which is the basis for creating Software Product Lines (SPL) [9, 26]. An advantage of managing variability is to bring flexibility when constructing families of software systems. For example, variability allows engineers to delay design decisions to later stages during the software development process by using mechanisms to define in which moment concrete design choices are bound to the software products (i.e., binding times) [10, 21, 49].

In a broader view, variability is described by three pieces of information: (i) *variation point* occurs in generic SPL artifacts, allowing the resolution of its variability in one or several locations through its associated variants; (ii) *variants*[1] represent software artifacts or possible elements, which can be chosen and resolved through a variation point; and (iii) *constraints* establish the relationships between two or more variants to resolve their respective points of variation [21, 29]. Once variability is described in an SPL, the configuration of valid products (i.e., configurations) is defined by resolving all variation points using available variants, taking into account existing constraints. As mentioned above, the resolution of variability can be in different binding times, as for example, at design time, compilation time, or runtime.

Variability involves all life cycle phases of a system development through the identification, modeling, derivation, and evaluation of variation points and variants to create specific products in an SPL [9]. Variability can then be associated with different levels of abstraction associated with different stages of software development [29, 39], for instance, at requirements level, architecture description, design documentation, source code, compiled code, linked code, or even executable code. In addition, variability can be initially identified through the concept of *feature* that can be defined as a characteristic of a system that is relevant and visible to end users [4, 26, 39]. Once the set of desired features for an SPL is established, the design of how features are configured to create products is done by defining variation points (i.e., where a feature can vary) and the variants (i.e., which are the alternatives that can be selected for a variation point).

This chapter is structured as follows. Section 2.2 presents the basics for how to model and implement variability. Given the relevance in the topic of the book, Sect. 2.3 focuses on UML-based SPL. Then, Sect. 2.4 presents a discussion, and Sect. 2.5 concludes this chapter with a summary and future directions.

[1] For clarification, sometimes the term “variants” is also used to refer to members of a system family (i.e., the whole product variant), as an alternative to the term “products.”

2.2 Implementing Variability

This section describes key concepts for implementing variability. Notably, a family of software products can be developed by properly specifying two dimensions of decomposition, known as *variability in problem space* and *variability in solution space*. Sections 2.2.1 and 2.2.2 present the variability in the problem space and solution space, respectively. Section 2.2.3 discusses existing tooling support.

2.2.1 Variability in the Problem Space

Variability in problem space refers to identifying features that may vary to express different products during domain analysis. Domain analysis assumes the existence of an SPL infrastructure to identify variations and features that may vary according to the needs of market segments or business goals [11]. Two main techniques to support domain analysis are:

- **Questionnaire-based analysis:** it is based on surveys, questions, and meetings with domain experts aiming to identify what can vary in SPL. Questions are used to support the identification of variability: *"what does it vary?"* is used for identifying variation points; *"why does it vary?"* and *"how does it vary?"* are used for identifying variants. Extending the questions proposed in [44], Milani et al. [36] developed a framework to identify and classify variation drivers in the business architecture layer based on w-questions (how, what, where, who, and when). The variability elicitation starts with identifying branching points (variation points) from the business process model. Each branching point is classified as a decision or as a variation point. When a variation point is identified, the analysis goes to identifying variation drivers (variants) with the support of w-questions. This activity must be repeated for each branching point of the process models; however, an architectural process must be given as input.
- **Scenario-based analysis:** it supports the identification of risks by analyzing anticipated changes to be made in the software systems and architectures; as a consequence, it results in suitable mitigation actions introduced before the software system is completely designed. Park et al. [41], Moon et al. [37], Pohl et al. [44], Weiss et al. [52], Meekel et al. [34], Tekinerdogan and Aksit [50], Kim et al. [27], and Bayer et al. [7, 8] proposed solutions to identify variability in Product Line Architectures (PLAs) through scenario-based analysis. The process starts with the detection of goals, and a scenario composed of one or more actions is created for each goal with purposeful interactions. Then, a feature is attached to scenario actions, which guide the identification of domain requirements, actors, and variability during the requirements elicitation analysis.

Several notations exist to capture the configuration space defined through the domain analysis, as the well-established ones listed below:

- **Feature modeling:** Feature models offer a formal way to describe variability by defining features and their dependencies [26]. The two main components of a feature model are features and relationships. Variability is described using a hierarchical decomposition of features connected between each other that yields a tree-like structure. Figure 2.1 presents an example of a feature model with eight features connected through different relationships [46].
- **Decision modeling:** Decision models focus on capturing variability in the form of a set of requirements and engineering decisions that are mandatory to describe and construct a product [17]. Additionally, these models enable to determine the extent of possible variation among desired products of the domain [49]. Decision models are usually represented using tables or spreadsheets. Figure 2.2 illustrates a decision model [49].
- **Orthogonal variability modeling:** Orthogonal variability models (OVMs) are based on a language that defines the variability of products using a cross-sectional view across all product line artifacts [44]. For example, OVM allows modeling variability by interrelating base models such as requirement models, design

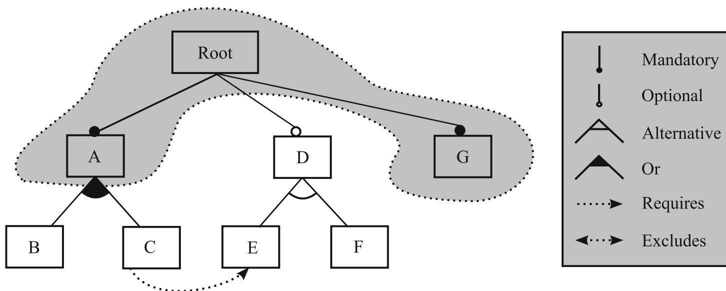

Fig. 2.1 Example of feature model, extracted from [46]

Example of a decision model

Name	Relevance	Description	Range	Selection	Constraints	Binding Times
Memory	System_Mem = True	Does the system have memory?	TRUE, FALSE	1		Compile Time
Memory_Size		The amount of memory the system has (KB)	0..100.000	1	Memory=TRUE => Memory_Size > 0	Installation, System Initialisation
Time_Measure-ment		How is time measurement done?	Hardware, Software	1		Compile Time

Fig. 2.2 Example of decision model, extracted from [49]

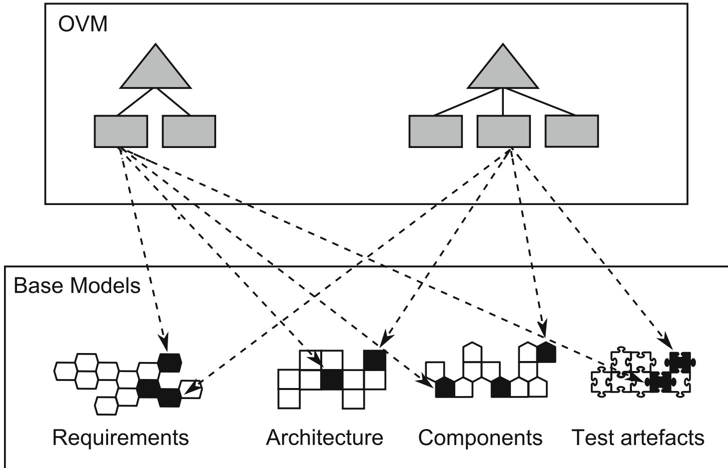

Fig. 2.3 Illustration of an orthogonal variability model, extracted from [46]

models, component models, and test models, as illustrated in Fig. 2.3. These dependencies among base models and OVM enable traceability to support the SPL engineering [35].

2.2.2 Variability in the Solution Space

Variability in the solution space describes the representation of features that are used to realize the problem space. Different approaches can describe variability in the solution space, mostly depending on the artifacts type. Existing approaches are classified mainly into two categories [48]: (i) *negative variability* considers one model for all products (also known as 150% models) with variant information determining which model elements are present in which products or features, and (ii) *positive variability* associates model fragments to features and composes them for a given feature configuration. Negative and positive variability can be used with any artifact during the SPL development, including hybrid approaches; however, in this chapter, we focus on the main approaches to deal with source code and design models.

Variability in source code. There are two main approaches to describe variability in the source code, namely, annotative or compositional:

- **Annotative approach**: This negative variability approach is based on annotative directives used to indicate pieces of code that should be compiled/included or not based on the value of variables [24]. The pieces of code can be marked at the granularity of a single line of code or to a whole file. Feature toggling is also

```
public final class DiagramFactory {
  private DiagramFactory() {
    super();
    diagramClasses.put(DiagramType.Class, UMLClassDiagram.class);
    //#if defined(USECASEDIAGRAM)
    diagramClasses.put(DiagramType.UseCase, UMLUseCaseDiagram.class);
    //#endif
    //#if defined(STATEDIAGRAM)
    diagramClasses.put(DiagramType.State, UMLStateDiagram.class);
    //#endif
    //#if defined(DEPLOYMENTDIAGRAM)
    diagramClasses.put(DiagramType.Deployment, UMLDeploymentDiagram.class);
    //#endif
    //#if defined(COLLABORATIONDIAGRAM)
    diagramClasses.put(DiagramType.Collaboration, UMLCollaborationDiagram.class);
    //#endif
    //#if defined(ACTIVITYDIAGRAM)
    diagramClasses.put(DiagramType.Activity, UMLActivityDiagram.class);
    //#endif
    //#if defined(SEQUENCEDIAGRAM)
    diagramClasses.put(DiagramType.Sequence, UMLSequenceDiagram.class);
    //#endif
  }
  ...
}
```

Fig. 2.4 Example of variability management with annotative directives (extracted from [38])

a used annotative approach [32] where there is no need for specific variability management libraries, as the annotations are based on the standard `if` clauses of the target programming language. Variability annotations have long been used in programming languages like C but can also be used in object-oriented languages, such as C++ [24] and Java [38]. Figure 2.4 presents a code snippet illustrating the use of preprocessor directives in ArgoUML-SPL [38]. The preprocessor directives `//#if` and `//#endif` are applied to indicate the beginning and end of each line of code belonging to a specific feature.

- **Compositional approach**: This positive variability approach is based on the addition of implementation fragments in specified places of a system [4]. The compositional approach enables SPL engineers to define separated reusable assets composed during derivation when features are selected. A widely known implementation is the superimposition approach [3]. The code snippet in Fig. 2.5 was extracted from Gruntfile,[2] which is another solution used to compose products implemented in JavaScript. The modules in the figure have a name and are followed by a brief description, if they are optional or not, and if they are replaceable by a stub version. During the building process, developers are able to define which modules they would like to exclude from their build.

[2] https://gruntjs.com/sample-gruntfile.

```
var modules = {
    'intro':         { 'description': 'Phaser UMD wrapper',
                     'optional': true, 'stub': false },
    'phaser':        { 'description': 'Phaser Globals',
                     'optional': false, 'stub': false },
    'geom':          { 'description': 'Geometry Classes',
                     'optional': false, 'stub': false },
    'core':          { 'description': 'Phaser Core',
                     'optional': false, 'stub': false },
    'input':         { 'description': 'Input Manager + Mouse and Touch
                     Support', 'optional': false, 'stub': false },
    'gamepad':       { 'description': 'Gamepad Input',
                     'optional': true, 'stub': false },
    'keyboard':      { 'description': 'Keyboard Input',
                     'optional': true, 'stub': false },
    'components':    { 'description': 'Game Object Components',
                     'optional': false, 'stub': false },
    'gameobjects':   { 'description': 'Core Game Objects',
                     'optional': false, 'stub': false },
    'bitmapdata':    { 'description': 'BitmapData Game Object',
                     'optional': true, 'stub': false },
    'graphics':      { 'description': 'Graphics and PIXI Mask Support',
                     'optional': true, 'stub': false },
    'rendertexture':{ 'description': 'RenderTexture Game Object',
                     'optional': true, 'stub': false },
    'text':          { 'description': 'Text Game Object (inc. Web
                     Support)', 'optional': true, 'stub': false },
    'bitmaptext':    { 'description': 'BitmapText Game Object',
                     'optional': true, 'stub': false },
    'retrofont':     { 'description': 'Retro Fonts Game Object',
                     'optional': true, 'stub': false },
    ...
};
```

Fig. 2.5 Example of variability management with compositional approach (extracted from [38])

Variability in design models. Most approaches to represent variability in design models are UML-based, ADL-based, and domain-specific [2]. Following, we present an overview of these approaches:

- **UML-based approaches:** They describe variability in software systems based on UML properties, such as stereotypes and inheritance. Different UML-based approaches have been developed to model variability in SPL. These approaches usually describe a metamodel where inheritance associations are represented as variants, and variability relationship properties are expressed as a Boolean formula.
- **ADL-based approaches:** They describe variability using code and formal representation, supporting the variability's evolution and automatic formal analysis. An example of an ADL-based approach is FX-MAN [14], a component model that incorporates variation points and composition mechanisms to handle variability in PLAs. EAST-ADL [28] is focused on a formal architecture

description and offers a complete feature model technique to represent variability in embedded system domains (automotive electronic systems). ADLARS [5], an architecture description language, captures variability information from feature models and links them to architecture structure using keyword descriptions. ArchStudio4 [16] is an open-source tool that implements an environment of integrated tools for modeling, visualizing, analyzing, and implementing software and systems architectures. For variability management, ArchStudio has a tool called product line selector with a user interface that enables graphically invoking the Selector, Pruner, and Version Pruner components. Hence, product architecture can be derived from a PLA automatically selected based on user-specified variable-value bindings. Finally, xLineMapper [15] is an Eclipse-based toolset to manage the relationships automatically (e.g., traceability, conformance) among product line features, architecture, and source code.

- **Domain-specific approaches:** They provide specific constructs and other techniques that complement UML and ADL notations. For example, Common Variability Language (CVL) models variability in architecture with metamodels combining representation of variability with its resolution [19]. The Variability Modeling Language (VML) represents variation points, features, constraints, and variants as entities in a textual language format. VML links the features in the feature model to architectural elements (e.g., components and compositions) by allowing features to be selected for specific variation points [31]. Alternatives to CVL are KCVL and BVR. KCVL[3] is bundled as a set of Eclipse plugins with a basic implementation of the OMG CVL with several additional features, namely, a textual editor for expressing variability abstraction models, variability realization models, and resolution models. Base Variability Resolution (BVR) [51] is a tool bundle to support SPL engineering and implements a language with advanced concepts for feature modeling, reuse, and realization of components in SPL. BVR covers design, implementation, and quality assurance to close the development cycle.

2.2.3 SPL Variability Tools

For nearly 30 years, industry and academia have proposed many variability tools to cope with the complexity of modeling variability in SPL [23]. Much research effort and investment have been already devoted to investigating, developing, and making available these tools [6, 30, 43].

Most tools represent variability with a graphical representation of features using *feature-oriented domain analysis* (FODA) [26] and its extensions. Examples of such tools are pure::variants, FeatureIDE, SPLOT, fmp, Clafer, GEARS, Fama, CVL, Hephaestus, CaptainFeature, PlugSPL, EASy-Producer, FW Profile, PREEVision,

[3] https://diverse-project.github.io/kcvl/.

Kconfig, and TypeChef [6]. Commercial tools like pure::variants and GEARS and open-source tools like FeatureIDE, CVL, and PLUM provide integration support to different tools to encompass more variability management functionalities. Some practitioners from the industry claimed that some tools fail to integrate with new technologies, including cloud and mobile applications. They stated the need for an independent application with graphical editors apart from Eclipse-based plugins.

The main concern to adopting feature models instead of UML on these tools is related to the graphical representation. UML suffers from low expressiveness to represent detailed variability concerns due to visualization to handle large variability models with multiple dependencies among them. Scalability has been a big challenge when considering millions of features (and variants) in the variability model. Ways to improve such scalability and usability of tools concerning adequate model visualization are still open research issues.

2.3 Overview of UML-Based SPL

Over the last three decades, the SPL community has witnessed a significant evolution from traditional feature-based model representation such as FODA [26] to more sophisticated languages to represent and configure variability to derive distinct products. UML is the most well-known and easier notation for modeling software systems [25] and has also been widely adopted to model variability in SPL [45]. UML can be easily extended in standardized ways; hence, generic UML models can be used to describe variability.

Some initiatives have been proposed based on the possibility of extending UML to describe variability. Ziadi et al. [53, 54] leveraged the idea that UML models can be considered reference models from which product models can be derived and created. Based on that, those authors proposed using UML extension mechanisms to specify product line variability in UML class diagrams and sequence diagrams. Basically, Ziadi et al. introduced two types of variability using stereotypes: (i) *optionality*, which indicates that a UML element is optional for the SPL members, represented by «`optional`», and (ii) *variation*, in which a variation point will be defined by an abstract class stereotyped «`variation`» and a set of subclasses stereotyped «`variant`». Figure 2.6 presents an example of a camera SPL with the proposed stereotypes.

Other UML-based approaches have been developed to model variability in SPL. Clauß [13] describes a metamodel where inheritance associations are represented as variants, and variability relationship properties are expressed as a Boolean formula (i.e., (component 1) XOR (component 2)). Pascual et al. [42] also made use of inheritance associations and included the use of cardinality properties to represent variability (i.e., Optional (0..*), alternatives (1..*)). Albassam and Gomaa [1, 18] introduced the Product Line UML-based Software (PLUS) method, an extension of UML to explicitly model variability and commonality in PLAs. VarSOAML [12] is another UML extension that allows modeling variability in services architectures.

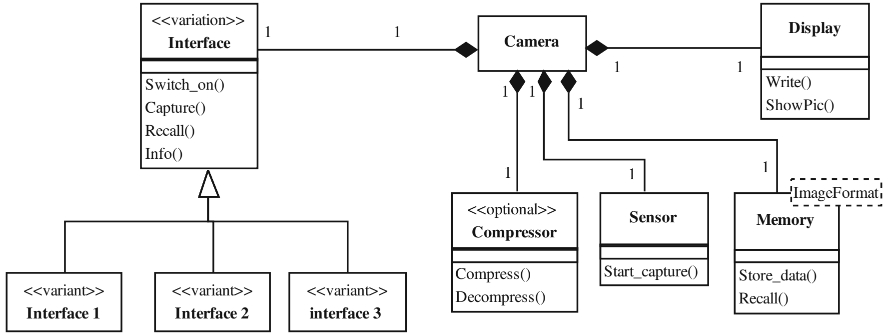

Fig. 2.6 Class diagram of a camera SPL with Ziadi et al. stereotypes, extracted from [53]

VxUML [22] models variability in architecture using many properties extracted from UML notation (i.e., stereotypes, diagrams, inheritance). Systems Modeling Language (SysML) is based on UML and can represent variability by combining SysML block diagrams with variants in the variability model. Ortiz et al. [40] present an example of a study that addresses variability with SysML. It uses thick lines to represent mandatory elements and dotted lines to represent variability elements. Guessi et al. [20] show a regular notation for SysML where optional blocks are represented with bold lines and fonts. This notation can be used to describe reference architectures and facilitate the identification of variable elements.

One of the main benefits of specifying variability in UML-based approaches is the usability for stakeholders. One disadvantage is the poor scalability of UML models when the number of variants increases [11] and the poor derivation process of variability models from UML notations to code. Besides that, maintaining the variability model embedded with UML constructs is hard when we add or remove variants and their corresponding constraints [11]. Therefore, expressing variability with UML must count on several UML diagrams, including use case, class, activity, components, and sequence diagrams. The Stereotype-based Management of Variability (SMarty), presented further in Chap. 4, explores various UML diagrams to provide a broader view of variability in SPL [39].

2.4 Discussion

This chapter summarizes the main approaches proposed to handle variability, considering its identification and representation. Most approaches for variability identification focus on SPL and were proposed in academic contexts. Domain analysis is a key process for eliciting reusable assets for SPL. Many domain analysis approaches depend on SPL infrastructure and experience from stakeholders for identifying variability and commonalities. In general, the list of features (different characteristics of a given system and relations among them) identified during the

domain analysis are further modeled with feature models. Domain analysis encompasses many activities to guide stakeholders during the identification of variability in SPL, and some activities might be helpful at different architectural levels, including enterprise architecture, software architecture, and reference architecture. However, these architectures are often designed without explicitly considering information about variability. A crucial cause of this problem is that the variability information exists as tacit knowledge in architects' minds, and it is rarely documented explicitly [33, 47]. Although UML-based approaches are easy to understand, their limitations in terms of visualization and scalability to describe large variability models are major drawbacks. There is still a lack of a unique variability modeling solution that various variability modeling tools could adopt. Apart from this, the inability of graphical representations and the fact that most domain-specific languages lack support for runtime concerns or dynamic variability are another drawbacks of the existing approaches. New notations are needed to handle the changes in variability models at a post-deployment time. For the future, we believe more effort must be put into interoperability concerns among the existing tools and languages to facilitate a smooth transition from variability models to implementation and from configuration to derivation process where the variability is realized.

2.5 Final Remarks

This chapter provided a general view of existing approaches to handle variability in SPL. Such approaches offer important advantages for software development, as efforts for introducing variability in the architecture are reduced, and the reusability of architectural elements is improved. UML-based approaches simplify the variability representation through stereotypes and inheritance mechanisms, whereas ADL-, CVL-, and VML-based approaches, combined with tools, provide better support for configuration and runtime derivation capabilities.

Variability in SPL lacks a support tool capable of fully handling variability from identification, representation, and evaluation of variability and also derivation of concrete architectures. To improve such approaches, more empirical evaluations with industrial partners are needed, aiming to demonstrate the importance of handling variability in SPL. Results from such evaluations would allow other researchers to adequate their approaches to industrial needs.

Acknowledgments The work is supported by the Brazilian funding agencies FAPESP (grants 2015/24144-7, 2016/05919-0, 2018/20882-1), CNPq (Grant 313245/2021-5), and Carlos Chagas Filho Foundation for Supporting Research in the State of Rio de Janeiro (FAPERJ), under the PDR-10 program, grant 202073/2020.

References

1. Albassam, E., Gomaa, H.: Applying software product lines to multiplatform video games. In: 3rd International Workshop on Games and Software Engineering: Engineering Computer Games to Enable Positive, Progressive Change (GAS), pp. 1–7. IEEE Computer Society, San Francisco (2013)
2. Allian, A.P., Capilla, R., Nakagawa, E.Y.: Observations from variability modelling approaches at the architecture level. In: Software Engineering for Variability Intensive Systems – Foundations and Applications, pp. 41–56. Auerbach Publications/Taylor & Francis, Milton Park (2019)
3. Apel, S., Kastner, C., Lengauer, C.: FEATUREHOUSE: Language-independent, automated software composition. In: Proceedings of the 31st International Conference on Software Engineering, ICSE '09, pp. 221–231. IEEE Computer Society, Washington (2009)
4. Apel, S., Batory, D., Kästner, C., Saake, G.: Feature-Oriented Software Product Lines. Springer, Berlin (2016)
5. Bashroush, R., Brown, T.J., Spence, I.T.A., Kilpatrick, P.: ADLARS: an architecture description language for software product lines. In: 29th Annual IEEE/NASA Software Engineering Workshop (SEW), pp. 163–173. IEEE Computer Society, Greenbelt (2005)
6. Bashroush, R., Garba, M., Rabiser, R., Groher, I., Botterweck, G.: CASE tool support for variability management in software product lines. ACM Comput. Surv. **50**(1), 14:1–14:45 (2017)
7. Bayer, J., Flege, O., Knauber, P., Laqua, R., Muthig, D., Schmid, K., Widen, T., DeBaud, J.: Pulse: a methodology to develop software product lines. In: Symposium on Software reusability (SSR), Los Angeles, pp. 122–131 (1999)
8. Bayer, J., Flege, O., Knauber, P., Laqua, R., Muthig, D., Schmid, K., Widen, T., DeBaud, J.M.: Pulse: a methodology to develop software product lines. In: Proceedings of the 1999 Symposium on Software Reusability (SSR), pp. 122–131. ACM, Los Angeles (1999)
9. Bosch, J., Capilla, R., Hilliard, R.: Trends in systems and software variability. IEEE Softw. **32**(3), 44–51 (2015)
10. Capilla, R., Bosch, J.: Binding Time and Evolution, pp. 57–73. Springer, Berlin (2013). https://doi.org/10.1007/978-3-642-36583-6_4
11. Capilla, R., Bosch, J., Kang, K.C.: Systems and Software Variability Management: Concepts, Tools and Experiences. Springer, Berlin (2013)
12. Chakir, B., Fredj, M., Nassar, M.: A model driven method for promoting reuse in SOA-solutions by managing variability. Computing Research Repository (CoRR), abs/1207.2742 (2012)
13. Clauß, M.: Modeling variability with UML. In: 3rd International Conference on Generative and Component-Based Software Engineering (GCSE), pp. 1–5. Springer, Berlin (2001)
14. Cola, S.D., Tran, C.M., Lau, K., Qian, C., Schulze, M.: A component model for defining software product families with explicit variation points. In: 19th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE), pp. 79–84. IEEE Computer Society, Venice (2016)
15. Cu, C., Ye, X., Zheng, Y.: Xlinemapper: a product line feature-architecture-implementation mapping toolset. In: 41st International Conference on Software Engineering: Companion Proceedings, ICSE '19, pp. 87–90. IEEE Press, Piscataway (2019). https://doi.org/10.1109/ICSE-Companion.2019.00045
16. Dashofy, E., Asuncion, H., Hendrickson, S., Suryanarayana, G., Georgas, J., Taylor, R.: Archstudio 4: an architecture-based meta-modeling environment. In: 29th International Conference on Software Engineering (ICSE'07 Companion), pp. 67–68. IEEE, Piscataway (2007)
17. Dhungana, D., Grünbacher, P.: Understanding decision-oriented variability modelling. In: Software Product Line Conference – SPLC (2), pp. 233–242 (2008)
18. Gomaa, H.: Designing Software Product Lines with UML – from Use Cases to Pattern-Based Software Architectures. ACM, New York (2005)

19. Gonzalez-Huerta, J., Abrahão, S., Insfrán, E., Lewis, B.: Automatic derivation of AADL product architectures in software product line development. In: 1st International Workshop on Architecture Centric Virtual Integration and 17th International Conference on Model Driven Engineering Languages and Systems (ACVI/MoDELS), pp. 1–10. CEUR-WS.org, Valencia (2014)
20. Guessi, M., Oquendo, F., Nakagawa, E.Y.: Variability viewpoint to describe reference architectures. In: Working IEEE/IFIP Conference on Software Architecture (WICSA), pp. 14:1–14:6. ACM, Sydney (2014)
21. Halmans, G., Pohl, K.: Communicating the variability of a software-product family to customers. Softw. Syst. Model. **2**(1), 15–36 (2003)
22. He, X., Fu, Y., Sun, C., Ma, Z., Shao, W.: Towards model-driven variability-based flexible service compositions. In: 39th IEEE Annual Computer Software and Applications Conference, COMPSAC, pp. 298–303. IEEE Computer Society, Taichung (2015)
23. Horcas, J.M., Pinto, M., Fuentes, L.: Software product line engineering: a practical experience. In: Proceedings of the 23rd International Systems and Software Product Line Conference, SPLC 2019, Paris, September 9–13, 2019, vol. A, pp. 25:1–25:13. ACM, New York (2019)
24. Hu, Y., Merlo, E., Dagenais, M., Lague, B.: C/c++ conditional compilation analysis using symbolic execution. In: 30th International Conference on Software Maintenance, ICSM ’00. ACM, New York (2000)
25. Júnior, E., Farias, K., Silva, B.: A Survey on the Use of UML in the Brazilian Industry, pp. 275–284. Association for Computing Machinery, New York (2021). https://doi.org/10.1145/3474624.3474632
26. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.: Feature-oriented domain analysis (FODA) feasibility study. Technical Report. CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh (1990). http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=11231
27. Kim, M., Yang, H., Park, S.: A domain analysis method for software product lines based on scenarios, goals and features. In: 10th Asia-Pacific Software Engineering Conference (APSEC, pp. 126–135. IEEE Computer Society, Chiang Mai (2003)
28. Leitner, A., Mader, R., Kreiner, C., Steger, C., Weiß, R.: A development methodology for variant-rich automotive software architectures. Elektrotechnik und Informationstechnik **128**(6), 222–227 (2011)
29. Linden, F.J.V.D., Schmid, K., Rommes, E.: Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering, vol. 20. Springer, New York (2007)
30. Lisboa, L.B., Garcia, V.C., Lucrédio, D., de Almeida, E.S., de Lemos Meira, S.R., de Mattos Fortes, R.P.: A systematic review of domain analysis tools. Inf. Softw. Technol. **52**(1), 1–13 (2010)
31. Loughran, N., Sánchez, P., Garcia, A., Fuentes, L.: Language support for managing variability in architectural models. In: 7th International Symposium on Software Composition (SC), pp. 36–51. Springer, Budapest (2008)
32. Mahdavi-Hezaveh, R., Dremann, J., Williams, L.: Software development with feature toggles: practices used by practitioners. Empir. Softw. Eng. **26**(1) (2021)
33. Martínez-Fernández, S., Ayala, C.P., Franch, X., Marques, H.M.: Benefits and drawbacks of software reference architectures: a case study. Inf. Softw. Technol. **88**, 37–52 (2017)
34. Meekel, J., Horton, T.B., Mellone, C.: Architecting for domain variability. In: 2nd International ESPRIT ARES Workshop on Development and Evolution of Software Architectures for Product Families, pp. 205–213. Springer, Berlin (1998)
35. Metzger, A., Pohl, K., Heymans, P., Schobbens, P.Y., Saval, G.: Disambiguating the documentation of variability in software product lines: a separation of concerns, formalization and automated analysis. In: 15th IEEE International Requirements Engineering Conference (RE 2007), pp. 243–253. IEEE, Piscataway (2007)
36. Milani, F., Dumas, M., Matulevicius, R.: Identifying and classifying variations in business processes. In: Enterprise, Business-Process and Information Systems Modeling – 13th International Conference, BPMDS 2012, 17th International Conference, EMMSAD 2012, and 5th EuroSymposium, held at CAiSE 2012, pp. 136–150. Springer, Gdańsk (2012)

37. Moon, M., Yeom, K., Chae, H.S.: An approach to developing domain requirements as a core asset based on commonality and variability analysis in a product line. IEEE Trans. Softw. Eng. **31**(7), 551–569 (2005)
38. Moreira, R.A.F., Assunção, W.K., Martinez, J., Figueiredo, E.: Open-source software product line extraction processes: the argoUML-SPL and phaser cases. Empir. Softw. Eng. **27**(4), 1–35 (2022)
39. OliveiraJr, E., Gimenes, I.M.S., Maldonado, J.C., Masiero, P.C., Barroca, L.: Systematic evaluation of software product line architectures. J. Univer. Comput. Sci. **19**(1), 25–52 (2013)
40. Ortiz, F.J., Pastor, J.A., Alonso, D., Losilla, F., de Jódar, E.: A reference architecture for managing variability among teleoperated service robots. In: 2nd International Conference on Informatics in Control, Automation and Robotics (ICINCO), pp. 322–328. INSTICC Press, Barcelona (2005)
41. Park, S., Kim, M., Sugumaran, V.: A scenario, goal and feature-oriented domain analysis approach for developing software product lines. Ind. Manag. Data Syst. **104**(4), 296–308 (2004)
42. Pascual, G.G., Pinto, M., Fuentes, L.: Automatic analysis of software architectures with variability. In: 13th International Conference on Software Reuse (ICSR), pp. 127–143. Springer, Pisa (2013)
43. Pereira, J.A., Constantino, K., Figueiredo, E.: A systematic literature review of software product line management tools. In: 14th International Conference on Software Reuse for Dynamic Systems in the Cloud and Beyond (ICSR), pp. 73–89. Springer International Publishing, Miami (2015)
44. Pohl, K., Böckle, G., van der Linden, F.: Software product line engineering: foundations, principles, and techniques, Springer, Berlin (2005)
45. Raatikainen, M., Tiihonen, J., Männistö, T.: Software product lines and variability modeling: a tertiary study. J. Syst. Softw. **149**, 485–510 (2019)
46. Roos-Frantz, F., Benavides, D., Ruiz-Cortés, A., Heuer, A., Lauenroth, K.: Quality-aware analysis in product line engineering with the orthogonal variability model. Softw. Qual. J. **20**(3–4), 519–565 (2011). https://doi.org/10.1007/s11219-011-9156-5
47. Rurua, N., Eshuis, R., Razavian, M.: Representing variability in enterprise architecture. Bus. Inf. Syst. Eng. **61**(2), 215–227, (2019)
48. Schaefer, I.: Variability modelling for model-driven development of software product lines. In: 4th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS), pp. 85–92 (2010)
49. Schmid, K., John, I.: A customizable approach to full lifecycle variability management. Sci. Comput. Program. **53**(3), 259–284 (2004)
50. Tekinerdogan, B., Aksit, M.: Managing variability in product line scoping using design space models. In: Journal of The American Chemical Society, pp. 1–8. Elsevier, Groningen (2003)
51. Vasilevskiy, A., Haugen, Ø., Chauvel, F., Johansen, M.F., Shimbara, D.: The BVR tool bundle to support product line engineering. In: Proceedings of the 19th International Conference on Software Product Line, pp. 380–384 (2015)
52. Weiss, D.M., Lai, C.T.R.: Software Product-Line Engineering: A Family-Based Software Development Process. Addison-Wesley Longman Publishing Co. Inc., Boston (1999)
53. Ziadi, T., Hélouët, L., Jézéquel, J.: Towards a UML profile for software product lines. In: Software Product-Family Engineering, 5th International Workshop, PFE 2003, Siena, November 4–6, 2003, Revised Papers. Lecture Notes in Computer Science, vol. 3014, pp. 129–139. Springer, Berlin (2003)
54. Ziadi, T., Jézéquel, J.: Software product line engineering with the UML: deriving products. In: Software Product Lines – Research Issues in Engineering and Management, pp. 557–588. Springer, Berlin (2006)