A Proposal for Organizing Source Code Variability in the Git Version Control System

Junior Cupe Casquina
junior.cupe@ic.unicamp.br
University of Campinas
Campinas, SP, Brazil

Leonardo Montecchi
leonardo@ic.unicamp.br
University of Campinas
Campinas, SP, Brazil

ABSTRACT

Often, either to expand the target market or to satisfy specific new requirements, software systems inside a company are cloned, refactored, and customized, generating new derived software systems. Although this is a practical solution, it is not effective in the long-term because of the high maintenance costs when maintaining each of these derived software systems. Software product lines (SPLs) were proposed to reduce these costs; however, the lack of integration between variability realization mechanisms and version control systems reduces its attractiveness in the software development industry, especially in small and medium software companies. In this paper we propose an approach to integrate the conditional compilation mechanism used to implement the SPL variabilities and the Git version control system used to manage software versions in order to increase the attractiveness of the SPLs in the industry. The proposed solution also could be seen as a method to manage software system families' evolution in space and time.

CCS CONCEPTS

• **Software and its engineering → Software product lines**.

KEYWORDS

Software product lines, Conditional Compilation, SPL, Git, VarCS

ACM Reference Format:
Junior Cupe Casquina and Leonardo Montecchi. 2021. A Proposal for Organizing Source Code Variability in the Git Version Control System. In *25th ACM International Systems and Software Product Line Conference - Volume A (SPLC '21), September 6–11, 2021, Leicester, United Kingdom.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3461001.3471141

1 INTRODUCTION

Software product line engineering (SPLE) provides notions for developing a software product line (SPL) across the entire development cycle through its common and variable features [21]. An SPL is a family of software systems (SSs) that share common features [63] and in which a unique combination of features defines a specific software [44]. SPLE aims to automatically construct specific SSs after selecting features that have to be included in them [69], and it is meant as an alternative to the clone-and-own (CaO) practice [32], whereby a new variant of a software system is built by copying and adapting an existing variant. The use of SPLE in a corporation reduces the cumulative costs in the long-term software development [52], and it is therefore an attractive approach to optimize costs. Although the initial cost of developing an SPL is high, there is a breakpoint—three or four SSs to be developed—after which it brings benefits to the company [81].

Variability models—e.g., the feature modeling [22], the decision modeling [70], among others [6, 8, 9, 28, 35, 37]—represent the common and variable features of products in an SPL [10]. These models guide the SPL development but do not implement its source code variability. To implement this variability, developers use variability realization mechanisms (VRMs)—e.g., the conditional compilation (CC) mechanism [41], feature-oriented programming [66], among others. Since the SPL topic is not new, at present, we can find diverse SPLs either in the industry—e.g., the ArgoUML SPL [20], E-Phenology Collector SPL [83]—or in the academy—e.g., the Chess SPL [84], Robocode SPL [51]. Organized catalogs have appreased over time; for instance, the LabSoft catalog [46] exhibits thirty-eight SPLs; the ESPLA catalog [50] exhibits case studies on extractive SPL adoption; and the Ferreira et al. dataset [31] exhibits configurable SSs with their test suite.

Most of the SPL case studies focus on how to manage the variability given by different versions of a software, and not on its evolution over time. The "variability in space" is understood as the concept that an artifact (or part of it) can appear in different shapes at the same time [60]. In the SPLs, this kind of variability is commonly represented in the traditional variability models—e.g., the feature model (FM). The FM is a hierarchical composition of related features where each feature describes some functionalities of the product line [76]. Conversely, the "variability in time" is commonly managed by version control systems (VCSs) and is understood as different versions of an artifact (or part of it) being valid at different times [60]. In SPLs, this kind of variability is represented only in specific variability models—e.g., the hyper feature model [72], which basically is a FM that adds versions to the features.

VCSs are the de facto mechanisms for, but not limited to, managing "variability in time". Git, for example, is a popular VCS [17] that facilitates the development of software projects, from small projects with only a few programmers to large projects with hundreds of programmers. Nowadays, most software projects rely on Git for version control, and the fact that the VRMs are not integrated with it (or in general with a VCS) reduces their attractiveness in the software industry. Still, effectively managing variability in space and time is among the main challenges of developing and maintaining large-scale yet long-living software-intensive systems [3].

SPLC '21, September 6–11, 2021, Leicester, United Kingdom

ACM ISBN 978-1-4503-8469-8/21/09…$15.00
https://doi.org/10.1145/3461001.3471141

In this work, we propose a new approach to integrate variability management and the Git VCS. The objective is to organize variability-enabled source code in Git, and at the same time abstract the variability details from the developer. In particular, in this first proposal we focus on conditional compilation. The approach organizes the code into branches, based on features, and then periodically propagates the changes (Git commits) to the other branches resolving product and down conflicts (see Section 4.3.3). The approach combines the advantages of applying VCSs (i.e., variability in time) and the CC mechanism (i.e., variability in space). The remainder of this paper proceeds as follows. Section 2 presents the essential concepts to understand the proposal. Section 3 overviews existing VRMs. Section 4 details our proposal. Finally, we present the related work and ongoing activities in Section 5.

2 BACKGROUND

2.1 Software Product Lines

A Software Product Line (SPL) is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment and are developed from a common set of core assets in a prescribed way [18]. For the development of SPLs, currently, SPLE establishes two complementary development processes [56, 65]: Domain Engineering (DE) and Application Engineering (AE). DE is the process of analyzing the domain of a product line and developing reusable artifacts [36]. In contrast, AE has the goal of developing a specific product for the needs of a particular customer [4]. Besides, each of these two processes can be observed in two different spaces: problem space and solution space. The problem space refers to systems' specifications established during the domain analysis and requirements engineering phases [7], commonly from the stakeholder's perspective [4, 39]. In contrast, the solution space refers to the concrete systems created during the architecture, design, and implementation phases [7], usually from the developer's perspective [4].

An essential output of the DE phase is the variability model of the SPL, which is typically represented as a Feature Model (FM). An FM essentially represents the software features that are available in an SPL [16]. An FM is a tree-like structure and consists of: i) features; ii) relations between a parent feature and its child features; and iii) cross-tree constraints that are typically inclusion or exclusion statements [5]. A feature is a distinctive characteristic of a SS that may refer to a requirement, a software-architecture component, or source code pieces [67]. A feature can be classified as abstract or concrete; an abstract feature is commonly used only to organize concrete features, and it does not appear in concrete product configurations.

2.2 Version Control Systems

A version control system (VCS), also known as a revision control system or source control system [68], is a specialized type of database used by developers to store the different versions of the source code that they are developing [61]. VCS can be categorized as centralized or distributed. Centralized VCSs are characterized by a single master repository accessed by all the developers to "check out" and "check in" version commits [73]. Distributed VCSs are characterized by complete local repositories to each developer, allowing developers to exchange and integrate code changes in a peer-to-peer fashion [34]. VCSs usually manage the "variability in time" of a software system, either storing only changes—like Darcs [25]—or storing a complete copy of a modified file—like Git. Git is currently the most popular VCS in the software industry [62] with various tools that support its use, such as the GitHub and Bitbucket web code-sharing platforms.

3 VARIABILITY REALIZATION MECHANISMS

3.1 Conditional Compilation

Conditional Compilation (CC) is a simple mechanism to implement compile-time variability and one of the most popular VRMs [42, 54, 77] in the SPL industry. The NASA's flight control software, HP's product line of printer firmware and the Linux kernel are some of the SPL projects that use this mechanism to implement its variability [42]. CC utilizes special directives to manage the variability and a lexical preprocessor to processes them according to a product configuration. This kind of annotations can be used with anything that is in textual form—e.g., annotations in dependability models [14]—, but with the drawback that it disrupts the language/format involved. To avoid this issue, the preprocessor directives are typically embedded in the comments of the host language. The C preprocessor [40], for example, implements this VRM, being one of the most used approaches in open-source and industry projects [45]. Other implementations include the Pascal's preprocessor [74], Munge preprocessor [77], Antenna preprocessor [77], pure::variants's preprocessor [11], and Gears's preprocessor [42].

The C preprocessor (CPP) uses simple directives such as `#if`, `#elif`, `#else`, and `#endif` to add variability inside source code written in C, allowing developers to create conditional statements like "if a specific feature is selected, include this specific source code." However, because these directives are not part of the syntax of the host programming language, development environments often report errors in conditional code. For example, multiple declarations of the same variable in different parts of a CC-controlled code can trigger errors in the development environment. Additionally, the fact of having the source code all the variants mixed together limits the readability of the code, thus increasing the difficulty of writing and maintaining the project. Directives defined by Munge (such as `if[tag]` , `else[tag]`, `ifnot[tag]`, and `end[tag]`), and Antenna (such as `#if expression`, `#elif expression`,`#else`, `#endif` and `#condition expression`) in Java comments also limit the readability of the code and, in some cases, in how to write the source code. Mainstream programming languages could integrate these directives in some standard (it is the case of C); however, for the moment, very few languages support this VRM off-the-shelf.

3.2 Object-Oriented Programming

In object-oriented programming (OOP), SSs are organized as cooperative collections of objects, each being an instance of some class, and whose classes are members of a hierarchy of classes connected via inheritance and usage relationships [12]. The distinctive characteristics of OOP like polymorphism, inheritance, and software design patterns (SDPs) allow creating the variability in the source code of an SPL. In [33], for example, we can find descriptions of how

SDPs can be used to implement the variability described in an FM. With OOP mechanisms, developers can also create frameworks or libraries to manage the variability in the source code of an SS or SPL. The COSMOS* model [23, 59] is an example of applying OOP mechanisms to define an implementation model for the specification and development of SPLs. The COSMOS* model is also a component implementation model, and it can also be classified as VRM focused on components (See Section 3.3). The COSMOS* model was used to create one of the implementations of the MobileMedia SPL [78].

3.3 Component-Oriented Programming

Component-Oriented Programming (COP) is focused on developing software by assembling components, while OOP emphasizes classes and objects [85]. The COP separates concerns into entities called components [64]. A component is not an object, but can provide the resources to instantiate objects [13] if it supports OOP. Components are reusable and could be seen as black boxes [53], that is, they describe what can be done, rather than how it is done. Components are characterized by their required (i.e., input) interfaces and provided (i.e., output) interfaces, and they can be connected though these interfaces. As a result, components can be put together in various configurations to form a SS [15]. Being able to interchange components and component configurations creates the variability required for the different SSs of an SPL.

OSGi [19] and Docker [27] are technologies that can be adapted to implement SPLs using this VRM. The OSGi platform, for example, defines modules that can be seen as components [2], while the Docker virtualization technology defines containers that could also be seen as components. In both cases, it depends on what we put inside the component (e.g., Microservices) and on how it defines the protocol of its interfaces of communication (e.g., REST). Additionally, both technologies have a kind of orchestrator to manage its kind of component that is useful for this VRM—e.g., in the context of Docker, container orchestrators allow to define how to select, deploy, track and dynamically manage the configuration of multi-container packaged applications

3.4 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) is a technology for dealing explicitly with the separation of concerns [82]. AOP is a programming paradigm that supports the modular implementation of crosscutting concerns [75]. For instance, considering the source that implements security policies across the different modules of a SS, AOP allows to factor these policies in a single element called aspect. Aspects change an existing application without modifying its source code [55]. Having a base application implemented in an object-oriented language (e.g., Java), developers can apply aspects to extend the application. AspectJ is an aspect-oriented extension to Java [43]. AspectJ defines an annotation called Pointcut to specify where the aspect modules can manipulate the base source code. Aspect modules may alter the control flow, overwrite methods, or add source code before or after a specific method. AspectJ code is compiled into standard Java bytecode.

3.5 Feature-Oriented Programming

Feature-oriented programming (FOP) is a paradigm to develop SPLs and a specialized form of generative programming [55]. Generative programming is a computing paradigm allowing the automatic creation of entire software families using the configuration of primary components [80]. FOP encapsulates features into separate feature modules to achieve separation of concerns, and through the integration of these feature modules, it generates a variety of software products [80]. Unlike COP, this integrates feature modules instead of linking components: when features are merged, consistent artifacts that define a program are synthesized [79]. Additionally, in FOP, the composition order matters because altering the order can alter the resulting product variant [55]. FeatureHouse [30] and AHEAD [1] are some of the available tools that support FOP.

3.6 Delta-Oriented Programming

Delta oriented programming (DOP) is a compositional approach to implement SPLs [24]. The delta modules comprise modifications of an object-oriented program similar to the aspects in the aspect-oriented programming. DOP allows generating a product by adding a set of actions encapsulated in a delta module (e.g., add a new method to the class X) to a core module. DeltaJ is a Java-like language that supports DOP by organizing classes and interfaces in delta modules [26]. There are two kinds of delta modules: core and delta. Core modules are collections of classes, while delta modules are a set of operations that allow adding, modifying, or removing classes or methods declared in other modules. Thus, a delta module allows adding classes, methods, and fields, removing classes, methods, and fields, changing superclasses or constructors, and finally renaming methods and fields. In DOP, a SS is assembled using a single core module and several delta modules, which are selected based on the features to be included.

4 THE PROPOSED APPROACH FOR ORGANIZING THE SPL SOURCE CODE

4.1 Motivation

Like SSs evolve, SPLs also evolve, and thus the SPL source code changes over time. Changes to the SPL source code, even minor ones, can affect multiple features and products of an SPL. Understanding the impact of change implies dealing with a high number of logical expressions, depending on how the SPL and its variability are implemented. Over time the number of variants to handle increases, and so does the number of revisions, thus becoming a cognitively complex task [48, 57]—this complexity is increased drastically when altering the variability model.

Thus, in the SPL development cycle, there is a need for a practical approach that can simplify the system evolution over time. On the other hand, developers should be able to focus on the specific or the generic source code separately as needed. For example, they should be able to focus on fixing a bug in a feature without the distraction of code involving other features. Besides, developers should be able to generate a product with all the latest modifications after committing changes. However, in current approaches, developers are somehow forced to work with code “extremely polluted” with

variability mechanisms, where a small unintended change can affect many features without the developer's knowledge.

4.2 The Feature Branching Graph

We notice that the Git branch structure could be seen as a tree where each node is a branch, and each relationship between parent and child is used to identify the branch taken as a base to create a new branch. We called this structure a *Feature Branching Graph*. The Feature Branching Graph (FBG) is used to organize the generic and specific source code of an SPL, with the more specific parts of the source code reside in the nodes with greater depth. The FBG is created from the FM of the SPL, following three main rules. We create: (i) a variability branch for each abstract feature that is not a feature leaf, (ii) a feature branch for each concrete feature that is a feature leaf, and (iii) two branches for each concrete feature that is not a feature leaf, being one branch for the feature and one for storing the variability of his child features.

By structuring the source code correctly in the FBG in Git, developers can focus on different variability aspects (feature or groups of features) without affecting each other's work, because branches are focused on features. Figure 1 shows the FBG created using the Elevator SPL's partial FM. As we can observe, the hierarchy in the FM is maintained in the FBG. When implementing the FBG in Git, the structure of the FBG is used to know which branches should be taken as base to create the other Git branches. For example, after creating the 'ElevatorSPL' branch from the master branch, it is created the 'ElevatorSPL Variability' branch from the 'ElevatorSPL' branch, and so on for the other branches of the graph (See Figure 1). This way, the FBG in Git can maintain commits related to the common parts in higher branches in the hierarchy, and commits related to the specific parts in lower branches in the hierarchy.

4.3 SPL Development Process

4.3.1 Workflow Overview. Figure 2 shows a detailed review of this process. When having existing artifacts, we have to collect them (step 1) and then manually or automatically analyze them (step 2) to find patterns and then design the FM (step 3). In the case there are no existing artifacts to be considered, we can skip steps 1 and 2. Having already the FM, we can define the valid product configurations (step 4) and then the corresponding FBG in Git (step 5). To use our approach correctly, developers have to follow a series of activities to implement the SPL. Of course, our objective is to create a tool that makes this as transparent as possible to them. First, developers choose what mechanisms to control the variability will be used in SPL development. For the moment, our proposal focuses on conditional compilation, being the most general and one of the most practical. Second, developers create the common parts of the SPL (e.g., the essential codebase), and store and organize the most common codebase into the FBG by depth. For example, the essential codebase common to all the features is added to the root branch. Then, these changes have to be propagated to all the derived branches (step 6).

When implementing a feature, commits related to common codebases must be stored in branches with low depth, while commits related to specific features or groups of features must be stored in branches with higher depth (step 7). As before, all the derived branches must be updated with these new changes. For this approach to work, the FBG must be maintained in a state that we call *pure* state, which implies that there are no conflict downwards the tree, or among the different features of a product. This is achieved by adding statements that have as function managing the variability (step 8), for example, Git commits with conditional compilation directives added inside the variability branches. We call *product conflict* a conflict that emerges when merging branches representing different features in the same product configuration. A *down conflict* emerges instead when propagating changes (i.e., commits) to more specific branches (child branches), following the structure of the FBG.

4.3.2 Generation of products. To generate a product with the proposed approach, a new temporary branch is generated from the root branch of the FBG (e.g., the ElevatorSPL branch in Figure 1). Then, this branch is merged with the specific branches related to the features selected in the product's configuration. The next step is to create another temporary branch from the resulting branch, to separate the merged branch from the branch with the resolution of the variability in order to have a backup of the merged branch. Finally, in this new created branch, we have to resolve the variability, depending on the approach used to implement the variability realization. At the end of this process, we will have a Git branch with the source code of the desired product according to the selected features of the product's configuration. However, when some merge generates conflicts, it indicates that the FBG is not in a pure state and needs changes to be pure.

4.3.3 Resolving down conflicts and product conflicts. When adding some commit to a parent branch, the approach proposes to spread the commit to the child branches (Git rebase command) respecting the hierarchical order. However, this activity can generate a Git rebase conflict with some of its child branches that in this paper we called a down conflict. When the conflict is detected, developers must resolve the conflict in that specific variability child branch or in the closest higher variability branch as possible, and then spread the now new commits to the child branches. On the other hand, the product conflicts emerge when verifying that a specific software product can be generated using its product configuration. In this case, the conflict has to be resolved in the closest higher variability branch and then spread the commit to his child branches, resolving also the down conflicts when spreading the commit.

We believe that, to a certain extent, the resolution of conflicts can be automated and possibly without human intervention. When no automated solution is possible, the tool should at least propose different candidate solutions, from which the developer could select the most appropriate one. In our first investigation of this proposal, conflicts will be solved by the CC mechanism. More in general, we foresee a component that takes care of solving conflicts in the code base; a specific implementation of this component would be needed to support other VRMs. When merging feature branches, this component would be in charge to resolve the down conflicts and the product conflicts in the FBG.

4.3.4 Discussion. Although our approach focuses on organizing branches to reduce the pollution generated by some variability mechanisms, there may be cases where the FBG is contaminated

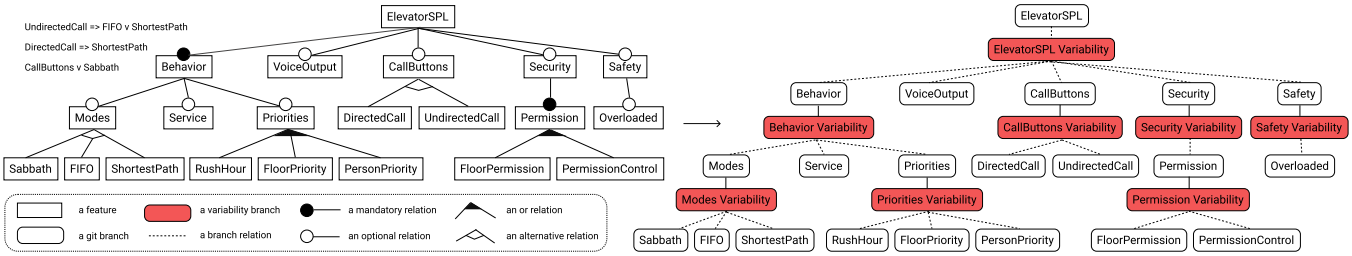

Figure 1: The FBG generated from the Elevator SPL's partial FM

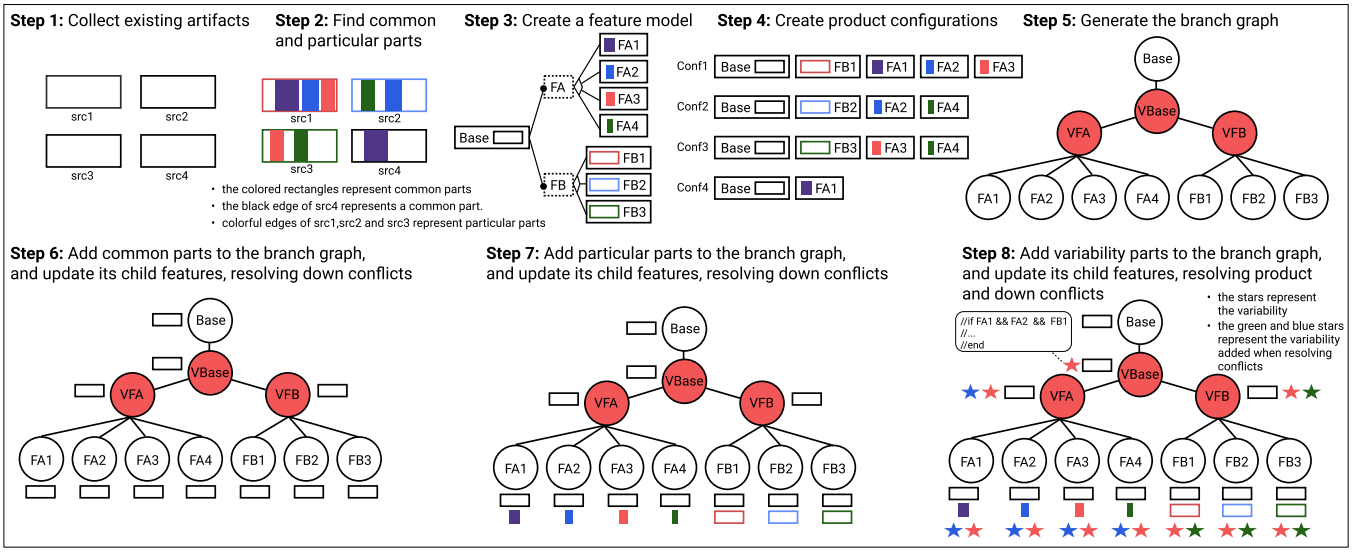

Figure 2: A review of the SPL development using the proposed approach

by them. To resolve this problem, we have been testing a different version of the approach with delay spreading, so commits related to conflict resolution only are stored in variability branches without polluted the feature branches. Finally, the scope of applicability of the proposed approach will be explored in future work.

5 RELATED WORK AND ONGOING ACTIVITIES

The grown-and-prune model [29, 58] and the PLE-Flow model [38] propose branching models to manage SPLs similarly to our approach. Each of these models has its advantages and disadvantages. Our approach focuses on the features of the SPL and not on its products, as the grown-and-prune model does. So products generated using our approach are stored in temporary branches. Besides, if the generated product has a bug, it is fixed in the feature branches and not in the product branch. On the other hand, our approach has a similar complexity as the PLE-Flow model; however, we are developing tools to reduce this complexity.

Variation control systems (VarCSs)—e.g., ECCO [47] and SuperMod [71]—allow working on one or multiple variants by providing views (or projections) that filter irrelevant details of configurable artifacts to facilitate their comprehension and lower the cognitive complexity when editing the variants [48]. However, current VarCSs have and depend on a particular restrictive style [57], so that it is not attractive for developers. On the other hand, our approach is less stringent because it is based on Git, a well-known tool in the software industry.

We are currently working on the concrete realization of this proposal, by defining the activities that should be performed by a supporting tool, and by developing a command-line prototype of such tool (new VarCS) as a layer on top of Git. Therefore, all the recurring mechanical activities, like propagating the commits or resolving product conflicts and down conflicts would be automatically handled by the tool. Regarding the management of conflicts, the tool must at least generate and list recommended solutions. Finally, the tool should also be able to create a product from a valid product configuration, and manage the graph branch on top of Git.

Following the VarCS characteristics proposed in [49], our new VarCS would be categorized as: "boolean" for the *entity* dimension; "variability model" as *constraints*; "text" and "files and folders" for the kind of *variable artifacts*; "per feature" for *revisions*; "database" as *internal storage*; "annotative" as *internal variation points*; "materialized" as *external type*; "fixed" as *external state*; and "distributed" as *collaboration*. We plan to validate our approach using one of the existing SPLs whose source code is publicly available, for example the ArgoUML SPL [20].

ACKNOWLEDGMENTS

This work is partially supported by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), and by the Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP) under Grant No. 2019/02144-6.

REFERENCES

[1] AHEAD. 2021. AHEAD Tool Suite. https://www.cs.utexas.edu/~schwartz/ATS/fopdocs/ accessed 2021-04-03.

[2] E. Almeida, E. C. Santos, A. Alvaro, V. Garcia, S. Meira, D. Lucrédio, and R. Fortes. 2008. Domain Implementation in Software Product Lines Using OSGi. *Seventh International Conference on Composition-Based Software Systems (ICCBSS 2008)* (2008), 72–81.

[3] Sofia Ananieva, T. Kehrer, Heiko Klare, A. Koziolek, Henrik Lönn, S. Ramesh, A. Burger, G. Taentzer, and B. Westfechtel. 2019. Towards a Conceptual Model for Unifying Variability in Space and Time. *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume B* (2019).

[4] S. Apel, D. Batory, Christian Kstner, and G. Saake. 2013. Feature-Oriented Software Product Lines: Concepts and Implementation.

[5] David Benavides, A. Cortés, Pablo Trinidad Martín-Arroyo, and S. Segura. 2006. A Survey on the Automated Analyses of Feature Models. In *JISBD*.

[6] Nelly Bencomo and Gordon Blair. 2009. Using architecture models to support the generation and operation of component-based adaptive systems. In *Software engineering for self-adaptive systems*. Springer, 183–200.

[7] Kathrin Berg, Judith Bishop, and Dirk Muthig. 2005. Tracing software product line variability: from problem to solution space. In *SAICSIT*, Vol. 5. Citeseer, 182–191.

[8] T. Berger. 2012. Variability modeling in the wild. In *SPLC '12*.

[9] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wasowski, and Krzysztof Czarnecki. 2013. A study of variability models and languages in the systems software domain. *IEEE Transactions on Software Engineering* 39, 12 (2013), 1611–1640.

[10] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki. 2013. A Study of Variability Models and Languages in the Systems Software Domain. *IEEE Transactions on Software Engineering* 39, 12 (2013), 1611–1640. https://doi.org/10.1109/TSE.2013.34

[11] Danilo Beuche. 2013. pure:: variants. In *Systems and Software Variability Management*. Springer, 173–182.

[12] G. Booch, R. A. Maksimchuk, M. Engle, Bobbi J. Young, and Jim Conallen. 2008. Object-oriented analysis and design with applications, third edition. *ACM SIGSOFT Softw. Eng. Notes* 33 (2008).

[13] J. Bosch, C. Szyperski, and W. Weck. 2003. Component-Oriented Programming. In *ECOOP Workshops*.

[14] L. Bressan, A. L. D. Oliveira, Fernanda Campos, and R. Capilla. 2021. A variability modeling and transformation approach for safety-critical systems. *15th International Working Conference on Variability Modelling of Software-Intensive Systems* (2021).

[15] A. Brooks, T. Kaupp, Alexei Makarenko, Stefan B. Williams, and Anders Orebäck. 2005. Towards component-based robotics. *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems* (2005), 163–168.

[16] Junior Cupe Casquina, Jane DA Sandim Eleuterio, and Cecilia MF Rubira. 2016. Adaptive deployment infrastructure for android applications. In *2016 12th European Dependable Computing Conference (EDCC)*. IEEE, 218–228.

[17] Chaminda Chandrasekara and Pushpa Herath. 2020. Getting Started with Azure Git Repos. In *Hands-on Azure Repos*. Springer, 139–170.

[18] Paul Clements and Linda Northrop. 2002. Software product lines - practices and patterns. In *SEI series in software engineering*.

[19] Equinox Committers. 2021. Equinox | The Eclipse Foundation. https://www.eclipse.org/equinox/ accessed 2021-04-03.

[20] Marcus Vinicius Couto, Marco Tulio Valente, and Eduardo Figueiredo. 2011. Extracting software product lines: A case study using conditional compilation. In *2011 15th European Conference on Software Maintenance and Reengineering*. IEEE, 191–200.

[21] Javier Cuenca, Felix Larrinaga, and Edward Curry. 2019. Experiences on applying SPL Engineering Techniques to Design a (Re) usable Ontology in the Energy Domain.. In *SEKE*. 606–777.

[22] K. Czarnecki, P. Grünbacher, Rick Rabiser, K. Schmid, and A. Wasowski. 2012. Cool features and tough decisions: a comparison of variability modeling approaches. In *VaMoS '12*.

[23] MC Jr da Silva, Paulo A de C Guerra, and Cecília MF Rubira. 2003. A java component model for evolving software systems. In *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.* IEEE, 327–330.

[24] Ferruccio Damiani and Ina Schaefer. 2011. Dynamic delta-oriented programming. In *Proceedings of the 15th International Software Product Line Conference, Volume 2 (SPLC '11)*. Association for Computing Machinery, Munich, Germany, 1–8. https://doi.org/10.1145/2019136.2019175

[25] Darcs. 2021. Darcs. http://darcs.net/ accessed 2021-04-03.

[26] João P Diniz, Gustavo Vale, Felipe Gaia, and Eduardo Figueiredo. 2017. Evaluating delta-oriented programming for evolving software product lines. In *2017 IEEE/ACM 2nd International Workshop on Variability and Complexity in Software Design (VACE)*. IEEE, 27–33.

[27] Docker. 2021. Empowering App Development for Developers | Docker. https://www.docker.com/ accessed 2021-04-03.

[28] H. Eichelberger and K. Schmid. 2013. A systematic analysis of textual variability modeling languages. In *SPLC '13*.

[29] D. Faust and C. Verhoef. 2003. Software product line migration and deployment. *Software: Practice and Experience* 33 (2003).

[30] FeatureHouse. 2021. FeatureHouse. https://www.se.cs.uni-saarland.de/apel/fh/ accessed 2021-04-03.

[31] Fischer Ferreira, Markos Viggiato, M. Souza, and E. Figueiredo. 2020. Testing configurable software systems: the failure observation challenge. *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A - Volume A* (2020).

[32] Stefan Fischer, Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2014. Enhancing clone-and-own with systematic reuse for developing software variants. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 391–400.

[33] Rachel Gawley. 2007. Automating the identification of variability realisation techniques from feature models. In *ASE '07*.

[34] D. Germán, B. Adams, and A. Hassan. 2014. Continuously mining distributed version control systems: an empirical study of how Linux uses Git. *Empirical Software Engineering* 21 (2014), 260–299.

[35] H. Gomaa. 2005. Designing Software Product Lines with UML. *29th Annual IEEE/NASA Software Engineering Workshop - Tutorial Notes (SEW'05)* (2005), 160–216.

[36] Hassan Haidar, Manuel Kolp, and Yves Wautelet. 2017. Goal-oriented requirement engineering for agile software product lines: an overview. *Louvain School of Management Research Institute Working Paper Series, Louvain, Belgium* (2017), 1–36.

[37] Øystein Haugen, Andrzej Wąsowski, and Krzysztof Czarnecki. 2012. CVL: common variability language. In *Proceedings of the 16th International Software Product Line Conference-Volume 2*. 266–267.

[38] Robert Hellebrand, M. Schulze, and Martin Becker. 2016. A branching model for variability-affected cyber-physical systems. *2016 3rd International Workshop on Emerging Ideas and Trends in Engineering of Cyber-Physical Systems (EITEC)* (2016), 47–52.

[39] Jose Miguel Horcas Aguilera et al. 2018. WeaFQAs: A Software Product Line Approach for Customizing and Weaving Efficient Functional Quality Attributes. (2018).

[40] Ying Hu et al. 2000. C/C++ conditional compilation analysis using symbolic execution. In *Proceedings 2000 International Conference on Software Maintenance*. IEEE, 196–206.

[41] Y. Hu, E. Merlo, M. Dagenais, and B. Laguë. 2000. C/C++ conditional compilation analysis using symbolic execution. *Proceedings 2000 International Conference on Software Maintenance* (2000), 196–206.

[42] C. Kaestner, Paolo G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. 2011. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *OOPSLA '11*.

[43] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. 2001. An Overview of AspectJ. In *ECOOP 2001 — Object-Oriented Programming (Lecture Notes in Computer Science)*, Jørgen Lindskov Knudsen (Ed.). Springer, Berlin, Heidelberg, 327–354. https://doi.org/10.1007/3-540-45337-7_18

[44] Chang Hwan Peter Kim, Eric Bodden, Don Batory, and Sarfraz Khurshid. 2010. Reducing configurations to monitor in a software product line. In *International Conference on Runtime Verification*. Springer, 285–299.

[45] Jacob Krüger, Ivonne Schröter, Andy Kenner, Christopher Kruczek, and Thomas Leich. 2016. FeatureCoPP: compositional annotations. In *Proceedings of the 7th International Workshop on Feature-Oriented Software Development*. 74–84.

[46] LabSoft. 2021. LabSoft. http://labsoft.dcc.ufmg.br/doku.php?id=about:spl_list accessed 2021-04-03.

[47] Lukas Linsbauer. 2016. A variability aware configuration management and revision control platform. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 803–806.

[48] Lukas Linsbauer, T. Berger, and P. Grünbacher. 2017. A classification of variation control systems. *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (2017).

[49] Lukas Linsbauer, Felix Schwägerl, T. Berger, and Paul Grünbacher. 2021. Concepts of variation control systems. *J. Syst. Softw.* 171 (2021), 110796.

[50] J. Martinez, Wesley K. G. Assunção, and T. Ziadi. 2017. ESPLA: A Catalog of Extractive SPL Adoption Case Studies. *Proceedings of the 21st International Systems and Software Product Line Conference - Volume B* (2017).

[51] Jabier Martinez, Xhevahire Tërnava, and Tewfik Ziadi. 2018. Software product line extraction from variability-rich systems: the robocode case study. In *Proceedings of the 22nd International Systems and Software Product Line Conference-Volume 1*. 132–142.

[52] John D McGregor, Linda M Northrop, Salah Jarrad, and Klaus Pohl. 2002. Initiating software product lines. *IEEE Software* 19, 4 (2002), 24.

[53] M Douglas McIlroy, J Buxton, Peter Naur, and Brian Randell. 1968. Mass-produced software components. In *Proceedings of the 1st international conference on software engineering, Garmisch Pattenkirchen, Germany*. 88–98.

[54] Jens Meinicke, Thomas ThŘm, Reimar Schr, Fabian Benduhn, Thomas Leich, Gunter Saake, et al. 2017. Conditional Compilation with FeatureIDE. In *Mastering Software Variability with FeatureIDE*. Springer, 97–103.

[55] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering Software Variability with FeatureIDE*. Springer International Publishing, Cham. https://doi.org/10.1007/978-3-319-61443-4

[56] Andreas Metzger and Klaus Pohl. 2007. Variability management in software product line engineering. In *29th International Conference on Software Engineering (ICSE'07 Companion)*. IEEE, 186–187.

[57] G. K. Michelon. 2020. Evolving System Families in Space and Time. *Proceedings of the 24th ACM International Systems and Software Product Line Conference - Volume B* (2020).

[58] Leticia Montalvillo-Mendizabal, O. Díaz, and Thomas Fogdal. 2018. Reducing coordination overhead in SPLs: peering in on peers. *Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1* (2018).

[59] A. S. Nascimento, C. Rubira, R. Burrows, and F. C. Filho. 2013. A Model-Driven Infrastructure for Developing Product Line Architectures Using CVL. *2013 VII Brazilian Symposium on Software Components, Architectures and Reuse* (2013), 119–128.

[60] D. Nestor, L. O'Malley, Aaron J. Quigley, E. Sikora, and S. Thiel. 2007. Visualisation of Variability in Software Product Line Engineering. In *VaMoS*.

[61] B. O'Donovan and J. Grimson. 1990. A distributed version control system for wide area networks. *Softw. Eng. J.* 5 (1990), 255–262.

[62] Nicolás Paez. 2018. Versioning Strategy for DevOps Implementations. In *2018 Congreso Argentino de Ciencias de La Informática y Desarrollos de Investigación (CACIDI)*. IEEE, 1–6.

[63] Richard F Paige, Xiaochen Wang, Zoë R Stephenson, and Phillip J Brooke. 2006. Towards an agile process for building software product lines. In *International Conference on Extreme Programming and Agile Processes in Software Engineering*. Springer, 198–199.

[64] N. Pessemier, L. Seinturier, T. Coupaye, and L. Duchien. 2006. A Safe Aspect-Oriented Programming Support for Component-Oriented Programming.

[65] K. Pohl and A. Metzger. 2018. Software Product Lines. In *The Essence of Software Engineering*.

[66] C. Prehofer. 1997. Feature-Oriented Programming: A Fresh Look at Objects. In *ECOOP*.

[67] C. Prehofer. 2001. Feature-oriented programming: A new way of object composition. *Concurrency and Computation: Practice and Experience* 13 (2001).

[68] Nayan B Ruparelia. 2010. The history of version control. *ACM SIGSOFT Software Engineering Notes* 35, 1 (2010), 5–9.

[69] Pablo Sánchez, Neil Loughran, Lidia Fuentes, and Alessandro Garcia. 2008. Engineering languages for specifying product-derivation processes in software product lines. In *International Conference on Software Language Engineering*. Springer, 188–207.

[70] K. Schmid, Rick Rabiser, and P. Grünbacher. 2011. A comparison of decision modeling approaches in product lines. In *VaMoS '11*.

[71] Felix Schwägerl, Thomas Buchmann, and Bernhard Westfechtel. 2015. SuperMod—A model-driven tool that combines version control and software product line engineering. In *2015 10th International Joint Conference on Software Technologies (ICSOFT)*, Vol. 2. IEEE, 1–14.

[72] C. Seidl, I. Schaefer, and U. Assmann. 2014. Capturing variability in space and time with hyper feature models. In *VaMoS '14*.

[73] Russell G. Shirey, K. Hopkinson, K. E. Stewart, D. Hodson, and Brett J. Borghetti. 2015. Analysis of Implementations to Secure Git for Use as an Encrypted Distributed Version Control System. *2015 48th Hawaii International Conference on System Sciences* (2015), 5310–5319.

[74] Stéphane S Somé and Timothy C Lethbridge. 1998. Parsing minimization when extracting information from code in the presence of conditional compilation. In *Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (Cat. No. 98TB100242)*. IEEE, 118–125.

[75] Olaf Spinczyk and Daniel Lohmann. 2007. The design and implementation of AspectC++. *Knowledge-Based Systems* 20, 7 (2007), 636–651.

[76] Mohammad Tanhaei, Jafar Habibi, and Seyed-Hassan Mirian-Hosseinabadi. 2016. Automating feature model refactoring: A Model transformation approach. *Information and Software Technology* 80 (2016), 138–157. https://doi.org/10.1016/j.infsof.2016.08.011

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

[78] Leonardo P Tizzei, Marcelo Dias, Cecília MF Rubira, Alessandro Garcia, and Jaejoon Lee. 2011. Components meet aspects: Assessing design stability of a software product line. *Information and Software Technology* 53, 2 (2011), 121–136.

[79] Salvador Trujillo, Don Batory, and Oscar Diaz. 2007. Feature oriented model driven development: A case study for portlets. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 44–53.

[80] Edoardo Vacchi and W. Cazzola. 2015. Neverlang: A framework for feature-oriented language development. *Comput. Lang. Syst. Struct.* 43 (2015), 1–40.

[81] Frank J Van der Linden, Klaus Schmid, and Eelco Rommes. 2007. *Software product lines in action: the best industrial practice in product line engineering*. Springer Science & Business Media.

[82] John Viega, Joshua T Bloch, and Pravir Chandra. 2001. Applying aspect-oriented programming to security. *Cutter IT Journal* 14, 2 (2001), 31–39.

[83] Gustavo M Waku, Edson R Bollis, Cecilia MF Rubira, and Ricardo da S Torres. 2015. A robust software product line architecture for data collection in android platform. In *2015 IX Brazilian Symposium on Components, Architectures and Reuse Software*. IEEE, 31–39.

[84] Gustavo M Waku, Cecilia MF Rubira, and Leonardo P Tizzei. 2015. A case study using aop and components to build software product lines in android platform. In *2015 41st Euromicro Conference on Software Engineering and Advanced Applications*. IEEE, 418–421.

[85] Andy Ju An Wang, Kai Qian, et al. 2005. *Component-oriented programming*. Vol. 319. Wiley Online Library.