Migrating Individual Applications into Software Product Lines using the Mobioos Forge Platform

Karim Ghallab, Tewfik Ziadi, Zaak Chalal

Migrating Individual Applications into Software Product Lines using the Mobioos Forge Platform

Karim Ghallab[1], Tewfik Ziadi[1,2], Zaak Chalal[1]
[1]*RedFabriQ*, Paris, France
[2]*Sorbonne University - LIP6 - CNRS*, Paris, France
Karim.Ghallab@mobioos.ai, Tewfik.Ziadi@lip6.fr, Zaak.Chalal@mobioos.ai

***Abstract*—The adoption of Software Product Lines in the industry remains a major challenge. This paper presents an experience report focused on the application of a novel tool-based approach called *Mobioos Forge*. We introduce the vision and operational activities of *Mobioos Forge*, emphasizing its significance through an examination of the complex process of migrating the *ArgoUML* application – an open-source codebase exceeding 400KLOCs. We highlight the achieved feature model and detail the feature-to-source code mapping. Additionally, we explain the derivation process used to generate the source code for multiple variants. We discuss the time and effort expended on this migration, showcasing that, even with no prior familiarity with ArgoUML, it took less than 11 hours to successfully migrate the entire application into an SPL.**

***Index Terms*—Software Product Lines, Mobioos Forge, ArgoUML**

I. Introduction

Software Product Line Engineering (SPLE) aims to improve reuse by focusing not on the development of a single software product but on a family of related products. The systems in a Software Product Line (SPL) approach are developed from a common set of assets in a prescribed way, in contrast to being developed separately, from scratch, or in an ad-hoc manner. This production economy makes the SPL approach attractive. SPLE considers the existence of a single architecture describing all the variants that implement different software products of a single product line. The particularity of this architecture is that it includes what is refereed as a variability model (also called feature model), in which variability and commonality are explicitly specified using high level characteristics of the so-called features [1]. These are then mapped to the source code, which are organized according to the identified features. Specific software variants can be derived (generated) by choosing from the feature model a set of desired features, then SPL tools customize the source code to generate specific variants [1].

Still, adopting SPL remains a major challenge for companies [2], [3]. Compared to single-system development, SPL variability management implies a methodology that highly impacts the life cycle of the products and the processes and roles inside the company [4]. From a developer perspective, it is very challenging to manage at the same time constraints related to classical development activities and tasks dealing with software variability implementation. These constraints represent barriers to SPL adoption in the industry.

To facilitate SPL adoption in industry, we recently proposed a new tool-supported approach called *Mobioos Forge* (MF). Instead to create the SPL from scratch, MF offers the guidance to developers for migrating existing individual applications to SPL. This paper presents our experience on using MF to migrate the ArgoUML open-source application into SPL. By exploring ArgoUML's source code, we showcase the conceived feature model and illustrate how MF effectively facilitates the establishment of a complete mapping between the features and the source code. Metrics related to feature mapping are presented to provide a comprehensive view of this feature-to-source code mapping. Moreover, the advantages inherent in the resultant SPL are underscored by employing MF's derivation engine. This engine is employed to generate tailored new variants by deriving source code from diverse configurations. In addition, we also present metrics related to the time and effort spent on the migration process.

The paper is organized as follows: Section II exposes background information about SPL and its adoption challenges. Section III introduces the MF platform. Section IV describes our experimentation and evaluation of MF thought the migration the ArgoUML case study into an SPL. Lastly, Section V provides an overview of related works concerning the migration of applications into SPLs.

II. Background

A. Software Product Lines Engineering

A Software Product Line (SPL) is defined as *"a set of software-intensive systems that share a common, managed set of features satisfying 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"* [5]. The inspiration for proposing this engineering practice is commonly attributed to the manufacturing industry where different predefined reusable components are usually combined to satisfy different customer needs.

Achieving large-scale productivity gains and improving time-to-market and product quality are some of the claimed benefits of SPLE [5], [6]. Some acknowledged examples can be found in the SPL hall of fame [7] which reports commercially successful implementation of the SPL paradigm in companies from different domains ranging from avionics and automotive software, to printers, mobile phones or web-based systems. As illustrated in Figure 1, the general SPL

Engineering (SPLE) framework is defined with the dual phases of *domain* and *application* engineering [8].

During domain engineering, the scope of the SPL is defined, *commonality* and *variability* among products are explicitly specified using the concepts of *features* and *feature models* [3], [9]. feature models are typically tree-like structures that hierarchically organize features. Cross-tree constraints can also be specified in a feature model, representing logical expressions that establish additional relationships among features. For instance, Figure 1 illustrates a feature model of a simple text editor application. The features *Save*, *Editor actions*, and *Find* are sub-features of the root feature *Notepad*, whereas *Copy* and *Cut* are sub-features of *Editor actions*. Moreover, *Save* and *Editor actions* are mandatory, implying their selection alongside their parent feature *Notepad*. Conversely, *Find* is optional, and features *Copy* and *Cut* share an *OR* relationship with their parent feature *Editor actions*. Domain engineering also aims at implementing the SPL at the source code level. Various techniques exist for SPL implementation using *annotations*, *templates*, *aspects*, *deltas*, or *modules* [10]–[20]. While these approaches may vary, most of them fall into two categories: annotative or compositional [21]–[23]. The annotative approaches involve textual or visual annotations that encapsulate feature implementations, as depicted in Figure 2a. While in the compositional approaches, features are implemented as separate code units, often referred to as *composition units* [20], *feature modules* [9], [18], or *delta modules* [15], [16] (see Figure 2b). The underlying concept involves commencing with a foundational core module and incrementally incorporating supplementary modules to progressively introduce novel elements while refining, substituting [19], [24], or eliminating existing ones [15], [16], [25]. Despite the substantial divergence between these two methodologies, each one presents distinct merits and limitations [26].

The application engineering phase is related to *product derivation*, where specific product variants are derived and created by reusing the results of domain engineering. This includes create what is called a valid *configuration* that outlines the desired features for a specific product variant, adhering to the rules established by the feature model. Once the configuration has been established, the process of *product derivation* aims generating a customized variant that aligns with the specifications outlined in the configuration. The specifics of the product derivation process depend on the chosen implementation strategy within the domain. This process could involve removing or replacing annotated code segments or relying on a composer. The composer constructs the product by assembling the selected modules, guided by the configuration.

Over the past years, several tools have been proposed to support the SPLE process. We distinguish between research tools like FeatureIDE [27], as well as industrial tools such as Pure::variant [28] and BigLiver/Gears [29]. However, the adoption of SPLs within the industrial landscape remains a substantial challenge. The following subsection discuss these challenges.

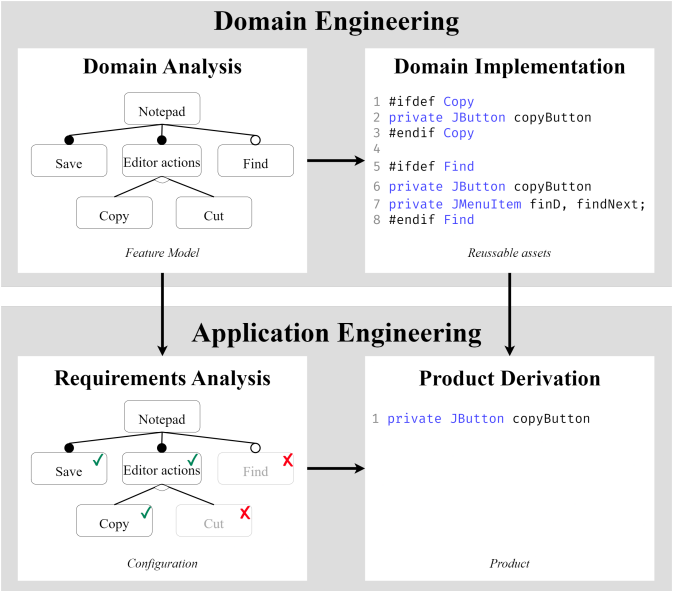
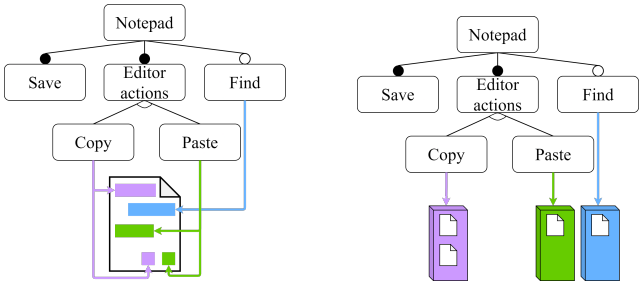


Fig. 1: The Software product lines engineering process.



(a) Annotative approach (b) Compositional approach

Fig. 2: SPL implementations approaches.

B. Challenges for Software Product Lines Adoption

Even though the SPLE process, encompassing both domain and application engineering phases, has reached a level of maturity and is widely supported by both research and industrial tools, the adoption of SPLs by companies remains a major challenge. Instead of following the SPLE process to create product lines, companies increasingly employ to using ad hoc techniques to develop variants of the same software. For example, the clone-and-own approach, which involves creating variants by duplicating and modifying existing products to meet new requirements, remains extensively adopted by companies [30].

The use of ad hoc techniques to create variants of the SPL instead of following the SPLE process can be justified by two main reasons. Firstly, implementing an SPL from scratch, which involves the dual phases of domain and application engineering, requires an upfront investment that companies may struggle to support. Secondly, implementing SPL from scratch demands an understanding of variability from the beginning. As discussed in numerous studies [31], software variability often becomes apparent only after initial variants

have been implemented using ad hoc reuse approaches.

To facilitate the adoption of SPL, we recently proposed a new platform named Mobioos Forge (MF). This platform is distributed for free and aims to re-visit the SPLE process with a *migration-oriented* perspective. Instead of creating SPL from scratch, MF offers the guidance to developers for migrating existing individual applications to SPL. The following section presents the foundational principles and the various activities provided by MF.

III. The Mobioos Forge Platform

Mobioos Forge (MF) is an extension of *Visual Studio Code* (VScode) that can be freely downloaded from the *VScode MarketPlace*[1]. MF aligns with the SPLE process but proposes a migration-oriented vision allowing developers creating SPLs from the source code of individual applications. In the next subsections, we introduce MF's vision and activities.

A. Overview

MF's vision is divided into four activities: 1) *Feature Model Specification*, 2) *Feature Mapping*, 3) *Constraints and Anomalies Detection*, and, 4) *Variants Derivation*. Figure 3 summarizes the initial three activities, while Figure 4 provides an overview of the final one. To provide a clearer understanding of MF's distinct activities, the upcoming subsections base the presentation of these activities on the Notepad application featured in the MF tutorial [32].

As illustrated in Figure 3, MF's vision follows existing ones and proposes specifying variability at the domain level using the well-known formalism related to feature models [33]. In addition to feature models specification, MF aims to help developers locating the features in the source code of their applications. To do so, it relies on a language-independent semi-automatic process that: given some initial knowledge of the feature implementation, proposes to the developer potential new locations where the features might be implemented. Finally, as seen in Figure 4, MF allows the derivation of variants thought the *Variants Derivation* activity. Taking as input a configuration as well as the mappings between the feature model and the source code, this activity lets developers derive variants fitting the requirements specified in the configuration. The following subsections describes in detail the activities of MF.

B. Feature Model Specification

This activity is the first one in MF's approach. It consists in the description of the features of the application via the specification of a feature model. Taking as input an existing application, the developer initiates the process by identifying the application's features. Subsequently, these features are designed along with their respective relationships by constructing a feature model.

MF integrates a feature model designer in VScode to allow developers specifying the feature model of their applications. Each feature is associated with a color used in the *Feature Mapping* activity The feature model designer also proposes an editor to add *cross-tree constraints* to describe dependencies between features. Figure 5 shows MF's integrated feature model designer used to design the feature model of the Notepad application. This feature model contains 8 features, including the root feature named *Notepad*. It also has 2 *cross-tree constraints* displayed at the bottom of the figure.

[1]https://marketplace.visualstudio.com/items?itemName=Mobioos.mobioos-forge

C. Feature Mapping

Mapping the features to the source code of the application requires making choices about how variability is implemented. As seen in the Section II, there are several approaches to map features to code fragments. This includes annotative and compositional approaches [1], [26]. MF's vision is inspired by the one proposed by CIDE [21] using visual annotations. This means that the code fragments associated with a specific feature are colored with the feature color. Advantages of these colored annotations are multiples: 1) The source code of the application is not altered (information about the annotations are stored in external files), and 2) It is easy to visualize to which feature a given code fragment belongs as the editor applies the feature's color to its related code fragments. Mapping features into the source code uses what is called *markers* that refer to the preliminary information about the mapping of features into the source code. Markers are provided by developers based on their knowledge of the source code and the identified features. MF proposes two types of markers depending on the granularity of the mapping: 1) *Code-markers* that are applied to code fragments. They associate the selected code fragment with a specific feature, and 2) *File-markers* which associate a whole file/folder to a specific feature. Those markers are then used to help the developer identify code fragments and files associated with each feature of the feature model. Once a marker is added, the *Feature Mapping* activity assists developers with a semi-automatic process that infers *Maps*, i.e. code fragments and files/folders belonging to the feature from the initial marker. For MF to compute maps from a marker, the marker must be added on a declaration (class declaration, method declaration etc.). Once a marker is added, the developer must validate/delete the maps provided by MF. Validating a map involves confirming its position within the source code or modifying it to cover a different area of the source code. Once the map is validated, MF computes other potential maps within the source code that could be associated with the related feature. It is important to note that validating one map can lead to the automatic validation of other maps. If the newly validated map's area covers other maps related to the same feature, these maps will be automatically validated. Hence decreasing the number of required validation, resulting in a reduction of the developer's workload.

As previously mentioned, MF shows the mapping of a given feature in the source code by applying its related feature's color on the code fragment. Regarding code-markers, they are displayed as colored annotations in the VScode editor, whereas file-markers are shown in the *Mobioos Forge Feature-*

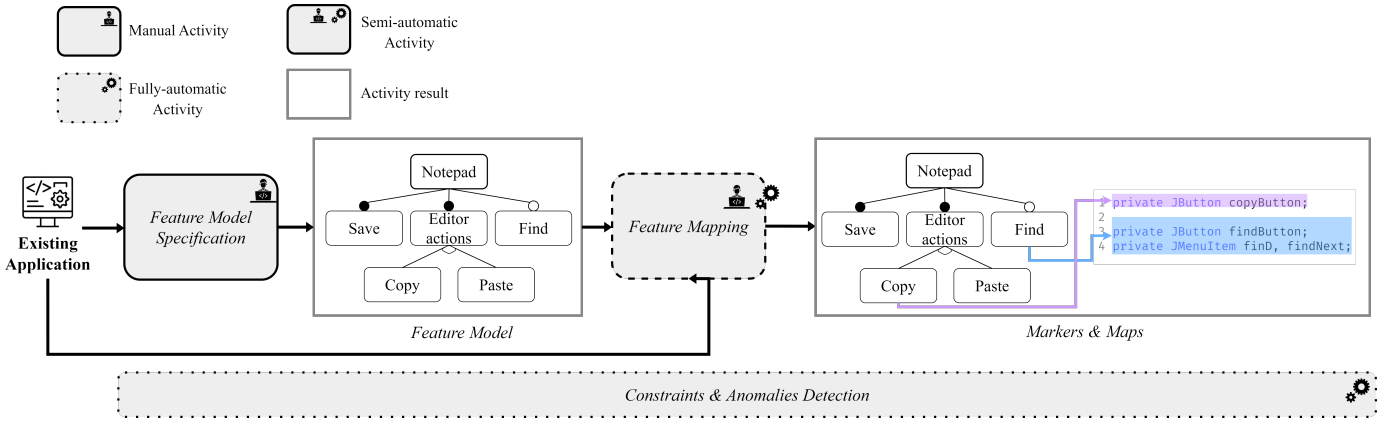


Fig. 3: Domain Engineering in MF's vision.

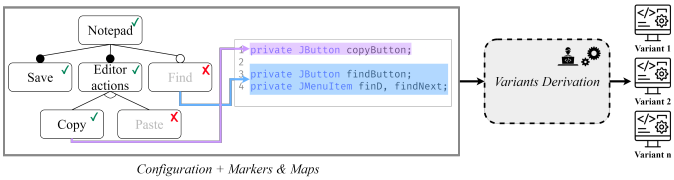


Fig. 4: Application Engineering in MF's vision.

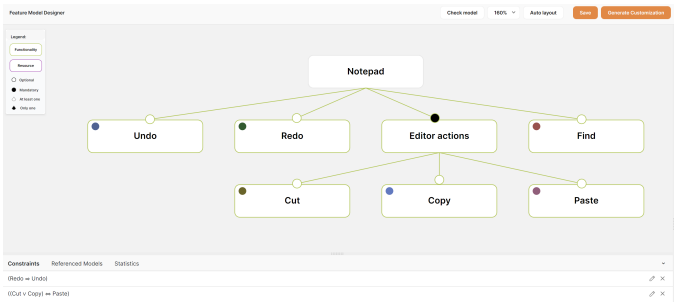


Fig. 5: Notepad's feature model created using MF.

Maps View. Figure 6 enhances examples of markers and maps present within the Notepad application. In this figure, we can see on the left an example of a file-marker added on file `RedoAction.java` as well as maps inside the file `Noetpad.java`. On the right side of the figure, several maps are displayed on the VScode editor. They are distinguishable by the colors applied in the editor.

Fig. 6: Examples of markers and maps in MF.

D. Constraints and Anomalies Detection

In addition to helping the migration of a given application into an SPL, MF also aims to analyze the built SPL in order to detect possible anomalies in the domain [34] and its implementation [35]. Not only does MF detect these anomalies, but it also offers suggestions for constraints to be incorporated into the feature model to rectify the identified anomalies whenever feasible. This activity is fully automatic as its analysis run in background while the developers design the feature model or map the features into the source code.

Detecting and solving such anomalies can be done by using a SAT solver [36]–[38], thereby, MF's VScode extension embed a SAT solver to allow detecting and solving anomalies. The detection of anomalies related to the feature model is run in background while developers design the feature model. The detected anomalies are then shown in a pop-up window of the feature model designer. Anomalies related to the features' implementations in the source code are computed while the *Feature Model Specification* and the *Feature Mapping* activities. Computed anomalies are shown in VScode's *problem* view. Figure 7 shows an example of anomaly and how it is shown in MF. In the depicted figure, a feature model is shown containing two optional features, namely B and C. In the implementations of these features, the source code of C is encompassed within B. However, this relationship existing in the source code is not present in the feature model. MF detects this inconsistency and highlights it by presenting a message in VScode's *problem* view. Additionally, MF proposes a resolution by suggesting the addition of the constraint $C \implies B$ to the feature model.

E. Variants Derivation

As for the *Feature Model Specification* activity, MF follows other existing approaches and proposes specifying specific requirements though the usage of configurations. A configuration is a subset of features that need to be taken in the generated variant. Regarding the derivation process, as MF implements the variability using an annotative approach, the approach follows a classical derivation process where code

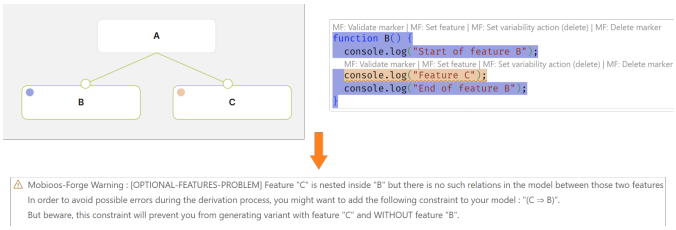


Fig. 7: A constraint detection in MF.

fragments and files belonging to features not included in the configuration are deleted/replaced.

MF's derivation engine includes a graphical configuration editor enriched by a SAT Solver component [37] to ensure the creation of *valid* configurations, i.e., configurations that respect the rules issued by the feature model. The editor lets the user specify the features to include in the variant. Once the configuration is fully specified, MF derives the associated variant following other derivation processes regarding annotative approaches. Thus, deriving a variant in MF consists in deleting/replacing the code-fragments and files related to the features not included in the configuration. Figure 8 shows the described configuration editor embedded in MF. This configuration has been used on the Notepad SPL. As you can see, only the features *Redo* and *Undo* are selected.

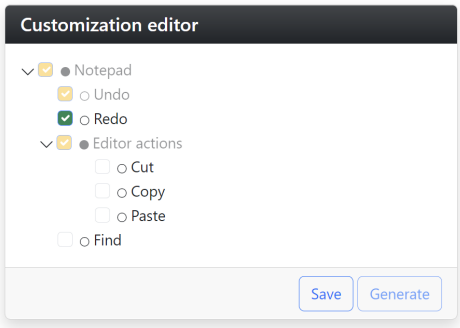


Fig. 8: The configuration editor in MF.

IV. The Migration of ArgoUML into an SPL

This section presents the ArgoUML case study, as well as our outcomes concerning its migration into an SPL using MF's VScode extension. We present a detailed account of the outcomes achieved during the migration process, as well as the time and effort required for the complete migration of ArgoUML into an SPL.

A. The ArgoUML Case Study

In the context of SPL migration using the capabilities of MF, the ArgoUML application serves as an illustrative case study. ArgoUML, is a Java/Maven open-source UML modeling tool available on Github[2]. Previous works have explored the migration of ArgoUML into an SPL [39], [40]. Therefore, we believe it provides an interesting scenario to demonstrate the effectiveness of MF in migrating an individual application into an SPL. ArgoUML allows the graphical modeling of various diagrams. These diagrams are: *Class*, *State*, *Activity*, *Use Case*, *Sequence*, *Deployment*, and *Collaboration* diagrams. The application has about 400KLOCs lines of code (LOCs). In this paper, the LOCs metrics counts the number of lines of code without blanks lines.

[2]https://github.com/argouml-tigris-org/argouml

B. Results and Discussion

In the subsequent sections, we delve into the results of the ArgoUML migration process. This exploration not only sheds light on the migration process using MF, but also provides valuable insights into the challenges and successes encountered during the SPL migration using ArgoUML as a reference point.

1) Feature Model: Prior works already exist regarding the migration of ArgoUML into an SPL [39], [40]. Therefore, we have opted to create a feature model similar to those developed in these prior studies. Figure 9 show the designed feature model for the ArgoUML SPL. This feature model contains the seven diagram types supported by ArgoUML (*Class*, *State*, *Activity*, *Use Case*, *Collaboration*, *Deployment* and *Sequence*). It also describes two optional features named *Cognitive Support* and *Logging*. The *Cognitive Support* feature offers insights to help diagram designers identify and resolve issues within their models [41]. This feature is realized through software agents operating persistently within a background thread of control. the *Logging* feature, as its name suggests, logs messages throughout the application. These messages can be error messages, as well as simple informational messages enabling the tracking of execution progress. These last two features distinguish themselves from the others due to their crosscutting nature as they have an impact across the entire application.

The presented feature model is akin to those in prior works, with the exception of the cross-tree constraint $Activity \implies State$, which we have introduced ourselves. The motivation behind this constraint addition is explained upon in Section IV-B3 related to MF's detected constraints.

2) Markers and Maps: This activity is of notable interest for analysis since we had no prior knowledge regarding the ArgoUML implementation. The initial step involved identifying candidates for marker addition. To accomplish this, we conducted searches for class or package names that align with the defined features' names in the feature model. Once these classes and packages were identified, we proceeded to add both code markers and file markers based on the context. Following these additions, which generated various maps, the ensuing step was to validate these maps. This validation process allowed us to uncover the features' implementations within the source code. If necessary, we subsequently introduced additional markers to comprehensively complete the feature mapping. Figure 10 illustrates two pie-charts that shows for each feature: the quantity of markers that were

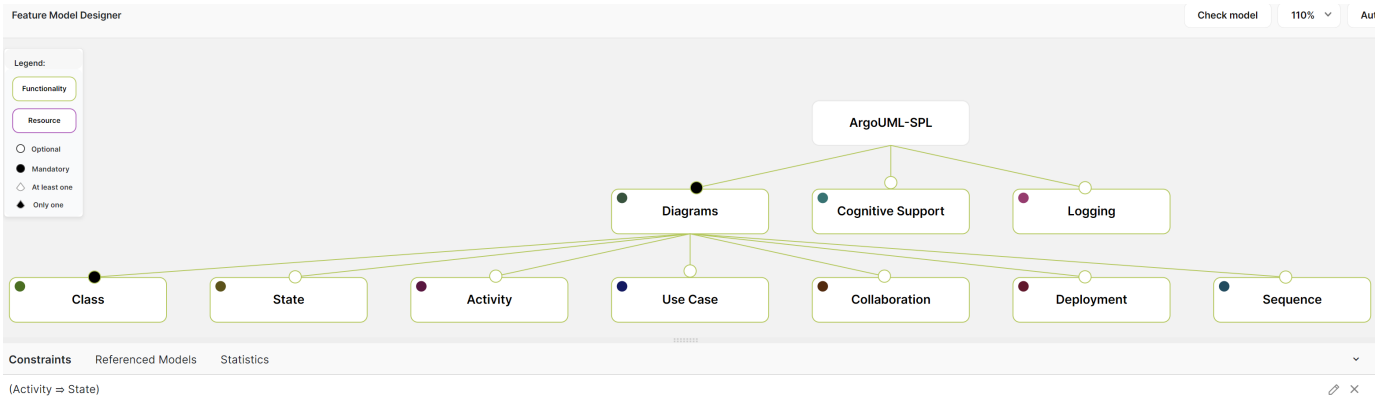


Fig. 9: ArgoUML's resulting feature model.

added (left pie-chart), and, the total number of maps computed by MF from these markers (right pie-chart).

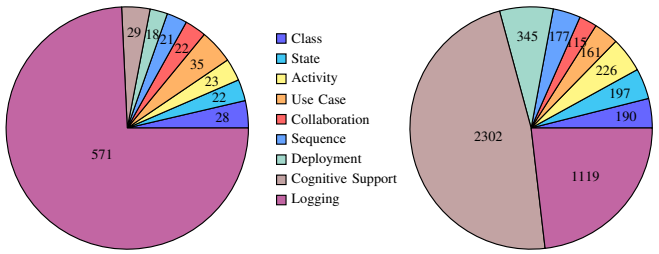


Fig. 10: Quantity of markers & maps in the ArgoUML SPL.

By examining these two charts, we can easily discern which features required the highest number of markers to be fully mapped within the code. This allows for a direct comparison between the marker count and the number of maps calculated by MF.s The first observation that becomes evident when examining the marker count is that the *Logging* feature required a significantly higher number of markers compared to all other features. While the marker count for the other eight features ranges between 18 and 35, with an average of $\approx$ 25 markers per feature, the *Logging* feature alone accounts for 571 markers, approximately 74.25% of the total number of added markers. As we proceed, upon considering the number of maps generated from these markers, it is noticeable that despite its extensive number of markers, the *Logging* feature does not have the highest number of associated maps. The reason behind this limited number of maps lies in the fact that MF exclusively computes maps when markers are added on declarations. However, the loggers used throughout the application were: either integrated in Java or provided by the third-party library `Log4j`. Consequently, there were no declarations of these logger classes within the project. Surprisingly, the *Cognitive Support* feature, with a relatively small marker count of 29, has computed a substantial 2302 maps. Accounting for 47.64% of the SPL total number of map. Meaning that we obtained approximately ≈ 79.3 times the number of maps. In comparison, we only obtained ≈ 1.96 times more maps than markers for the *Logging* feature.

The notable number of maps generated from the markers emphasizes MF's contribution in facilitating the mapping of the features into the source code. In the absence of MF, manual identification of feature locations throughout the source code would have been necessary. Table I presents metrics concerning the *Feature Mapping* activity within the ArgoUML SPL. This table provides, for each feature, the associated number LOCs, the percentage of LOCs relative to the total mapped LOC count, the number of impacted files, as well as the Maps/Marker ratio (i.e. the average number of maps generated by MF given one marker).

Feature	LOCs	Percentage of Mapped LOCs	Impacted Files	Maps / Marker
Class	6042	6.93%	68	6.79
State	4696	5.39%	64	8.95
Activity	6845	7.85%	96	9.83
Use Case	8239	9.45%	81	4.60
Collaboration	4030	4.62%	51	5.23
Sequence	6053	6.94%	113	8.43
Deployment	7067	8.11%	93	19.17
Cognitive Support	40763	46.77%	335	79.38
Logging	3426	3.93%	283	1.96

TABLE I: Metrics about the feature-mapping activity.

By examining the table, we can observe that the number of LOCs for the various diagrams ranges between 4.62% and 9.45% of the total number of mapped LOCs. In contrast, *Cognitive Support* accounts for almost half of the total number of mapped LOCs (46.77%). This explains its high number of maps compared to the other features. The feature *Logging*, on the other hand, is the smallest feature in terms of LOCs. If we examine the *Impacted Files* column, we can observe that, as mentioned during the ArgoUML's feature model presentation (Section IV-B1), the *Cognitive Support* and *Logging* features have the most widespread impact. The implementation of *Cognitive Support* spans across 335 different files, while *Logging*, on the other hand, is found in 283 files. Despite having the fewest number of LOCs, *Logging* has the second highest number of impacted files.

3) Detected Constraints: During the *Feature Model Specification* and *Feature Mapping* activities, MF executed background constraints and anomalies detection. This analysis notably led to the detection of a constraint between two features that was not present in the feature models proposed in previous works. The origin of this constraint arises from an `extends` relation between two classes defined within the project. Those two classes are what ArgoUML calls *graph model*. A graph model establishes rules for manipulating a specific diagram (such as possible additions or deletions). Thus, MF detected that the implementation of *State* is nested inside the implementation of *Activity*. This nestedness between *Activity* and *State* exists because: the graph model for activity diagrams (class `ActivityDiagramGraphModel`) extends the graph model of state diagram (class `StateDiagramGraphModel`). Figure 11 illustrates a screenshot of the detection and proposed resolution by MF. In this figure, we can observe that the implementation of the *State* feature is included within the *Activity* feature. Detecting that no similar relationship existed in the feature model, MF suggested adding the following constraint: $Activity \implies State$. The addition of this cross-tree constraint is visible in Figure 9

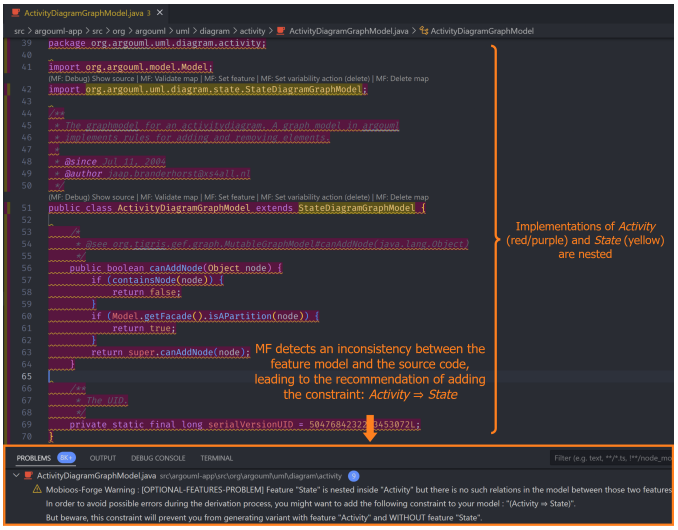


Fig. 11: Constraint discovered by MF.

Other constraints and anomalies were detected by MF. However, only this one, due to its impact on the source code, motivated us to introduce a cross-tree constraint. Specifically, the relationship between *Activity* and *State* is such that *Activity* cannot properly work when *State* is disabled.

4) Derived Variants:

a) Metrics: Once all the features were mapped within the source code, we derived ArgoUML variants to test our migration. Table II presents metrics for the original application and 5 variants generated from MF. For each variant, we provide its number of LOCs as well as the number of deleted LOCs relative to the SPL's total LOCs. The Variants column describes the variants by listing their respective enabled features.

Variants (enabled features)	**LOCs**	**Deleted LOCs**
All features (original application)	413086	0
Only *Class*	355349	57737
Class, *Use Case* and *Collaboration*	362824	50262
Class, *State*, *Activity* and *Cognitive*	394790	18296
Class, *State*, *Deployment*, *Sequence* and *Logging*	361187	51899
All diagram features	369039	44047

TABLE II: Metrics about some generated variants.

Figure 12 displays the execution of the *Only Class* variant. We can observe that the buttons used to create diagrams other than class diagrams have been removed. Additionally, we can notice the removal of the "ToDo item" panel. This panel enables the *Cognitive Support* feature to indicate actions for improving/correcting diagrams.

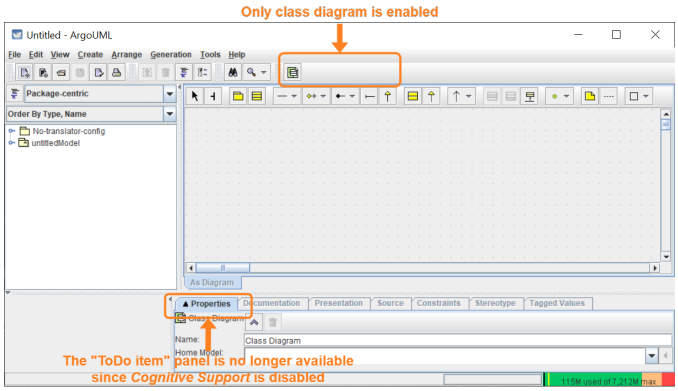


Fig. 12: Execution of the variant with only *Class* enabled.

b) Correctness: Testing the correctness of an SPL is a complex task [42], [43]. To conduct the testing of our ArgoUML's migration, we opted to use the extensive unit test suite already implemented in the ArgoUML application. Our variant testing procedure encompassed two key steps: initializing the unit test suite for the specific variant, and upon successful execution of all tests without any errors, manually triggering the execution of the variant. This enabled us to observe its proper operation and verify that the correct features were enabled/disabled.

The ArgoUML application contains 1225 unit tests implemented across various test files. Naturally, while mapping features within the source code, we also mapped the corresponding feature unit tests. Consequently, a variant containing at least one disabled feature will have a decrease of its number of tests, as these tests were deleted along with the code for the respective disabled feature. Table III shows the number of tests that were executed for each of the presented variants. Given that one of the prerequisites for validating the correctness of a variant is the successful execution of its complete test suite, the number of tests presented in the table aligns with the number of passed tests.

Variants (enabled features)	Unit Tests
All features (original application)	1225
Only *Class*	397
Class Use Case and *Collaboration*	953
Class, *State*, *Activity* and *Cognitive*	1046
Class, *State*, *Deployment*, *Sequence* and *Logging*	952
All diagram features	970

TABLE III: Test metrics about the generated variants.

The ArgoUML SPL built using MF is available on GitHub[3]. The variants presented in this paper are accessible through several branches of the repository.

5) Time and Effort during SPL migration: MF aims to help developers migrate their own applications into SPLs. Therefore, an important data to watch is the time required to map each feature's implementation into the source code using MF. Hence, we quantified the time taken for feature mapping by a junior developer (with less than three years of software development experience) for each individual feature. The timed interval encompasses both the marker addition phase and the complete validation of the resultant maps induced by these markers, culminating in the total mapping of the features into the source code. Throughout our experimentation, a compelling pattern emerged—an apparent correlation between the mapping-time for a feature and the quantity of manually-validated maps associated with it. Figure 13 visually encapsulates this phenomenon, portraying a histogram showcasing the measured durations for mapping each feature. Overlaying this histogram is a plotted line that signifies the count of manually-validated maps corresponding to each feature.

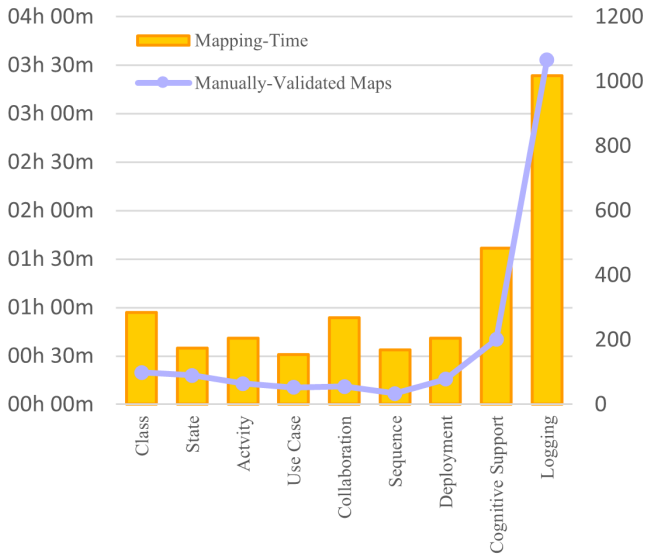


Fig. 13: Mapping-time and manually-validated maps.

Upon analyzing the provided chart, it becomes evident that the majority of features were successfully mapped in under an hour. In fact, only *Cognitive Support* and *Logging* extended beyond this time-frame. For the former, the mapping process demanded 1 hour and 37 minutes, while the latter required 3 hours and 24 minutes. Notably, the *Logging* feature's mapping alone accounts for over one-third of the total mapping duration, representing 34.4% of the overall mapping-time. A strong correlation seems to emerge between the number of manually-validated maps for a feature and its mapping-time, a connection that becomes particularly apparent in the case of *Logging*. This feature's extensive mapping-time can be attributed to its 1067 manually-validated maps, which stands as the highest count among all features. The relationship between manual validation and mapping-time is illustrated through the histogram displayed in Figure 14, which juxtaposes the count of manually-validated maps against that of automatically-validated ones for each feature.

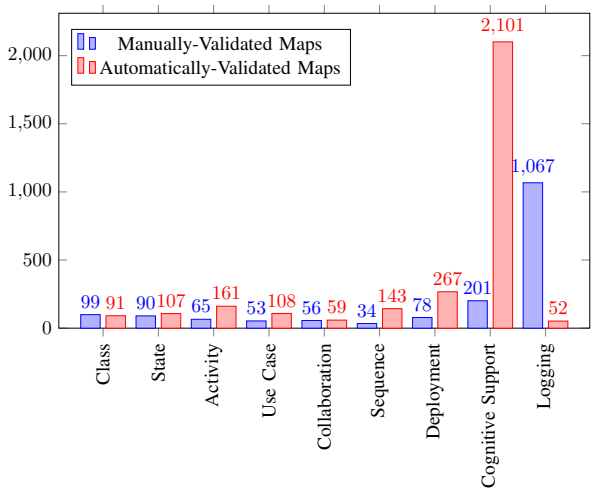


Fig. 14: Number of manually/automatically-validated maps.

The graph presents a notable disparity between the manual and automatic validations for the maps of the *Cognitive Support* feature. Notably, the manual validation of 201 maps resulted in the automatic validation of 2101 maps, nearly tenfold the manual validations. In contrast, we can observe an almost inverse relationship for *Logging*. Only 52 maps were automatically validated, while 1067 demanded manual intervention. This underlines the substantial impact of MF's automated validation. Additionally, it emphasizes that the duration of an SPL migration is heavily influenced by the balance between manually and automatically-validated maps.

V. Related Work

In this paper, we have presented an experience report on the use of the MF platform for migrating the ArgoUML individual application into an SPL. The initial effort to extract an SPL from the ArgoUML source code was proposed by *Couto et al.* [44]. While MF offers an automated mapping mechanism, *Couto et al.* employ a manual approach to locate the source code fragments related to features. Furthermore, while MF uses decorations without modifying the code, *Couto et al.* utilize compilation directives to annotate the code for each

[3]https://github.com/KarimGhallab/ArgoUML-SPL

feature, resulting in code modification. We believe that a non-destructive approach like MF is better suited for industrial constraints.

In addition, our work is based on the latest commit made to the repository at the time of writing this paper. It's important to note that the ArgoUML application has evolved between our research and the previously cited works. Those prior works date back to 2011. If we count the number of commits made from 2012 until today, we can observe that 147 commits have been applied to the ArgoUML repository. However, our examination of the ArgoUML commits leads us to speculate that the 2011 works are based on a commit made in 2008 (we speculate it is around commit `f9084b0`). This speculation is drawn from the pronounced resemblance between the application's architecture in 2008 and the one proposed by in the prior works.

The ArgoUML case study has also served as a benchmark for feature location [40]. Our work in this paper can thus complement this benchmark and be used as an anticipated outcome for endeavors related to product line construction.

Martinez et al. [45] also provided an experience report on migrating an existing application to a SPL. They conducted their study using the Robocode case study and extracted an SPL utilizing the FeatureHouse framework. At present, we are working on the process of migrating the Robocode application to an SPL using the MF platform.

Instead of migrating individual applications into SPLs, recent research work consider the migration of a collection of product variants (created using the clone-and-own approach) into SPL [46]. One of our future perspectives is to enhance MF to address this scenario.

VI. Conclusion

This paper presents our experience on using MF to migrate ArgoUML into an SPL. We first present background information about SPLs adoptions. Then, we introduce MF's vision and activities, followed by sharing our results and engaging in a discussion regarding the migration of ArgoUML using MF. We present the designed feature model and provide metrics related to the mapping of the features in the source code. Subsequently, we describe the constraints detected by MF before detailing the variants generated from the SPL using MF. Then, we give insights about the time and effort needed to map all the features into the source code. Finally, we present works related to the migration of applications into SPLs. The source code of the SPLs and their variants is available at: https://github.com/KarimGhallab/ArgoUML-SPL.
In addition to the ArgoUML case study, other examples of migrated applications are available on the official documentation of MF [32]. Noteworthy non-Java applications include: 1) Hive: A mobile application crafted using Ionic and ASP.NET Core. 2) BankWeb: A web application featuring a pure HTML/CSS/JS client and an ASP.NET Core server.

References

[1] Sven Apel, Don Batory, Christian Kstner, and Gunter Saake. *Feature-Oriented SPLs: Concepts and Implementation*. Springer Publishing Company, Incorporated, 2013.

[2] Charles W. Krueger. Easing the transition to software mass customization. In *Revised Papers from the 4th International Workshop on Software Product-Family Engineering*, PFE '01, page 282–293, Berlin, Heidelberg, 2001. Springer-Verlag.

[3] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. A survey of variability modeling in industrial practice. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-Intensive Systems*, VaMoS '13, New York, NY, USA, 2013. Association for Computing Machinery.

[4] Jabier Martinez, Tewfik Ziadi, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Bottom-up adoption of spls: a generic and extensible approach. In Douglas C. Schmidt, editor, *Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015*, pages 101–110. Association for Computing Machinery, 2015.

[5] Linda M. Northrop, Paul C. Clements, et al. A Framework for Software Product Line Practice, Version 5.0. www.sei.cmu.edu/productlines/framework.html, 2009.

[6] Frank van der Linden, Klaus Schmid, and Eelco Rommes. *Software product lines in action - the best industrial practice in product line engineering*. Springer, 2007.

[7] SEI. Product Line Hall of Fame. http://splc.net/fame.html, 2016.

[8] Sven Apel, Don S. Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines - Concepts and Implementation*. Springer, 2013.

[9] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990.

[10] Márcio Ribeiro, Paulo Borba, and Christian Kästner. Feature maintenance with emergent interfaces. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 989–1000, New York, NY, USA, 2014. Association for Computing Machinery.

[11] S. Jarzabek, P. Bassett, Hongyu Zhang, and Weishan Zhang. Xvcl: Xml-based variant configuration language. In *25th International Conference on Software Engineering, 2003. Proceedings.*, pages 810–811, 2003.

[12] Duc Le, E. Walkingshaw, and M. Erwig. #ifdef confirmed harmful: Promoting understandable software variation. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2011)*, pages 143–150, Los Alamitos, CA, USA, sep 2011. IEEE Computer Society.

[13] Cristina Lopes, Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Jean-marc Loingtier, and John Irwin. Aspect-oriented programming. *Association for Computing Machinery Computing Surveys*, 28, 10 1999.

[14] Vander Alves, Pedro Matos, Leonardo Cole, Alexandre Vasconcelos, Paulo Borba, and Geber Ramalho. *Extracting and Evolving Code in Product Lines with Aspect-Oriented Programming*, page 117–142. Springer-Verlag, Berlin, Heidelberg, 2007.

[15] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. pages 77–91, 09 2010.

[16] Ina Schaefer and Ferruccio Damiani. Pure delta-oriented programming. In *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development*, FOSD '10, page 49–56, New York, NY, USA, 2010. Association for Computing Machinery.

[17] Ina Schaefer, Lorenzo Bettini, and Ferruccio Damiani. Compositional type-checking for delta-oriented programming. In *Proceedings of the Tenth International Conference on Aspect-Oriented Software Development*, AOSD '11, page 43–56, New York, NY, USA, 2011. Association for Computing Machinery.

[18] Christian Prehofer. Feature-oriented programming: A fresh look at objects. *Lecture Notes in Computer Science*, 1241, 10 1997.

[19] D. Batory, J.N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. In *25th International Conference on Software Engineering, 2003. Proceedings.*, pages 187–197, 2003.

[20] Sven Apel, Christian Kästner, and Christian Lengauer. Language-independent and automated software composition: The featurehouse experience. *IEEE Transactions on Software Engineering*, 39(1):63–79, 2013.

[21] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in software product lines. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, page 311–320, New York, NY, USA, 2008. Association for Computing Machinery.

[22] Sven Apel, Don Batory, Christian Kstner, and Gunter Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer Publishing Company, Incorporated, 2013.

[23] Ina Schaefer, Rick Rabiser, Dave Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo, and Karina Villela. Software diversity: State of the art and perspectives. *International Journal on Software Tools for Technology Transfer*, 14, 10 2012.

[24] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[25] Jonathan Koscielny, Sönke Holthusen, Ina Schaefer, Sandro Schulze, Lorenzo Bettini, and Ferruccio Damiani. Deltaj 1.5: Delta-oriented programming for java 1.5. In *Proc. of the 2014 Int. Conf. on Principles and Practices of Prog. on the Java Platform*, PPPJ '14, page 63–74, New York, NY, USA, 2014. Association for Computing Machinery.

[26] Christian Kästner and Sven Apel. Integrating compositional and annotative approaches for product line engineering. 10 2008.

[27] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. *Mastering Software Variability with FeatureIDE*. Springer, 2017.

[28] Danilo Beuche. Industrial variant management with pure: : variants. In Carlos Cetina et al., editor, *Proceedings of the 23rd International Systems and Software Product Line Conference, SPLC 2019, Volume B, Paris, France, September 9-13, 2019*, pages 64:1–64:3. ACM, 2019.

[29] Charles W. Krueger and Paul Clements. Feature-based systems and software product line engineering with gears from biglever. In Carlos Cetina, Oscar Díaz, Laurence Duchien, Marianne Huchard, Rick Rabiser, Camille Salinesi, Christoph Seidl, Xhevahire Tërnava, Leopoldo Teixeira, Thomas Thüm, and Tewfik Ziadi, editors, *Proceedings of the 23rd International Systems and Software Product Line Conference, SPLC 2019, Volume B, Paris, France, September 9-13, 2019*, pages 66:1–66:2. ACM, 2019.

[30] Thorsten Berger, Jan-Philipp Steghöfer, Tewfik Ziadi, Jacques Robin, and Jabier Martinez. The state of adoption and the challenges of systematic variability management in industry. *Empir. Softw. Eng.*, 25(3):1755–1797, 2020.

[31] Jabier Martinez. *Exploration des variantes d'artefacts logiciels pour une analyse et une migration vers des lignes de produits*. PhD thesis, 2016. Thèse de doctorat dirigée par Le Traon, Yves et Ziane, Mikal Informatique Paris 6 2016.

[32] Mobioos Forge Official Documentation. https://documentation.mobioos.ai. Accessed: 2023-08-15.

[33] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. *Mastering Software Variability with FeatureIDE*. Springer, 2017.

[34] Jacopo Mauro. Anomaly detection in context-aware feature models. In *15th International Working Conference on Variability Modelling of Software-Intensive Systems*, VaMoS'21, New York, NY, USA, 2021. Association for Computing Machinery.

[35] Chang Hwan Peter Kim, Christian Kästner, and Don Batory. On the modularity of feature interactions. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering*, GPCE '08, page 23–34, New York, NY, USA, 2008. Association for Computing Machinery.

[36] Don Batory. Feature models, grammars, and propositional formulas. In *Proceedings of the 9th International Conference on Software Product Lines*, SPLC'05, page 7–20, Berlin, Heidelberg, 2005. Springer-Verlag.

[37] Marcilio Mendonca, Andrzej Wąsowski, and Krzysztof Czarnecki. Sat-based analysis of feature models is easy. In *Proceedings of the 13th International Software Product Line Conference*, SPLC '09, page 231–240, USA, 2009. Carnegie Mellon University.

[38] Chico Sundermann, Michael Nieke, Paul M. Bittner, Tobias Heß, Thomas Thüm, and Ina Schaefer. Applications of #sat solvers on feature models. In *15th International Working Conference on Variability Modelling of Software-Intensive Systems*, VaMoS'21, New York, NY, USA, 2021. Association for Computing Machinery.

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

[40] Jabier Martinez, Nicolas Ordoñez, Xhevahire Tërnava, Tewfik Ziadi, Jairo Aponte, Eduardo Figueiredo, and Marco Tulio Valente. Feature location benchmark with argouml spl. In *Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1*, SPLC '18, page 257–263, New York, NY, USA, 2018. Association for Computing Machinery.

[41] J.E Robbins and D.F Redmiles. Cognitive support, uml adherence, and xmi interchange in argo/uml. *Information and Software Technology*, 42(2):79–89, 2000.

[42] Matthias Kowal, Sandro Schulze, and Ina Schaefer. Towards efficient spl testing by variant reduction. In *Proceedings of the 4th International Workshop on Variability & Composition*, VariComp '13, page 1–6, New York, NY, USA, 2013. Association for Computing Machinery.

[43] Jihyun Lee, Sungwon Kang, and Danhyung Lee. A survey on software product line testing. In *Proceedings of the 16th International Software Product Line Conference - Volume 1*, SPLC '12, page 31–40, New York, NY, USA, 2012. Association for Computing Machinery.

[44] Marcus Vinicius Couto, Marco Túlio Valente, and Eduardo Figueiredo. Extracting software product lines: A case study using conditional compilation. In Tom Mens, Yiannis Kanellopoulos, and Andreas Winter, editors, *15th European Conference on Software Maintenance and Reengineering, CSMR 2011, 1-4 March 2011, Oldenburg, Germany*, pages 191–200. IEEE Computer Society, 2011.

[45] Jabier Martinez, Xhevahire Tërnava, and Tewfik Ziadi. Software Product Line Extraction from Variability-Rich Systems: The Robocode Case Study. In *Systems and Software Product Line Conference (SPLC)*, Gothenburg, Sweden, September 2018.

[46] Roberto E. Lopez-Herrejon, Jabier Martinez, Wesley Klewerton Guez Assunção, Tewfik Ziadi, Mathieu Acher, and Silvia Regina Vergilio, editors. *Handbook of Re-Engineering Software Intensive Systems into Software Product Lines*. Springer International Publishing, 2023.