Bottom-Up Adoption of Software Product Lines - A Generic and Extensible Approach

Jabier Martinez
SnT, University of Luxembourg
& LIP6, Sorbonne Universités,
UPMC Univ Paris 06, CNRS
Luxembourg
jabier.martinez@uni.lu

Tewfik Ziadi
Sorbonne Universités, UPMC
Univ Paris 06, CNRS,
LIP6 UMR 7606
Paris, France
tewfik.ziadi@lip6.fr

Tegawendé F. Bissyandé,
Jacques Klein,
Yves Le Traon
SnT, University of Luxembourg
firstName.lastName@uni.lu

ABSTRACT

Although Software Product Lines are recurrently praised as an efficient paradigm for systematic reuse, practical adoption remains challenging. For bottom-up Software Product Line adoption, where a set of artefact variants already exists, practitioners lack end-to-end support for chaining (1) feature identification, (2) feature location, (3) feature constraints discovery, as well as (4) reengineering approaches. This challenge can be overcome if there exists a set of principles for building a framework to integrate various algorithms and to support different artefact types. In this paper, we propose the principles of such a framework and we provide insights on how it can be extended with adapters, algorithms and visualisations enabling their use in different scenarios. We describe its realization in BUT4Reuse (Bottom–Up Technologies for Reuse) and we assess its generic and extensible properties by implementing a variety of extensions. We further empirically assess the complexity of integration by reproducing case studies from the literature. Finally, we present an experiment where users realize a bottom-up Software Product Line adoption building on the case study of Eclipse variants.

CCS Concepts

•Software and its engineering → Software reverse engineering; Software product lines;

Keywords

Software Product Line Engineering; Reverse Engineering; Mining existing assets

1. INTRODUCTION

Software reuse is the process of creating new software by reusing pieces of existing software rather than from scratch [28]. It may be performed simply for convenience, as in the cases of the opportunistic *copy-paste-modify* approach and the project *clone-and-own* technique, or it may represent more thoughtful solutions to complex engineering problems, as in the case of Software Product Line Engineering (SPLE). SPLE decreases time-to-market and increases product quality among other benefits, but it requires a high up-front investment [40]. In practice, the adoption of the latter, higher-level, reuse paradigm, is only considered when developers can justify opportunities of productivity and economical gains [12, 25]. In the case of SPLE, it has been reported that more than 50% of industrial practitioners formally implement a Software Product Line (SPL) only after the instantiation of several similar product variants using ad-hoc reuse techniques [6]: this is known as a *bottom-up* approach to implementing systematic software reuse.

A typical bottom-up process requires a thorough analysis of existing software artefacts. The literature proposes various methods related to various aspects of this process, also called rehabilitation process [40], and interest on providing solutions for bottom-up SPL adoption has been growing recently [3,17,33]. As we will detail in the related work section (Section 2), bottom-up SPL adoption approaches have targeted three main **objectives**:

- *Feature Identification and Analysis.* In SPLE, a Feature is defined as a prominent or distinctive characteristic, quality or user-visible aspect of a software system or systems [26]. In Feature identification, the bottom-up approach takes as input a set of artefact variants and analyses them to identify features. These features will represent optional functionalities throughout the artefact variants. Other Feature analysis approaches allow *discovering constraints* that specify the different relationships between the features. Finally, the analysis of the features and their constraints allows to *synthesize a feature model* with an appropriate feature model structure.
- *Feature Location.* In this context, the objective is to map features to their concrete implementation in the artefact variants. Therefore, compared to the Feature identification objective, the assumption is that the features are previously known and their presence or absence in the artefact variants is also known.
- *Reengineering.* This objective is the transformation phase where the artefact variants are actually refactored to conform to an SPLE approach. This includes extracting reusable assets from the artefact variants and establishing their mapping to a feature model.

Unfortunately, there are still research challenges in bottom-up SPL adoption that go beyond specific approaches or tech-

SPLC '15 Nashville, Tennessee USA
© 2015 ACM. ISBN 978-1-4503-3613-0/15/07...$15.00
DOI: http://dx.doi.org/10.1145/2791060.2791086

niques. Without detracting from the importance of specific techniques, this work focuses on the following concerns to reduce the high up-front investment for SPL adoption:

- *The proposed approaches are often related to a specific kind of software artefacts.* There is a large body of algorithms and techniques for achieving the different objectives in SPL adoption. Unfortunately, because the implementation of such algorithms are often specific to a given artefact type, it is often a barrier to re-adapt it for another kind of artefacts. There is however an opportunity to reuse the principles guiding these existing techniques for other artefacts. When such algorithms are underlying in a framework, they could be transparently used in different scenarios.
- *The absence of a unified process:* From feature identification and location until actually reengineering the artefact variants, there is the challenge of addressing, within the same environment, the three different objectives. Each approach requires inputs and provide outputs at different granularity levels and with different formats. These constraints complicate the integration of different approaches in a unified process.
- *The need for benchmarking:* The existing approaches and techniques for bottom-up SPL adoption differ on their artefact variants' analysis and manipulation. Thus, they also differ in their performance and results' quality and their assessment and comparison become a real challenge. The realization of a framework can provide a common ground for assessing and comparing different algorithms as well as comparing different ways to chain them during the different steps of bottom-up SPL adoption.

In this paper, we propose the principles for building a generic and extensible framework for SPL adoption. Thanks to the abstraction layer provided to integrate various algorithms and artefact adapters, we enable to overcome these three concerns. The contributions of this work are:

- A unifying framework to support bottom-up SPL adoption. The process promoted in the framework is built upon an analysis of the steps from the state-of-the-art on bottom-up SPL adoption.
- An intermediate model which allows to easily reuse and integrate existing specific solutions for feature identification, location, constraints discovery, feature model synthesis and reusable assets construction.
- A framework that allows domain experts to combine and compare existing approaches for each of the steps of the process.
- A framework that includes extensibility for visualisations allowing to test and share visualisation paradigms which can be reused in different SPL adoption processes.
- A realization of the framework: BUT4Reuse (Bottom-Up Technologies for Reuse) being a tool-supported bottom-up SPL adoption framework specially designed for genericity and extensibility.

The paper is structured as follows: Related work is presented early in Section 2 in order to show their influence in the proposed process as well as to differentiate our framework from previous work. Section 3 presents the framework principles and the promoted process. Section 4 presents the realization of the framework. Section 5 presents an empirical evaluation. Finally, Section 6 summarizes our work and outlines future directions.

2. RELATED WORK

SPLE is a maturing field that has witness a number of contributions, in particular in the form of frameworks for dealing with different aspects of variability management. Unfortunately, general approaches for mining existing artefacts is still not mature enough, delaying real-world SPL adoption. Through the overview of the related work from the literature, we highlight the differences with our framework.

Feature identification and location: Given several variants of a product, in order to build the feature model for SPLE, one must identify the existing features. The literature contains many approaches of feature identification and location that deal with specific artefact types. For instance, [1, 18, 27, 52, 53, 54] consider source code artefacts using different techniques, [36, 44] identify features from models, and [22] analyses function block diagrams. E.g. ECCO [18] performs feature location on artefact variants and was evaluated using Java source code artefact variants.

Clone detection techniques have been used as generic approaches to identify features in a set of artefact variants. The clone analysis workflow of ConQAT [24] supports several languages and provides extensibility for adding new types of artefacts while reusing visualisations and other analysis workflow elements. MoDisco [8] is another example of extensible framework to develop model-driven tools to support software modernization. However, these tools do not specifically target artefact variants. Instead they work for single systems and thus cannot realize the objectives of our framework. In previous work, we introduced the cross-product clone detection approach to deal with source code artefact variants using ConQAT [35]. However, the finalized tool was still not adapted to SPL adoption since feature identification is only a part of the process.

Constraints discovery: After identifying features from a set of variants, one must also infer the constraints to build a feature model that accurately capture the domain configuration space. The literature proposes approaches for mining feature constraints from existing artefacts, although they focus on specific artefact types, such as C source code [38]. Other approaches does not rely on the internal elements of the artefact variants. For example, some approaches use the existing feature combinations from the artefact variants [10, 20, 32, 34, 37] and other approaches use existing documentation [11]. Our framework, that proposes a generic and extensible approach for constraints discovery can leverage these specific techniques. Thus, these techniques can be generalized to other artefact types and they can be used simultaneously.

Feature model synthesis: Once the features and the constraints are identified, creating comprehensive feature diagrams is an NP-hard problem [46]. A feature diagram contains a hierarchy of features, enriched by cross-tree feature constraints and its structure should be heuristically defined. Some approaches rely on the constraints defined in propositional formulas [46] while others embed also ontological information of the domain [4]. WebFML [5] is a framework specialized in feature model synthesis. Our framework can integrate these approaches.

Reusable assets construction: Once the features are identified from the artefact variants, the final step towards SPL adoption is to construct the reusable assets that are associated to the features. This construction should enable the creation of new artefacts by composing or manipulating the associated reusable assets of the features. Approaches for identifying and extracting features from single software systems has been proposed [27]. In the case of artefact variants, other approaches constructed reusable assets based on source code abstract syntax trees [18, 55]. Finally, other approaches focused on defining a framework of n-way merging of models to create SPL representations [43]. An extensible framework offers the possibility for leveraging and integrating all the aforementioned approaches.

Visualisation: Some existing work in the context of SPL also consider visualisation. Tools that introduce visualisation techniques to address product configuration have been proposed [39]. Other authors propose the use of visualisation techniques to display features for pairwise testing [31]. However, our framework integrates visualisation components that help analysing the artefact variants during the SPL adoption process. In this direction, we proposed in previous work a paradigm that provides a visualisation for constraints discovery [37].

SPL generic and extensible frameworks: Other generic and extensible frameworks have been proposed tackling different aspects of SPLE. Pure::Variants [7] and Gears [29] provide variability management functionalities for specific artefacts including DOORS requirements, Microsoft Rational or Microsoft Excel spreadsheets. For each artefact type, they provide adapters. *Extensions* in Pure::Variants, and *bridges* in Gears, permit to add new adapters for new artefact types. The Common Variability Language [21] defines the variability on models and the composition of model elements in a meta-model-independent way. These tools do not tackle the reengineering process by themselves. The feature model and the reusable assets are designed and developed directly for reuse. Our framework instead is aimed at helping domain experts in the bottom-up process. However, these tools are used to leverage and manage the mined variability [23, 45].

FAMA [50] is a feature model analysis framework. It is extensible to new variability modeling languages and new reasoning operators. It also performs benchmarking for highlighting the advantages and shortcomings of different analysis approaches. FAMA's assumption is that the feature model already exists while the objective of our framework is to create it by analysing the artefact variants. It relies on an intermediate representation of feature models, while our framework's intermediate representation is related to artefact variants. FeatureHouse [2] is a generic composition framework for source code artefacts. The objectives of all these SPL frameworks are different compared to the objectives of bottom-up SPL adoption.

3. APPROACH

This section presents our framework principles and its different layers.

3.1 Principles

In order to be *generic* in the support of various artefact types, the principles of our approach is built upon the following three principles:

(1) A typical software artefact can be decomposed into distinct elements, hereafter referred to as *Elements*.

(2) Given a pair of Elements in a specific artefact type, a similarity metric can be computed for comparison purposes.

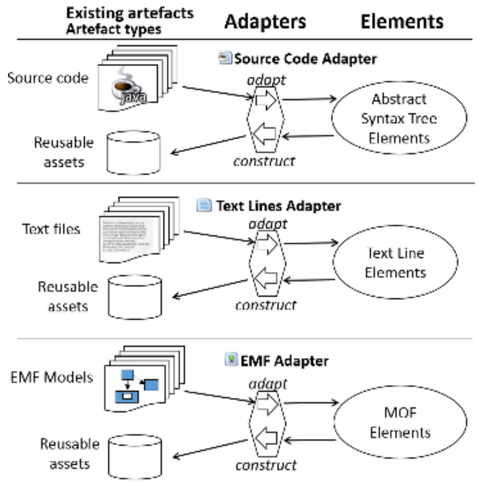


Figure 1: Artefact types examples and Elements representation creation through the adapters

(3) Given a set of Elements recovered from existing artefacts, a new artefact, or at least a part of it (which would be a reusable asset), can be constructed.

Principles (1) and (2) make it possible to reason on a set of software artefacts for identifying commonalities and variabilities, which in turn will be exploited in feature identification and location processes. Principle (3) on the other hand promises to enable the construction of the reusable assets based on the Elements found in existing artefacts.

Because our approach aims at supporting different types of artefacts, and to allow extensibility, we propose to rely on **Adapters** for the different artefact types. These Adapters are implemented as the main components of the framework. An Adapter is responsible for decomposing each artefact type into the Elements that constitute it, and for defining how a set of Elements should be constructed to create a reusable asset.

Figure 1 shows examples of Adapters dealing with different artefact types. We can see how an Adapter allows two operations, (1) to adapt the artefact to Elements and (2) to take Elements as input to construct a reusable asset for this artefact type. Source Code can be adapted to Abstract Syntax Tree Elements which capture the modular structure of source code, Text file can be adapted to Line Elements, and EMF Models can be adapted to Meta-Object Facility Elements [41] such as Class, Attribute and References. More information about Adapters are presented in section 4.1.

Designing an Adapter for a given Artefact type requires the following activities:

Activity 1: Element identification: Identifying the Elements that compose an Artefact. This will define the granularity of the Elements in a given artefact type.

Activity 2: Similarity metric definition: Defining a Similarity metric between any pair of Elements.

Activity 3: Structural constraints definition: Identifying Structural Dependencies for the Elements.

Activity 4: Reusable assets construction: Defining how to use Blocks (set of Elements) for Reusable Assets construction.

As a running example to illustrate the design of an Adapter, we will consider the following scenario: A graphic artist wants to adopt an SPL from image variants that were initially created following the *copy-paste-modify* reuse paradigm.

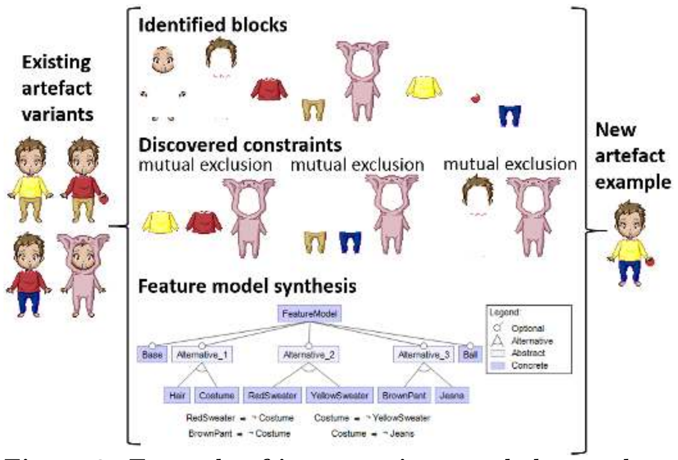


Figure 2: Example of image variants and the result of applying the Images Adapter

On the left side of Figure 2 we present the set of existing image variants.[1] In Activity 1, it was decided that the Images Adapter will adapt these variants in an intermediate representation based on Pixel Elements. The adaptation of an image into Pixel Elements will consist in loading the pixels matrix and adding the non-transparent pixels as Pixel Elements. In Activity 2, the similarity metric will consist in comparing Pixel Elements' position (cartesian coordinates), color and alpha channel. In Activity 3, each Pixel Element, will have a structural dependency with its position. This will allow to automatically discover structural constraints (e.g., it was decided that pixels overlapping is not allowed so two shirts cannot be used in the same image). In Activity 4, the construction for a set of Pixel Elements will be based in creating a transparent image with the corresponding pixels.

3.2 Bottom-Up SPL adoption Framework

Figure 3 presents the framework that we propose. We can see the bottom-up process where we have the artefact variants at the bottom of the figure and the reengineered SPL at the top. The framework builds on an architecture composed of various layers. Next paragraph will describe them and how they are related to the other layers.

Adapters: The first layer, at the bottom of Figure 3, is the *Adapters* layer. As discussed before, this layer is extensible by providing support for different artefact types. This layer enables the necessary step for creating the *AdaptedArtefacts Representation* which provides a common internal representation of the artefact variants by conforming each of them to Elements.

Blocks identification: For introducing the second layer, we present the concept of Block. A Block is a set of Elements that is obtained by comparing the artefact variants. Blocks permit to increase the granularity of the analysis by the domain experts for not to reason at Element level. This is specially relevant when trying to identify or locate features, and also during constraints discovery. In the context of Feature identification, Blocks identification represents an initial step before reasoning at feature level.

Different Block identification algorithms can be defined. In our running example, the identified Blocks shown in Figure 2, corresponded to the sets of Pixel Elements of the different "parts" of the images. The used generic algorithm was *Interdependent elements* as described in Section 4.2.

[1]Images obtained from: http://deco-kun.deviantart.com/art/Hikari-Yagami-all-outfits-in-Pixel-304142452

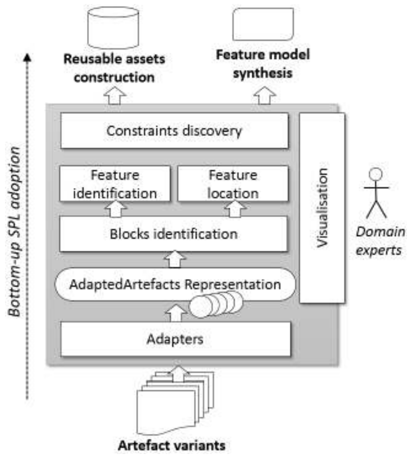


Figure 3: Layers of the framework for bottom-up SPL adoption

Feature identification and location: This layer corresponds to two different objectives as presented in Section 1. For feature identification, Blocks can be operated after the analysis. Block names can be modified by domain experts to have representative names. Also, a Block that has been identified as feature-specific can be converted into a Feature of the Feature Model. In our running example, the graphic artist decided that the Blocks will be directly assigned to Features and a meaningful name of each feature was given.

In addition, a Block can be merged with another Block or we can split the Elements of one Block in two Blocks in order to adjust it to identified features. To illustrate the usefulness of this, we present a recurrent issue when dealing with copy-paste-modified source code artefacts. This issue is the situation where a bug fix was released in a set of these artefacts but not in all of them. In this case, the Block identification algorithm could identify three different Blocks: One containing the modified statements for the bug fix, one containing the "buggy" statements, and, finally, the one that contains the shared statements that were not part of the modified statements. In this case, we would like to remove the "buggy" Block and merge the bug fix Block with the shared Block. For feature location, the reasoning on the list of known features and the identified Blocks aims to locate the features.

Constraints Discovery: The identification of constraints by mining existing assets has been identified as an important challenge for research on the SPL domain [3]. For this purpose, our framework is extensible to contribute constraints discovery approaches. In our running example, as discussed before, the Pixel Elements cannot overlap, so mutual exclusion constraints can be found as presented in Figure 2.

Feature model synthesis: Feature model creation or refinement demands a feature model synthesis approach to obtain comprehensible feature diagrams. In our running example, the identified features and constraints can be used to automatically create the feature diagram of Figure 2 by including the alternative features notation. Different approaches can be integrated in this layer.

Reusable assets construction: The framework supports the step of actually creating the reusable assets from a set of Elements. Depending on the needs, these Elements could correspond to a Block, a identified Feature, or the

Elements that corresponded to a located Feature. In the running example, the reusable assets were constructed and they correspond to the images of the identified Blocks as shown in Figure 2. On the right side of this figure we can see how the framework realizes a complete SPL adoption by creating a new artefact through the reuse of these assets.

Visualisation: Visualisation and interactive techniques reduce the complexity of comprehension tasks and helps to have in-sights and make decisions on the tackled problem [9]. The Visualisation layer, is orthogonal to all others as it is intended to present to the domain expert any relevant information yielded by any other layer. Visualisations can be used not only to show, but also to interact with the results of the different layers.

4. BUT4REUSE

Bottom-Up Technologies for Reuse (BUT4Reuse) is our realization of the presented framework. Significant efforts have been dedicated to engineering a complete tool-supported approach.[2] The assessment of the realization of the framework consists of two parts: first we assess its genericity by presenting the available Adapters, and secondly we asses its extensibility by presenting the different algorithms integrated in each of the framework's layers.

4.1 Genericity

Currently, BUT4Reuse features 10 adapters dealing with different artefact types that can be directly used or which one can build on to develop tailored or improved adapters. Table 1 presents the characteristics of each of these adapters. Despite of the different nature of the artefacts, the different similarity metrics and their construction mechanisms, they can all take benefit of the unified framework to perform bottom-up SPL adoptions.

4.2 Extensibility

Blocks identification algorithms: The algorithm for realizing Blocks identification in BUT4Reuse consists in computing *Interdependent Elements* using an approach proposed in previous work [54]. This approach is based on a formal definition of a Block that uses the notion of interdependent Elements, which is defined as follows: Given a set A of artefacts that we want to compare, two Elements (of artefacts of A) e_1 and e_2 are interdependent if and only if they belong to exactly the same artefacts of A. Therefore, e_1 and e_2 are interdependent if the two following conditions are fulfilled:

1. $\exists a \in A \;\; e_1 \in a \wedge e_2 \in a$
2. $\forall a \in A \;\; e_1 \in a \Leftrightarrow e_2 \in a$

Since interdependence is an equivalence relation on the set of Elements of A, when using this algorithm, the following definition can be provided for a Block: Given A a set of artefacts, a Block of A is an equivalence class of the interdependence relation of the Elements of A. Figure 4 illustrates this Blocks identification algorithm: Each of the n artefacts is represented as an ellipse and the rhombuses are the Elements within these artefacts. The similarity metric between Elements establishes when Elements from different artefacts are equal and therefore we can compute the intersections among them. Each of these separated intersections will be the Blocks. For example, Block 0 groups the Elements that are shared in all the artefacts, Block 4 groups those that are specific to Artefact n and Block 2 groups the Elements that are shared between Artefact 3, 4 and n.

BUT4Reuse also has integrated another Blocks identification algorithm called *Similar Elements*. Following with Figure 4, in this algorithm each rhombus will be one Block. This algorithm provides the most fine-grained granularity where one Element corresponds to one Block.

Feature identification and location: For feature identification, the domain experts will analyse the elements of each of the identified Blocks. That means that currently it is a manual process only supported by the visualisations. However, the *Interdependent Elements* Block identification

Table 1: List of available Adapters for framework genericity assessment

Text Lines Adapter: Any file that is not a folder.
Elements: Line Element. The file is read line by line.
Similarity: Levenshtein distance between strings [30]
Dependencies: None
Construct: Append the line strings to an empty file

File Structure Adapter: Any folder.
Elements: File Element and Folder Element. Pre-Order tree traversal of its structure.
Similarity: Name and relative path to the initial folder. Optionally file contents based on MD5 hashing
Dependencies: Containment dependency
Construct: Copy the resources in a given destination

Java Source Code Adapter [55]**:** A folder containing source code in java.
Elements: FSTNonTerminalNode and FSTTerminalNode. FeatureHouse [2] Java source code visitor.
Similarity: Feature Structure Tree (FST) [2] positions and names comparison
Dependencies: Source code dependencies (on-going work)
Construct: FeatureHouse extraction creating code fragments

C Source Code Adapter [55]**:** A folder containing source code in C. Same as Java Source Code Adapter but for C

EMF Models Adapter [36]**:** MOF compliant model [41]. Pre-Order model traversal of containment relations.
Elements: EMFResourceElement, EMFClassElement, EMFAttributeElement and EMFRefrenceElement
Similarity: EMF DiffMerge [14] operations
Dependencies: Container dependency of Classes, Attributes and References. Referenced elements dependencies.
Construct: Fragments or CVL realization layers [21] (on-going work)

CSV Adapter: Comma-separated values file.
Elements: CellElement. Cells' matrix visitor.
Similarity: String comparison
Dependencies: A cell depends on its row and column
Construct: A csv file the corresponding cells and empty cells in the non existing cells

Requirements Adapter: Requirements Interchange Format (ReqIF) file [42]
Elements: RequirementElement. ReqIF model visitor ProR [15]
Similarity: WUP natural Language comparison technique [51]
Dependencies: None defined, (on-going work)
Construct: Append the requirements to an empty file

Graphs Adapter: A GraphML or GML file.
Elements: VertexElement, EdgeElement. Blueprints Tinkerpop graph visitor [49]
Similarity: Label similarity
Dependencies: Vertex dependency based on the edges
Construct: Subgraphs creation in GraphML format

Images Adapter: An image file in jpg, bmp, png, gif or ico format. See Section 1

Eclipse Adapter: The folder of an Eclipse installation. See Section 5.2

[2]Download, source code and documentation: http://but4reuse.github.io/web/

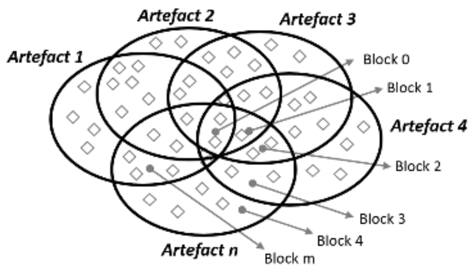


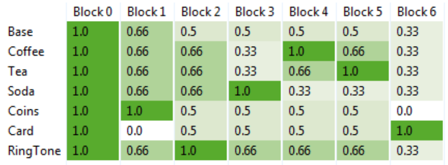
Figure 4: Illustrative example of the Interdependent Elements Blocks identification algorithm

algorithm ends up with some Blocks that can be directly associated to features. Other identified Blocks could be related to Feature intersections or noise introduced by independent evolutions of the artefact variants (e.g. bug fixes).

For feature location, the current implementation consists in trying to map the Blocks to the Features. In concrete, we provide a feature location heuristic called the *Feature-Specific heuristic* based on the idea that, for a given feature, the relevant Blocks are those that are always present when the feature is present. We calculate, for each feature and Block, the percentage of artefact variants that implementing this feature, it contains also a given Block. A percentage of 100% for a given pair of Block and Feature means that the Block *always* appear when the Feature is present in the artefacts. Figure 5 presents a matrix that relates known Features on Vending Machines artefacts to identified Blocks on this set of artefacts. For example, for feature Coffee, the location will be defined on Block 0 and Block 4 as we have 100% on them. This heuristic for feature location is highly conservative. We also provide the *Non-conservative Feature-Specific heuristic*. In this heuristic, from the set of Blocks that are always present for a feature and according to the discovered constraints, we remove the Blocks that are required from other Blocks of this set. For example, if it was discovered the structural constraint *Block 4 requires Block 0*, the feature Coffee will be located only on Block 4.

Constraints discovery: BUT4Reuse provides two strategies for discovering constraints among Blocks or Features. The first strategy, called *Binary Structural Constraints Discovery*, consists in analysing the structural dependencies between pairs of Blocks or Features. Specifically, we identify *requires* ($A \Rightarrow B$) and *mutual exclusion* ($\neg(A \wedge B)$) structural constraints by analysing the structural dependencies defined in the Elements. The *requires* constraint is defined, at Block and at Feature levels, when at least one Element from one side has a structural dependency to an Element of the other side. Formally, and being the same for Features (replacing B by F), this is the definition for the requires constraint:

$$B_1 \ requires \ B_2 \iff \exists\, e \in B_1 : \exists\, do \in e.dependencies : do \in B_2 \wedge do \notin B_1$$



	Block 0	Block 1	Block 2	Block 3	Block 4	Block 5	Block 6
Base	1.0	0.66	0.5	0.5	0.5	0.5	0.33
Coffee	1.0	0.66	0.66	0.33	1.0	0.66	0.33
Tea	1.0	0.66	0.66	0.33	0.66	1.0	0.33
Soda	1.0	0.66	0.66	1.0	0.33	0.33	0.33
Coins	1.0	1.0	0.5	0.5	0.5	0.5	0.0
Card	1.0	0.0	0.5	0.5	0.5	0.5	1.0
RingTone	1.0	0.66	1.0	0.66	0.66	0.66	0.33

Figure 5: Relation of Blocks and Features regarding their presence in the artefact variants displayed using a Heat Map visualisation

The *mutual exclusion* constraint discovery is also defined at Block and feature levels. The rationale of this discovery is that in some cases, a dependency object can only tolerate a maximum number of Elements depending on it. In our running example, the pixel position dependency object can only tolerate one Pixel Element depending on it. Another example are containment references in Classes of EMF Models where an upper bound can be defined. If the upper bound is 1 means that it is structurally invalid to try to reference 2 different containments at the same time. It can only be one or the other. Given *DO*s the set of dependency objects where $do \in DO$, and *dependencyIDs* the different types of structural dependencies where $id \in dependencyIDs$, the function *nRef* (number of references) represents the set cardinality of the subgroup of Elements in the Block that has a structural dependency with a given dependency object:

$$nRef(B_i, do, id) = |\{e : e \in B_i \wedge do \in e.getDependencies(id)\}|$$

With this definitions we find below the formula for mutual exclusion at Block level as defined by the *Binary Structural Constraints Discovery* approach when the set of Elements of the Blocks are disjoint:

$$B_1 \ excludes \ B_2 \iff \exists\, do \in DO, \exists\, id \in dependencyIDs : nRef(B_1, do, id) + nRef(B_2, do, id) > do.getMaxDependencies(id)$$

BUT4Reuse currently features another constraints discovery approach using association rule learning. Concretely we integrated the *A-Priory algorithm* as previously evaluated for this purpose in the SPLE literature [34]. In comparison to the structural constraints, that are suggested after internally analysing the variants, this approach mines the relationships of the presence or absence of the Blocks in the artefact variants.

Feature model synthesis: Currently, this step is covered by BUT4Reuse with two simple implementations. The first one just creates a *Flat feature diagram* with all the constraints included as cross-tree constraints. The second one is a heuristic called *Alternatives before Hierarchy* that is based on calculating first the Alternative constructions from the *mutual exclusion* constraints, and then create the hierarchy using the *requires* constraints. The constraints that were not included in the hierarchy are added as cross-tree constraints. Figure 2 showed the result of this heuristic in the case of only mutual exclusions. Currently, both synthesized feature models are exported to FeatureIDE [48].

Visualisations: BUT4Reuse provides a set of visualisations. The *Bars visualisation* is used for understanding how the Elements are distributed on the artefacts, how Blocks are distributed on the artefacts, how features span in the Blocks and how Blocks map the features. We have implemented these visualisations using a tool that was originally intended for visualising cross-cutting concerns in aspect oriented software development [16]. It has also been used for visualising source code clones [47] and in previous work on MoVaC (Model Variants Comparison) [36]. Figure 6 presents the visualisation that shows how Blocks are distributed across the mined artefacts. On the right side, we can see the list of identified Blocks, and on the left side we can see a bar for each of the variants. In this case, they are vending machines EMF Models. The height of each bar is proportional to the number of Elements in the artefact, and each stripe is colored accordingly to the associated Block. The order of the stripes represents the sequence of Elements returned by the adapter. In the case of EMF Models it was the result of the model tree-traversal.

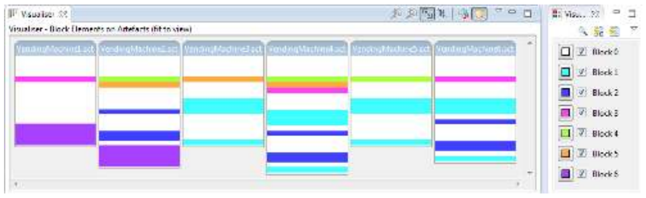


Figure 6: Visualisation showing the Blocks (colors) on the artefacts (bars)

Other currently available visualisations rely on *Heat Maps* where larger values in a matrix are represented by dark squares and smaller values by lighter squares. Figure 5, which was already introduced regarding the Feature-Specific feature location heuristic, visualises the relations between features and Blocks to help in feature location.

BUT4Reuse also provides four kinds of *Graph visualisations*. First of them is a graph where the nodes are all the Elements of the analysis and the edges are the dependency relations between them. An example of this graph is presented in Section 5.2.1 at Figure 7. The second one is a graph where the nodes are the identified Blocks and the edges are the identified constraints between them. The third one is a graph where the nodes are the features that want to be located and the edges are the identified constraints between them. In all these graphs, nodes and edges are labelled with different attributes for easy graph manipulation. Finally, the fourth graph corresponds to the Feature Relations Graphs (FRoGs) visualisation paradigm [37].

5. EMPIRICAL EVALUATION

In this section we discuss the usage in practice of the realization of the framework in BUT4Reuse. First, we quantitatively evaluate the effort for integrating new algorithms and adapters. Then, we detail an SPL adoption scenario building on the case study of Eclipse variants. We perform a qualitative evaluation in a controlled experiment scenario with Master students that designed and developed the Eclipse Adapter. We present also the results of the usage of this Adapter to discuss the benefits of an extensible framework.

5.1 Development and Integration Complexity

Based on our experience, we present the development time and lines of code (LOC) metrics concerning the development or integration of adapters and algorithms.

The Text Lines, File Structure, CSV, Graphs, and Images adapters are respectively made of 177, 207, 210, 274 and 230 LOC. Besides, each of them has been implemented in less than one day by an experienced developer. The C and Java Source Code adapter has been realized by integrating ExtractorPL [55]. The integration of this adapter took about one work-day and consists of 930 LOC. The integration of the EMF Models adapter was borrowed from MoVaC approach [36]. MoVaC implementation, before the integration, took seven days. Its integration took one day and consists of 499 LOC. The short time to integrate ExtratorPL and MoVaC can be justified because we were the developers of these previous works and both are also based on the principle of decomposing the artefacts in elements.

The *Interdependent Elements* Blocks identification algorithm was borrowed from [54] and integrated in one day. From MoVaC, the *Bars Visualisation* was integrated in one day. After these integrations we reproduced the same case studies presented in previous work [36, 54, 55]. The *Binary Relations constraints discovery* consists of 157 LOC and the *A-Priori* 154 LOC using the Weka data mining library [19].

As presented before, the development burden for a typical adapter is small in terms of LOC. This is because 1) BUT4Reuse Core implementation provides the dedicated extension points to ease the work of the Adapter developer and 2) the Adapters can rely on off-the-shelf libraries for the manipulation of the targeted artefact types including its decomposition, similarity calculation or construction.

5.2 SPL adoption scenario with BUT4Reuse

We consider the SPL adoption scenario of Eclipse variants.[3] Eclipse [13] is an integrated development environment that provides tool-sets for a wide range of software development needs. Different predefined tool-sets for targeting specific needs are distributed. The Kepler version of Eclipse has 12 default official distributions: *Standard*, *Java EE*, *Java*, *C/C++*, *Scout*, *Java and DSL*, *Modeling Tools*, *RCP and RAP*, *Testing*, *Java and Reporting*, *Parallel Applications* and *Automotive Software*.[4] In the context of this evaluation, we will refer to them as variants. We targeted the adoption of an SPL approach for initializing a personalized Eclipse. Currently, the initialization of an Eclipse consists in selecting one of these default distributions and then manually installing the desired extra functionalities. For example, if we want a java development environment with development utilities for functional testing, we will probably select the *Java* distribution and then install the *Jubula* feature using the Kepler update site. In addition, Eclipse is a plugin-based architecture and its customization sometimes requires dealing with complex dependency checks.

The nature of the Eclipse artefact is based on a root folder that contains the executable and a set of folders and configurations files. Two relevant folders are the *plugins* folder that contains the installed plugins, and the *features* folder that contains information about the features present in the variant. We ignored, on purpose, the information that the features could provide and we used it only for discussing feature location results. The case study of Eclipse is specially suitable for evaluating feature location approaches given that we can compare the results of BUT4Reuse with the actual features defined for Eclipse. However, evaluating specific techniques is out of the scope of the paper.

5.2.1 *Eclipse adapter design and implementation*

A BUT4Reuse adapter for Eclipse was designed and implemented in 3 weeks of development by a group of 8 master students that received a formation of 6 hours on the BUT4Reuse principles. They followed the activities presented in Section 3.1.

Elements identification: The Elements that compose an Eclipse artefact are the Plugin Elements (the plugins) and the File Elements (all the files of an Eclipse installation). In this case, Plugin Element is an extension of File Element. In order to decompose an artefact, the Adapter will perform a tree traversal of the Eclipse root folder.

Similarity metric definition: The similarity between Plugin Elements was implemented by comparing the plugin ids while the similarity between File Elements was imple-

[3]Reproducibility of the Eclipse scenario using BUT4Reuse: http://github.com/but4reuse/but4reuse/wiki/SPLC2015

[4]http://eclipse.org/downloads/packages/release/Kepler/SR2

mented by comparing their relative URI resolved with respect to the Eclipse root folder.

Structural dependencies identification: Each plugin depends on the required plugins defined in its bundle manifest declaration. Concretely, we considered the static non-optional dependencies defined in the *Require-Bundle* set. Therefore the Plugin Element will structurally depend on the Plugin Elements of these plugins. The id assigned to this dependency type is *requiredBundle*. The File Elements depend on their corresponding parent File Element and the defined dependency type's id is *container*. None of these dependency types need to establish an upper bound given that, in practice, a plugin can be imported from any number of plugins and a folder can contain any number of files.

Reusable assets construction: The construction was implemented by copying the plugins and files associated to each Block. As part of this construction, the *bundles.info* configuration file was adjusted if an Eclipse installation is being constructed. This final adjustment leads to completely functional Eclipses created through systematic reuse.

The Eclipse adapter consists of 471 LOC and it is publicly available and integrated in BUT4Reuse. The limitations of the design decisions presented before are 1) we only identify Plugin Elements that are in the Eclipse plugins folder. Other mechanisms in Eclipse exist like *dropins* folder or *bundle pooling* mechanism. However, these mechanisms are not used in the Eclipse official distributions so it was not relevant. 2) The similarity metric between Plugin Elements does not consider plugin versions. That means that two plugins with the same ids but different versions will be considered the same Plugin Element. This situation happened only in the case of 23 plugins, and only in 2 of them there were major version changes. Finally, 3) other methods to define structural dependencies such as *Import-Package* or *x-friends* are not considered.

Lessons learned: We consider that the learning curve in the framework principles and the basic usage of BUT4Reuse (6 hours) is acceptable. The students started to discuss in terms of Elements and Blocks quickly. We have experienced also that the effort is considerably bigger in the design of the Adapter than in the implementation itself. In fact, before starting the implementation, the students needed to obtain an in-depth knowledge of many aspects of Eclipse that is not common in the average user of Eclipse. This corroborated our experience in other Adapters where defining the granularity of the Elements, defining the similarity metric or identifying how to get the information of the dependencies were also a difficult decision-making process with many trade-offs.

5.2.2 Results

This section reports the results of the different layers defined in Section 3.2 and discusses the implications of the layers' extensibility. The reported performance in execution time are the average of 10 executions calculated using a laptop Dell Latitute E6330 with a processor Intel(R) Core(TM) i7-3540M CPU @3.00GHz 3.00GHz, 8GB RAM, with Windows 7 64-bit. We used the 12 Eclipse Kepler SR2 Windows 64-bits distributions as artefacts. The average number of Elements per artefact is 1483. The Eclipse Adapter developed by the students takes 11 seconds to decompose the 12 Eclipse variants into Plugin and File Elements that correspond to the Adapted Artefacts representation.

Block identification: We used the Interdependent Elements algorithm for Block identification. 61 Blocks were identified. The average of Elements per Block was 68. The Blocks identification algorithm took only 62 milliseconds.

Feature identification: For this step, we requested the expertise of 3 domain experts with more than ten years of experience on Eclipse development, who analysed, independently from each other, the Elements' textual representations of the 61 Blocks. They were able to manually identify Features by guessing the functionality that the Blocks can provide. Further, they were able to select a name for this functionality. The Feature identification process was possible with an average of 87% of the Blocks assigned to a named Feature. This manual task took an average of 51 minutes. We manually analysed the reported names of all the Blocks and their comments. Regarding the coincidences in the names: 56% of the Blocks were equally named by the 3 domain experts. In 33% of the Blocks there were coincidences in two of them. That ends up with an 11% where there was no coincidence. Also, not all the Blocks were easy to name: In 18% of them, at least one domain expert was not able to put a name. According to their comments, the reasons were 1) completely ignoring the plugins or 2) the plugins inside a Block had no evident relations among them. Regarding this second point, where the Block presents a mix of functionalities without evident relation, this is a known and discussed limitation of the Interdependent elements algorithm [54]. Also, 8% of the Blocks corresponded to plugins that are libraries. The 3 domain experts commented that these Blocks cannot be considered as features but as support for features. Another 5% of the Blocks were considered irrelevant from a functional perspective given that they completely consist of source code plugins that were found in some distributions but not in others.

Feature location: In this context, feature location consists in locating the plugins associated to each Eclipse feature. For assessment purposes, we programatically mined the Eclipse features of all the Eclipse distributions by getting the information from their features folder. 437 Eclipse different features are present in the distributions which represent a significantly bigger number than the 61 identified Blocks from the 12 variants. We included the information of which Eclipse features are present in each variant and we run the two available Feature location approaches to map Eclipse features to Blocks. We compared the plugins of the Eclipse features with the plugins from the Blocks of the located features. In order to calculate the plugins of the Eclipse features we mined each feature declaration (*feature.xml*) considering its declared plugins. We calculated the precision and recall of the conservative and non-conservative Feature-Specific heuristic. We evidenced a trade-off. We loss recall in the non-conservative compared to the conservative but we gain precision. This showed the advantages and disadvantages of selecting the Feature location algorithm. The proposed framework allows users to make decisions regarding the trade-offs of the different algorithms.

Constraints discovery: We used the Binary Structural constraints discovery algorithm on the 61 Blocks and 74790 structural constraints were discovered. That demonstrated how highly interconnected the Eclipse plugins are. The analysis took 88 seconds. We also used the A-Priory association rules (with a limit of 30000 rules to prevent current stack overflow issues in the algorithm). The analysis took 0.5 sec-

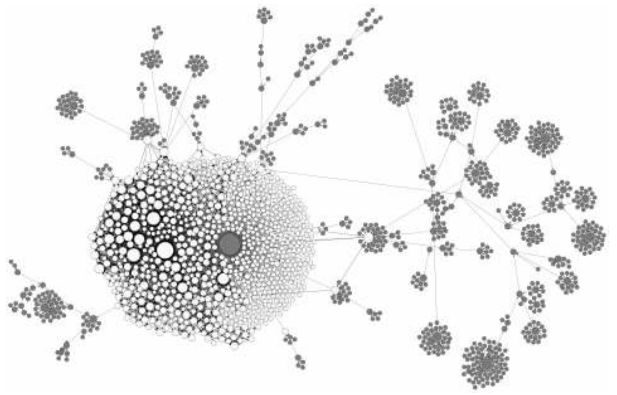


Figure 7: An Eclipse distribution decomposed in Plugin Elements (white) and FileElements (grey) with the structural dependencies between them

onds. This algorithm discovered also *excludes* constraints that are not expected to be true in the context of the analysis of Eclipse features. This algorithm is conservative in the sense that it prevents Block combinations that are not part of the existing variants. Again, there are trade-offs of using one algorithm or other. The A-Priory algorithm, that does not reason on the Elements' structural dependencies, is a more conservative approach against possible semantic constraints among the features. A user of our proposed framework may decide that selecting this algorithm is not appropriate for the Eclipse variants scenario.

Reusable assets construction: Each of the 61 Blocks were constructed separately. The reusable asset consists of a set of files that, if integrated in an Eclipse, can provide some functionality. We evaluated the validity of the reusable assets by re-constructing the 12 Eclipse variants. We compared the file structure from the original and the re-constructed and they were the same excluding few cases because of the mentioned limitation of not considering different plugin versions. After manually solving these versioning problems, we manually checked that the Eclipse artefacts were functional and the plugins can be started without dependency issues. We further generated non-existent variants from structurally valid configurations according to our discovered constraints. For example, we generated an Eclipse with only the core Block and another one with all the possible Blocks. We created other Eclipse with the core and CVS versioning system support. We created another with the union of the Blocks corresponding to the Java and the Testing Eclipse variants.

Feature model synthesis: The Blocks were renamed during feature identification. After that, using the two available feature model synthesis approaches, the Flat feature diagram and the Alternatives before Hierarchy heuristic, we created two different feature diagrams. In the case of the second one, the hierarchy was very limited because of the highly interconnected Blocks. The presence of the cross-tree constraints were more prominent given that classical feature diagrams only support one parent feature.

Visualisation: Figure 7 shows an example of the visualisation. Concretely it presents the result of the Graph visualisation of the Elements of one Eclipse distribution (concretely the Modeling distribution). The white nodes are Plugin Elements and the grey nodes are File Elements. The edges correspond to the structural dependencies. The size of the nodes are related to the number of Elements that depends on this node (in-degree). This visualisation helped to understand Eclipse structure and peculiarities in terms of Elements. The biggest grey File Element node corresponds to the mentioned *plugins* folder that contains all the plugins. The biggest white Plugin Element node corresponds to the *org.eclipse.core.runtime* plugin which is the foundation of the Eclipse platform. On the left side of the plugins File Element we can see a set of highly interconnected plugins while on the right side we can see a set of plugins with small or no dependencies to other Plugin Elements. This phenomena corresponds to the *requiredBundle* dependency type. We can see also the tree structures of the File Elements dependencies of the *container* dependency type.

6. CONCLUSIONS

We have introduced a generic and extensible framework for a bottom-up approach to SPLE. We presented its principles with the objective to reduce the current high up-front investment required for a systematic reuse end-to-end adoption. The framework (1) can be easily adapted to different artefact types and it (2) can integrate state-of-the-art algorithms and visualisation paradigms to help in this process. We have presented Bottom-Up Technologies for Reuse (BUT4Reuse) that is our realization of the framework. We demonstrated the generic and extensible characteristics of this realization by presenting a variety of ten adapters to deal with diverse artefact types. We also demonstrated the extensibility with algorithms and visualisation paradigms that have already been integrated to provide a complete solution. We empirically evaluated the framework integration and development complexity, and its usage, in the scenario of adopting an SPL approach from existing Eclipse variants.

As further work, apart from the improvement or proposition of concrete algorithms, there are still many challenges on generecity and extensibility. Nowadays software does not rely on only one type of artefact. For example, a software project uses to contain, simultaneously, requirements, design models, source code or test suites. We aim to assess the framework in covering, in combination, different types of artefacts. Also, extensibility in the layers of the framework creates the need of defining guidelines for different scenarios to select the most appropriate extensions.

7. ACKNOWLEDGMENTS

The present work is supported by the National Research Fund Luxembourg (FNR), under AFR 7898764 and MODEL C12/IS/3977071. The work of Tewfik Ziadi was supported by the University of Luxembourg as a visiting researcher.

8. REFERENCES

[1] M. H. Alalfi, E. J. Rapos, A. Stevenson, M. Stephan, T. R. Dean, and J. R. Cordy. Semi-automatic identification and representation of subsystem variability in simulink models. In *ICSME*, 2014.

[2] S. Apel, C. Kästner, and C. Lengauer. FEATUREHOUSE: Language-independent, Automated Software Composition. In *ICSE*, 2009.

[3] W. K. G. Assunção and S. R. Vergilio. Feature location for software product line migration: A mapping study. In *SPLC Volume 2*, 2014.

[4] G. Bécan, M. Acher, B. Baudry, and S. Ben Nasr. Breathing Ontological Knowledge Into Feature Model Synthesis: An Empirical Study. *Empirical Software Engineering*, page 51, 2015.

[5] G. Bécan, S. Ben Nasr, M. Acher, and B. Baudry. WebFML: Synthesizing Feature Models Everywhere. In *SPLC*, 2014.

[6] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wasowski. A survey of variability modeling in industrial practice. In *VaMoS*, 2013.

[7] D. Beuche. Modeling and building software product lines with pure::variants. In *SPLC Volume 2*, 2010.

[8] H. Bruneliere, J. Cabot, G. Dupé, and F. Madiot. Modisco: A model driven reverse engineering framework. *Information & Software Technology*, 56, 2014.

[9] S. K. Card, J. D. Mackinlay, and B. Shneiderman, editors. *Readings in Information Visualization: Using Vision to Think*. Morgan Kaufmann Publishers Inc., 1999.

[10] K. Czarnecki, S. She, and A. Wasowski. Sample spaces and feature models: There and back again. In *SPLC*, 2008.

[11] J.-M. Davril, E. Delfosse, N. Hariri, M. Acher, J. Cleland-Huang, and P. Heymans. Feature model extraction from large collections of informal product descriptions. In *FSE*, 2013.

[12] D. Domis, S. Sehestedt, T. Gamer, M. Aleksy, and H. Koziolek. Customizing domain analysis for assessing the reuse potential of industrial software systems: Experience report. In *SPLC*, 2014.

[13] Eclipse. Eclipse integrated development environment. http://eclipse.org, 2015.

[14] Eclipse. Emf diff/merge: a diff/merge component for models. http://eclipse.org/diffmerge, 2015.

[15] Eclipse. Requirements Engineering Platform. http://eclipse.org/rmf/pror/, 2015.

[16] Eclipse. The visualiser, AJDT: AspectJ Development Tools. http://www.eclipse.org/ajdt/visualiser, 2015.

[17] W. Fenske, T. Thüm, and G. Saake. A taxonomy of software product line reengineering. In *VaMoS*, 2014.

[18] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed. Enhancing clone-and-own with systematic reuse for developing software variants. In *ICSME*, 2014.

[19] M. A. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA data mining software: an update. *SIGKDD Explorations*, 11(1), 2009.

[20] E. N. Haslinger, R. E. Lopez-Herrejon, and A. Egyed. On extracting feature models from sets of valid feature combinations. In *FASE*, 2013.

[21] Ø. Haugen, A. Wasowski, and K. Czarnecki. CVL: common variability language. In *SPLC*, 2013.

[22] S. Holthusen, D. Wille, C. Legat, S. Beddig, I. Schaefer, and B. Vogel-Heuser. Family model mining for function block diagrams in automation software. In *SPLC Volume 2*, 2014.

[23] H. P. Jepsen, J. G. Dall, and D. Beuche. Minimally invasive migration to software product lines. In *SPLC*, 2007.

[24] E. Juergens, F. Deissenboeck, and B. Hummel. Clonedetective - a workbench for clone detection research. In *ICSE*, 2009.

[25] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *ICSE*, 2009.

[26] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, 1990.

[27] C. Kästner, A. Dreiling, and K. Ostermann. Variability mining: Consistent semiautomatic detection of product-line features. *IEEE Transactions on Soft. Eng.*, 40(1), 2014.

[28] C. W. Krueger. Software reuse. *ACM Comput. Surv.*, 24, 1992.

[29] C. W. Krueger and P. C. Clements. Systems and software product line engineering with BigLever software gears. In *SPLC Volume 2*, 2012.

[30] V. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10, 1966.

[31] R. E. Lopez-Herrejon and A. Egyed. Towards interactive visualization support for pairwise testing software product lines. In *VISSOFT*, 2013.

[32] R. E. Lopez-Herrejon, J. A. Galindo, D. Benavides, S. Segura, and A. Egyed. Reverse engineering feature models with evolutionary algorithms: An exploratory study. In *SSBSE*, 2012.

[33] R. E. Lopez-Herrejon, T. Ziadi, J. Martinez, and A. K. Thurimella. Second international workshop on reverse variability engineering (REVE 2014). In *SPLC*, 2014.

[34] A. Lora-Michiels, C. Salinesi, and R. Mazo. A method based on association rules to construct product line models. In *VaMoS*, 2010.

[35] J. Martinez and A. K. Thurimella. Collaboration and source code driven bottom-up product line engineering. In *SPLC Volume 2*, 2012.

[36] J. Martinez, T. Ziadi, J. Klein, and Y. L. Traon. Identifying and visualising commonality and variability in model variants. In *ECMFA*, 2014.

[37] J. Martinez, T. Ziadi, R. Mazo, T. F. Bissyandé, J. Klein, and Y. L. Traon. Feature relations graphs: A visualisation paradigm for feature constraints in software product lines. In *VISSOFT*, 2014.

[38] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki. Mining configuration constraints: Static analyses and empirical results. In *ICSE*, 2014.

[39] D. Nestor, S. Thiel, G. Botterweck, C. Cawley, and P. Healy. Applying visualisation techniques in software product lines. In *SOFTVIS*, 2008.

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

[41] OMG. Meta Object Facility (MOF) Core Specification. http://www.omg.org/spec/MOF/2.0/, 2006.

[42] OMG. Requirements Interchange Format (ReqIF). http://www.omg.org/spec/ReqIF, 2013.

[43] J. Rubin and M. Chechik. N-way model merging. In *FSE*, 2013.

[44] J. Rubin and M. Chechik. A survey of feature location techniques. In *Domain Engineering, Product Lines, Languages, and Conceptual Models*. 2013.

[45] J. Rubin, K. Czarnecki, and M. Chechik. Managing cloned variants: A framework and experience. In *SPLC*, 2013.

[46] S. She, U. Ryssel, N. Andersen, A. Wasowski, and K. Czarnecki. Efficient synthesis of feature models. *Information and Software Technology*, 56(9), 2014.

[47] R. Tairas, J. Gray, and I. Baxter. Visualization of clone detection results. In *OOPSLA Workshop on Eclipse Technology eXchange*, 2006.

[48] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming*, 79(0), 2014.

[49] Tinkerpop. TinkerPop3: A Graph Computing Framework. http://blueprints.tinkerpop.com, 2015.

[50] P. Trinidad, D. Benavides, A. R. Cortés, S. Segura, and A. Jimenez. FAMA framework. In *SPLC*, 2008.

[51] Z. Wu and M. Palmer. Verbs semantics and lexical selection. *Proceedings Association for Computational Linguistics*, 1994.

[52] Y. Xue, Z. Xing, and S. Jarzabek. Feature location in a collection of product variants. In *WCRE*, 2012.

[53] K. Yoshimura, F. Narisawa, K. Hashimoto, and T. Kikuno. Fave: factor analysis based approach for detecting product line variability from change history. In *MSR*, 2008.

[54] T. Ziadi, L. Frias, M. A. A. da Silva, and M. Ziane. Feature identification from the source code of product variants. In *CSMR*, 2012.

[55] T. Ziadi, C. Henard, M. Papadakis, M. Ziane, and Y. L. Traon. Towards a language-independent approach for reverse-engineering of software product lines. In *SAC*, 2014.