Reverse Engineering Feature Models in the Real

Guillaume Becan

Master research Internship

Internship report

Reverse Engineering Feature Models in the Real

From February 1st to June 28th, 2013

Author:
Guillaume Bécan

Supervisors:
Benoit Baudry
Mathieu Acher
Triskell team

Abstract

Numerous organizations face the need to develop highly configurable systems with thousands of functionalities and variable subsystems. Feature Models (FMs) are a popular formalism for modeling and reasoning about configurations of a system. As the manual construction or management of an FM is time-consuming, error-prone and not realistic for large software projects, automated operations for reverse engineering, slicing, diff, merging or refactoring FMs have been developed. With a basic strategy or without prior knowledge, these operations are likely to compute meaningless ontological relations (as defined by the hierarchy and feature groups) between features which may cause severe difficulties when reading, maintaining or exploiting the resulting FM. In this paper we address the problem of synthesizing an FM both valid w.r.t. a set of logical dependencies (representing a set of configurations) while having an appropriate ontological semantics. We define a generic procedure and evaluate a series of heuristics for clustering and weighting the syntactic and semantic relationships between feature names to assist users in selecting a hierarchy and feature groups. We also present an interactive environment that use our techniques and offer a way to include the user's knowledge. The experiments on hundreds of realistic FMs show that our tool-supported procedures effectively reduce the information a user has to consider during the synthesis of an FM. This work is a necessary step for reverse engineering feature models in the real, i.e., in realistic projects whose variability is scattered in numerous artefacts and where developers may not have a global view of the project.

Keywords

Software product line, variability modeling, feature model, reverse engineering, semantic similarity.

Acknowledgements

I would like to thank my supervisors Mathieu Acher for his advice and time during the internship and Benoît Baudry for sharing his experience and for guiding our research. I also thank Sana Ben Nasr, Ph.D. student, for her help during the development of the tool and the writing of the articles derived from the work presented in this report. Finally, I thank the other members of the Triskell team for their hospitality and the discussions that lead us to some ideas in this work.

Contents

1 Introduction

In a wide range of domains, numerous organizations need to extend, change, customize and configure multiple similar software products in order to meet the specific requirements of their customers . For instance, the Linux kernel offers more than 10,000 configuration options, the Eclipse project offers thousands of plugins, and the industry (e.g., automotive) faces the need to develop highly configurable systems with thousands of functionalities and variable subsystems [15,24,27,32,45].

Real world success stories of *Software Product Lines* (SPLs) show that the effective management of a large set of products is possible [2,39]. Central is the factorization and exploitation of common features of the products as well as the handling of their variability [9,45]. Thousands of features and many logical dependencies among them are observed in large scale open source or industrial SPLs. This complexity poses a challenge for both developers and users of SPLs.

Feature Models (FMs) are by far the most popular notation for modeling and reasoning about an SPL [15]. FMs offer a simple yet expressive way to define a set of legal *configurations* (i.e., combinations of features) each corresponding to a product of an SPL [5, 31, 46, 57]. Another important and dual aspect of an FM is the way features are conceptually related. A tree-like hierarchy and feature groups are notably used to organize features into multiple levels of increasing detail and define the *ontological semantics* [29] of an FM.

A manual elaboration of an FM is not realistic for large complex projects or for legacy systems. Many procedures have been developed to reverse engineer dependencies and features' sets from existing software artefacts – being source code, configuration files, spreadsheets or requirements [3, 4, 32, 47–50, 52, 58, 60] (see Figure 1, left). From these logical dependencies (typically formalized and encoded as a propositional formula in conjunctive or disjunctive normal form), numerous FMs can represent the exact same set of configurations, out of which numerous candidates are obviously not maintainable since the retained hierarchy is not adequate [6,52]. In many FM-based activities [3,10,17,27,30,36,41], the bad quality of the model may pose severe problems for a further exploitation by automated transformation tools or by stakeholders.

This report[1] addresses the following problem: *How to automate the synthesis of an FM from a set of constraints while both addressing the configuration and ontological semantics?* Existing synthesis techniques for FMs have been proposed, mostly in the context of reverse engineering FMs, but they neglected either configuration or ontological aspects of an FM [3,4,8,31,34,37, 50,60]. As a consequence, the resulting FM may be of poor quality despite the significant effort already spent in reverse engineering the propositional formula. Firstly, from a configuration perspective, the FM may expose to customers configurations that actually do not correspond to any existing product [11, 17, 23, 30, 56]. Secondly, from an ontological perspective, a naive hierarchical organization may mislead the meanings of the features and their relationships. It may dramatically increase the stakeholders' effort when communicating among others or when understanding, maintaining and exploiting an FM (e.g., see Figure 5). Other works proposed some support but users have to manually choose a relevant hierarchy among the thousands of possibilities [6,37], which is both error-prone and time-consuming. A notable exception is She et al. [52] who proposed a procedure to rank the correct parent features in order to reduce the task of a user. However they assume the existence of artefacts describing in a precise manner features, only consider possible parent-child relationships, and the experimented heuristics are specific to the operating system domain.

The main contribution of the report is the definition and evaluation of generic heuristics

[1] This report is an extension of the following submitted research paper (under review): Guillaume Bécan, Sana Ben Nasr, Mathieu Acher and Benoit Baudry. Breathing Ontological Knowledge Into Feature Model Management. In the *28th IEEE/ACM International Conference on Automated Software Engineering*. 2013.

for selecting a feature hierarchy and feature groups relevant both from a configuration and ontological perspective. By addressing the FM synthesis problem, we not only advance the state-of-the-art of reverse engineering FMs but we also equip important management operations of FMs – like slicing, merging, diff, refactoring – with ontological capabilities since all are based on the same synthesis procedure (see Figure 1).

We exploit a so-called implication graph, a structure that can be obtained from logical dependencies – the rationale is that all valid feature hierarchies are necessarily a spanning tree of the graph making our technique sound for *breathing ontological knowledge* into FM synthesis. We then develop a procedure that tries to maximize closest similarities between features when selecting the spanning tree. For this purpose, both parent-child and sibling relationships are computed and exploited. We describe different alternative techniques that can be combined together for clustering and weighting the syntactic and semantic relationships between features. The heuristics rely on general ontologies (e.g., from Wordnet or Wikipedia) and are applicable without prior knowledge or artefacts. We embedded the heuristics and the procedures in an interactive environment that offers a preview of the FM under construction and other features that helps the user in synthesizing a FM.

We evaluate our proposal on 124 FMs for which we have a ground truth. The experiments show that the number of parents to choose decrease by 10% to 20% in the best case and for at least 45% of the features, the user has to consider only 2 parent candidates. Our tool-supported procedures can reduce the information a user has to consider during the synthesis of an FM and open avenues for a practical usage in reverse engineering or maintenance scenarios.

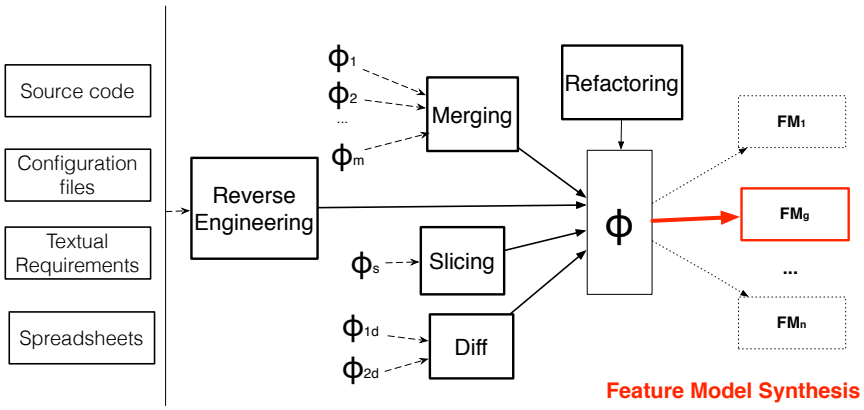


Figure 1: FM synthesis: (1) a key issue for automated operations (2) numerous FMs conformant to $[\![\phi]\!]$ exist but have an inappropriate ontological semantics

The report is organized as follows: Section 2 introduces the context of software product lines and feature models. Section 3 describes the challenges when synthesizing FMs and the limits of the related work. Section 4 present our techniques to overcome these challenges by using heuristics based on ontologies and algorithms to compute the best hierarchy for an FM. Section 5 present our interactive environment for FM synthesis. Section 6 evaluates the effectiveness of our techniques on 124 FMs. Finally Section 7 concludes this report and opens new perspectives for future work.

2 Context

2.1 Software Product Line

Organizations face new challenges when developing software. They no longer develop a software with one programming language, one platform, one operating system or in one language. They target each customer with an individualised product thus developing multiple similar products. This process is called mass customisation in contrast to mass production which targets all the customers with a few products [45].

For instance, a Linux-based operating system would not embed the same components of the kernel for a server, a desktop computer or a mobile phone because their purposes and requirements are different.

To offset the increasing development cost caused by the multiplication of products, organizations must adapt their development techniques. Classic techniques that aim at producing only one software product are inefficient in the context of mass customisation.

Software Product Lines (SPLs) address these problems by producing multiple similar products from a common base. The idea is to include variability in the development of a set of products. Software variability is "the ability of a software system or artefact to be efficiently extended, changed, customized or configured for use in a particular context" [54]. Such variability allows the software artefacts to be adapted and shared by different products.

Apart from reducing cost, SPLs enable other benefits: reusing software artefacts to derive new products, developing tests for a software artefact that validate every product, certifying the entire SPL instead of certifying it product by product or centralizing the management effort [45].

Such benefits are only possible if the variability is actively managed during the life cycle of the SPL. This is where variability modeling is important.

2.2 Variability Modeling

The main objectives of variability modeling are reasoning about an SPL and generate multiple artefacts. As other models, a variability model helps handling the complexity of an SPL. Thanks to an unambiguous and formal semantics, it helps reasoning on existing products, automatically checking their consistency or deriving new products. Moreover, Berger et al. showed that variability modeling is used in a wide range of domains in the industry and that the management of existing variability is its main benefit [15].

From such variability model, we can generate multiple artefacts like source code, tests, packages, documentation, recommendation systems, configurators or other models that help developers and users in their tasks [27].

Modeling the variability of an SPL consists in defining the commonalities and variabilities (i.e., differences) of its products. Several formalisms emerged to model variability. Among them, feature modeling (FM) and decision modeling (DM) are the most popular. Czarnecki et al. compared them and concludes that "the main difference between FM and DM is that FM supports both commonality and variability modeling, whereas DM focuses exclusively on variability modeling". They also observed a significant convergence between these two formalisms [25]. For this reason, we chose to focus on feature modeling.

There exist several definitions of a feature [22]. A feature can be "a prominent or distinctive user-visible aspect, quality or characteristic of a software system or systems" [38] or "an elaboration or augmentation of an entity that introduces a new service, capability or relationship" [13]. It can even be "anything users or client programs might want to control about a concept" [27].

Therefore, a feature and consequently a feature model can cover every aspect of a software from user requirements to technical details of the implementation.

2.3 Feature Model

Feature Models aim at characterizing the valid combinations of features (a.k.a. configurations) of a system under study. A feature hierarchy, typically a tree, is used to facilitate the organization and understanding of a potentially large number of concepts (features).

2.3.1 Syntax

Different syntactic constructions are offered to attach variability information to features organized in the hierarchy (see Definition 1). When decomposing a feature into subfeatures, the subfeatures may be optional or mandatory or may form *Mutex*, *Xor*, or *Or* groups. As an example, the FM of Figure 2 describes a family of Wiki engines. The FM states that Wiki has three *mandatory* features, Storage, Hosting and License and one *optional* feature Programming Language. There are two alternatives for Hosting: Hosted Service and Local features form an *Xor*-group (i.e., at least and at most one feature must be selected). Similarly, the features Open Source and Proprietary License form an *Xor*-group of License.

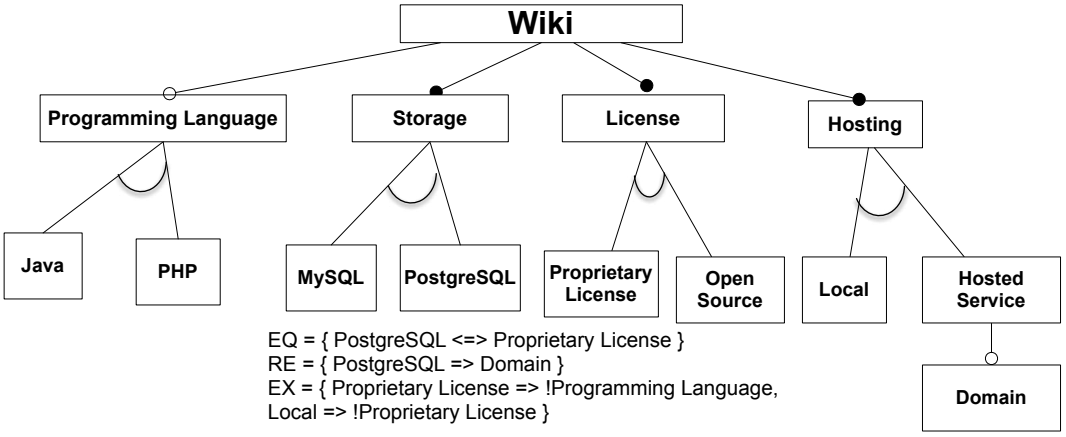


Figure 2: A FM of Wiki engines

Cross-tree *constraints* over features can be specified to restrict their valid combinations. Predefined forms of Boolean constraints (equals, requires, excludes) can be used. For instance, the feature PostgreSQL is logically related to Proprietary License and Domain.

Definition 1 (Feature diagram). *A feature diagram is a tuple FD where:*

- $FD = \langle G, E_{MAND}, G_{MUTEX}, G_{XOR}, G_{OR}, EQ, RE, EX \rangle$
- $G = (F, E, r)$ *is a rooted tree where* F *is a finite set of features,* $E \subseteq F \times F$ *is a finite set of edges and* $r \in F$ *is the root feature.*
- $E_{MAND} \subseteq E$ *is a set of edges that define mandatory features with their parents.*
- $G_{MUTEX} \subseteq P(F) \times F$, $G_{XOR} \subseteq P(F) \times F$ *and* $G_{OR} \subseteq P(F) \times F$ *define feature groups and are sets of pairs of child features together with their common parent feature.*
- *a set of equals constraints* EQ *whose form is* $A \Leftrightarrow B$*, a set of requires constraints* RE *whose form is* $A \Rightarrow B$*, a set of excludes constraints* EX *whose form is* $A \Rightarrow \neg B$ $(A, B \in F)$*.*

- *Features that are neither mandatory features nor involved in a feature group are optional features.*
- *A parent feature can have several feature groups but a feature must belong to only one feature group.*

As a result, a feature diagram represents the constraints on the features. These constraints can be translated into a propositional formula in conjunctive or disjunctive normal form [12]. For example, we can extract the formula ϕ_{wiki} in Equation 1 from the FM in Figure 2. This formula represents the constraints on the features of the FM and was obtained by following [31].

$$
\begin{aligned}
\phi_{wiki} = & \\
\text{mandatory feature:} \quad & \text{Wiki} \wedge \text{Storage} \wedge \text{License} \wedge \text{Hosting} \\
\text{child-parent relations:} \quad & \wedge (\text{Programming Language} \rightarrow \text{Wiki}) \\
& \wedge (\text{Java} \rightarrow \text{Programming Language}) \\
& \wedge (\text{PHP} \rightarrow \text{Programming Language}) \wedge (\text{MySQL} \rightarrow \text{Storage}) \\
& \wedge (\text{PostgreSQL} \rightarrow \text{Storage}) \wedge (\text{Proprietary License} \rightarrow \text{License}) \\
& \wedge (\text{Open Source} \rightarrow \text{License}) \wedge (\text{Local} \rightarrow \text{Hosting}) \\
& \wedge (\text{Hosted Service} \rightarrow \text{Hosting}) \wedge (\text{Domain} \rightarrow \text{Hosted Service}) \\
\text{Xor groups:} \quad & \wedge (\text{Programming Language} \rightarrow \text{Java} \veebar \text{PHP}) \\
& \wedge (\text{Storage} \rightarrow \text{MySQL} \veebar \text{PostgreSQL}) \\
& \wedge (\text{License} \rightarrow \text{Proprietary License} \veebar \text{Open Source}) \\
& \wedge (\text{Hosting} \rightarrow \text{Local} \veebar \text{Hosted Service}) \\
\text{additional constraints:} \quad & \wedge (\text{PostgreSQL} \leftrightarrow \text{Proprietary License}) \\
& \wedge (\text{PostgreSQL} \rightarrow \text{Domain}) \\
& \wedge (\text{Proprietary License} \rightarrow \neg\text{Programming Language}) \\
& \wedge (\text{Local} \rightarrow \neg\text{Proprietary License})
\end{aligned}
\tag{1}
$$

Translating a propositional formula into a feature diagram is not always feasible [51]. To express any possible product line, we need to add a propositional formula to the feature diagram [31], thus obtaining a FM (see Definition 2). We define the hierarchy of an FM as the rooted tree G in Definition 1.

Definition 2 (Feature model)**.** *An FM is a tuple $\langle FD, \phi \rangle$ where FD is a feature diagram and ϕ is a propositional formula over the set of features F.*

A similar abstract syntax is used in [8, 31, 52] while other existing dialects slightly differ [51]. Some authors added other interesting constructs on FM like feature cardinalities [28], attributes [14] or probabilities [26]. We restrict the work presented in this report on FMs as described in Definition 2.

2.3.2 Semantics

The essence of an FM is its *configuration semantics* (see Definition 3). The syntactical constructs are used to restrict the combinations of features authorised by an FM, e.g., Xor-groups require that at least one feature of the group is selected when the parent feature is selected. In Mutex-groups, features are mutually exclusive but one can deselect all of them, etc. The configuration semantics also states that a feature cannot be selected without its parent, i.e., all features,

except the root, logically imply their parent. Therefore, and importantly, the *feature hierarchy also contributes to the definition of the configuration semantics.* In our example in Figure 2, the set of valid configurations of the FM is the following:

$$
\begin{aligned}
[\![fm]\!] = \{ \\
&\{\text{PSQL, License, Domain, P. Li, Storage, Wiki, Hosting, HS}\} \\
&\{\text{MySQL, License, P. Lang, Java, Storage, OS, Wiki, Hosting, HS}\} \\
&\{\text{MySQL, License, Local, Storage, OS, Wiki, Hosting}\} \\
&\{\text{MySQL, License, Storage, OS, Wiki, Hosting, HS}\} \\
&\{\text{MySQL, License, Local, P. Lang, Storage, PHP, OS, Wiki, Hosting}\} \\
&\{\text{MySQL, License, Domain, P. Lang, Java, Storage, OS, Wiki, Hosting, HS}\} \\
&\{\text{MySQL, License, P. Lang, Storage, PHP, OS, Wiki, Hosting, HS}\} \\
&\{\text{MySQL, License, Domain, P. Lang, Storage, PHP, OS, Wiki, Hosting, HS}\} \\
&\{\text{MySQL, License, Domain, Storage, OS, Wiki, Hosting, HS}\} \\
&\{\text{MySQL, License, Local, P. Lang, Java, Storage, OS, Wiki, Hosting}\} \\
\}
\end{aligned}
\tag{2}
$$

Definition 3 (Configuration Semantics)**.** *A configuration of an FM fm is defined as a set of selected features. $[\![fm]\!]$ denotes the set of valid configurations of fm, that is a set of sets of features.*

Another crucial and dual aspect of an FM is the *ontological semantics* (see Definition 4). It defines the way features are conceptually related. Obviously, the feature hierarchy is part of the ontological definition. The parent-child relationships are typically used to *decompose* a concept into sub-concepts or to *specialize* a concept. Looking at Figure 2, the concept of Wiki is described along different properties like Hosting, License, or Programming Language ; License can be either specialized as an Open source or a Proprietary License, etc. Feature groups are also part of the ontological semantics (see Definition 4) since there exist FMs with the same configuration semantics, the same hierarchy but having different feature groups [6, 52]. For example in the FM of Figure 3a, PostgreSQL and Open Source are respectively grouped with MySQL and Proprietary License while in Figure 3b PostgreSQL is grouped with Open Source and MySQL is grouped with Proprietary License.

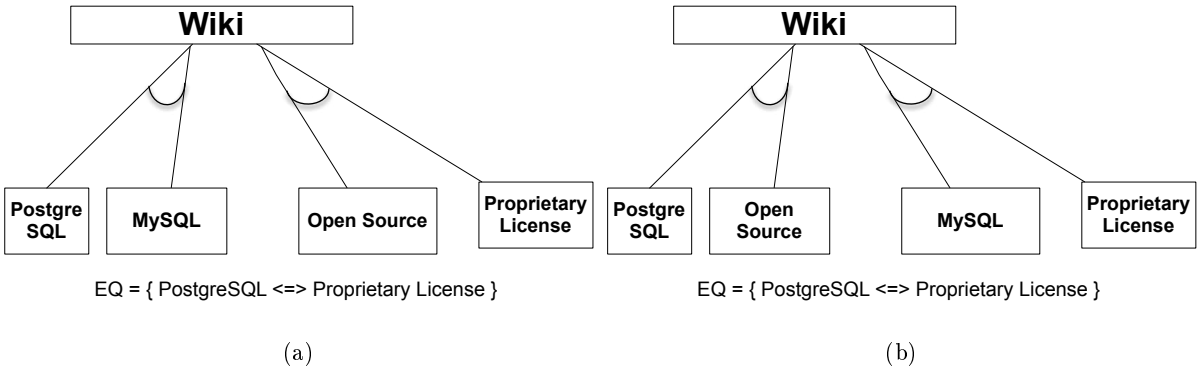


Figure 3: Two FMs with the same configuration semantics, the same hierarchy but different feature groups.

Definition 4 (Ontological Semantics)**.** *The hierarchy $G = (\mathcal{F}, E, r)$ and feature groups (G_{MUTEX}, G_{XOR}, G_{OR}) of an FM define the semantics of features' relationships including their structural relationships and their conceptual proximity.*

3 Problem Statement and Challenges

3.1 Motivation

A manual elaboration of an FM is not realistic for large complex projects or for legacy systems. Their variability can be expressed in a lot of different types of artefacts: requirements, source code, configuration files, tabular data, etc (see Figure 1, left on page 5 and Table 1). For instance, the Linux kernel contains more than 10,000 features scattered in configuration files, Makefiles, source code and documentation [32,52].

The modeler has to understand the intricate dependencies of the project by carefully reviewing all these artefacts. It requires tremendous effort and a single mistake may produce an inoperable FM [52]. An automated technique could significantly reduce the user effort. Moreover, such process would be reproducible thus helping the developers to maintain the variability of the project during its life cycle.

Reference	Objectives	Input artefacts
[31]	re-engineering	propositional formula
[4]	domain analysis	tabular data
[3]	re-engineering	source code and specification
[8]	re-engineering and generative approaches	propositional formula
[52]	understanding and generative approaches	documentation and code
[50]	re-engineering	incidence matrix
[26]	generative approaches	sample configurations
[21]	re-engineering	requirements
[33]	domain analysis and generative approaches	product descriptions
[60]	re-engineering	requirements

Table 1: Input artefacts of related work extracted from [19]

Many procedures have been developed to reverse engineer dependencies and features' sets from existing software artefacts [3, 4, 8, 21, 26, 31–33, 47–50, 52, 58, 60]. Table 1 shows that these prodedures are used for multiple objectives. In particular, the generated FMs are used for understanding an SPL and its inner dependencies, analyzing the existing products of a domain, re-engineering legacy code or generating recommendation systems and tests. Thus, the reverse engineering process is central in lots of realistic scenarios.

In these scenarios, the extracted dependencies can be formalized and encoded as a propositional formula. In our running example, Table 2 represents a family of Wiki engines as a product comparison matrix. The first column enumerates the features and the other ones describe the features contained in each product. We can easily extract the constraints on the features from this table thus resulting in Equation 1 on page 8.

Important management operations on FMs (merging, slicing, diff, refactoring) are also based on propositional formulas (see Figure 1). Merging two FMs fm_1, fm_2 consists in merging their

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
PostgreSQL	✓	×	×	×	×	×	×	×	×	×
MySQL	×	✓	✓	✓	✓	✓	✓	✓	✓	✓
License	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Domain	✓	×	×	×	×	✓	×	✓	✓	×
Proprietary License	✓	×	×	×	×	×	×	×	×	×
Local	×	×	✓	×	✓	×	×	×	×	✓
Programming Language	×	✓	×	×	✓	✓	✓	✓	×	✓
Java	×	✓	×	×	×	✓	×	×	×	✓
Storage	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
PHP	×	×	×	×	✓	×	✓	✓	×	×
Open Source	×	✓	✓	✓	✓	✓	✓	✓	✓	✓
Wiki	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Hosting	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Hosted Service	✓	✓	×	✓	×	✓	✓	✓	✓	×

Table 2: Tabular data representing a set of configurations

corresponding formulas ϕ_1 and ϕ_2. Slicing a FM allows to focus on a subset of the FM's features. For that we compute ϕ', a new formula obtained by removing the undesired features from the orginial formula ϕ. The difference of two FMs corresponds to the difference of their formulas. Refactoring a FM consists in redefining its feature diagram while ensuring the same configuration semantics, i.e. the same formula.

All these operations rely on a formula, but in order to enable the benefits of variability modeling described in Section 2.2, we have to synthesize an FM.

For example, a possible FM for the Equation 1 is depicted in Figure 2. From this FM we could, for example, generate a product configurator like in Figure 4. Such configurator asks the user what features he would like in his product and the configurator returns the product that best corresponds to his answers.

Therefore, the FM *synthesis problem* [8] is at the heart of many reverse engineering and FM management procedures. It also determines the quality and accuracy of numerous applications of feature modeling.

3.2 Problem Statement

Given a set of features' names and Boolean dependencies among features, the problem is to synthesize a maximal FM with a sound configuration semantics.

Formally, let ϕ a propositional formula in conjunctive or disjunctive normal form. A synthesis function takes as input ϕ and computes an FM such that $\mathcal{F}$ is equal to the Boolean variables of ϕ and $[\![FD, EQ, RE, EX, \psi]\!] \equiv [\![\phi]\!]$.

There are cases in which the feature diagram of an FM is not sufficient to express $[\![\phi]\!]$ (i.e., $[\![FD]\!] \subset [\![\phi]\!]$). It is thus required that the feature diagram is *maximal*: mandatory relationships are expressed through E_{mand}, without resorting to the cross-tree constraints RE ; no group definition can be strengthened (no Mutex-group or Or-group can become an Xor-group) ; no

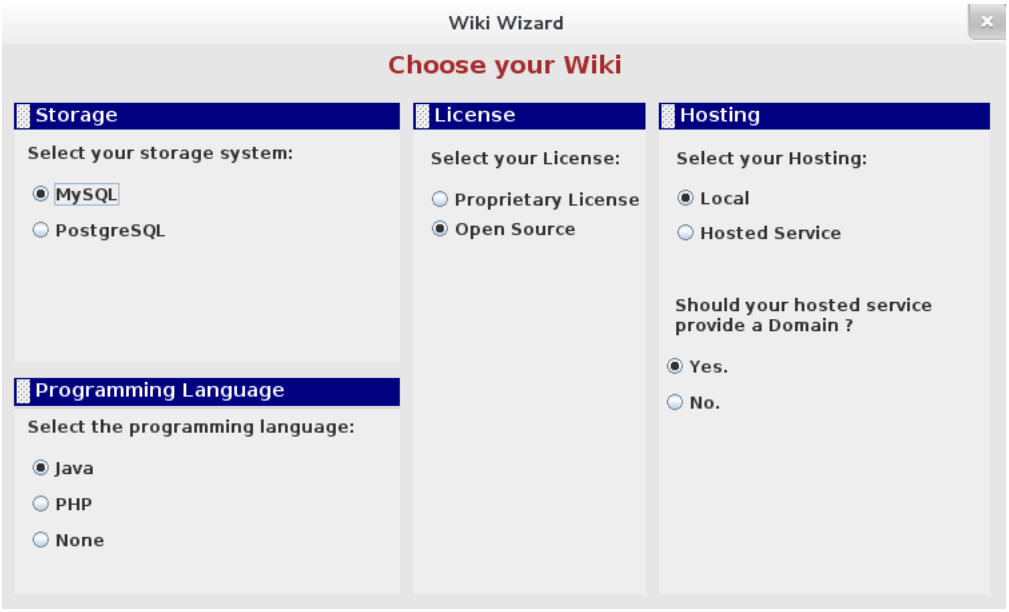


Figure 4: A user interface of a configurator that could have been generated from the FM in Figure 2.

additional groups can be computed without violating the well-formed rules of an FM (see Section 2.3.1) or the dependencies expressed by ϕ. Recently, the problem has been formalized in [8] and efficient synthesis techniques have been developed.

The problem tackled in this report is a generalization of the FM synthesis problem. Numerous FMs, yet maximal, can actually represent the exact same set of configurations, out of which numerous present a naive hierarchical or grouping organization that mislead the ontological semantics of features and their relationships (e.g., see Figure 2 on page 7 *versus* Figure 5a on page 13). We seek to develop an automated procedure that computes a well-formed FM both *i)* conformant to the configuration semantics (as expressed by the logical dependencies of a formula ϕ) and *ii)* exhibiting an appropriate ontological semantics.

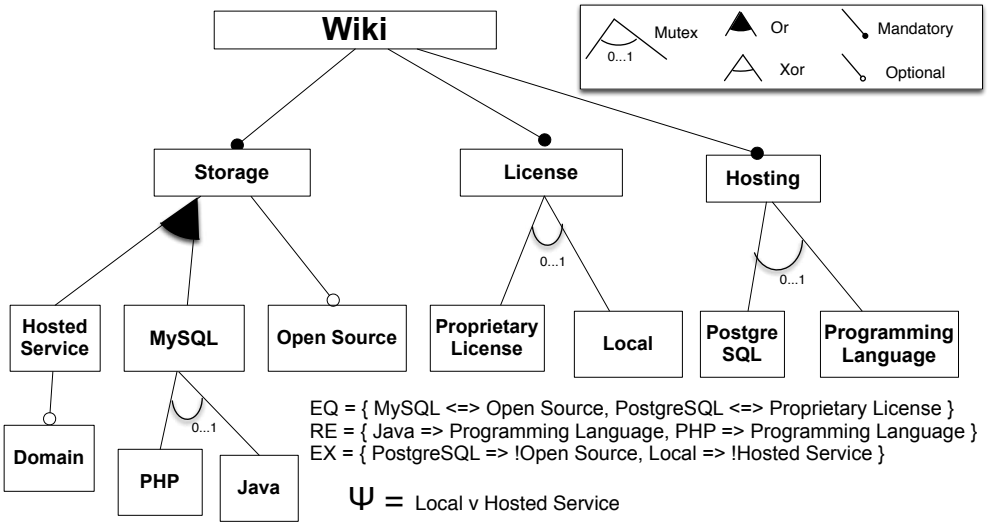
3.3 Challenges

The development of an ontologic-aware FM synthesis procedure raises new challenges that have been overlooked so far. In this section, we motivate further the prior importance of ontological semantics in FMs and show that none of existing works comprehensively addressed the problem.

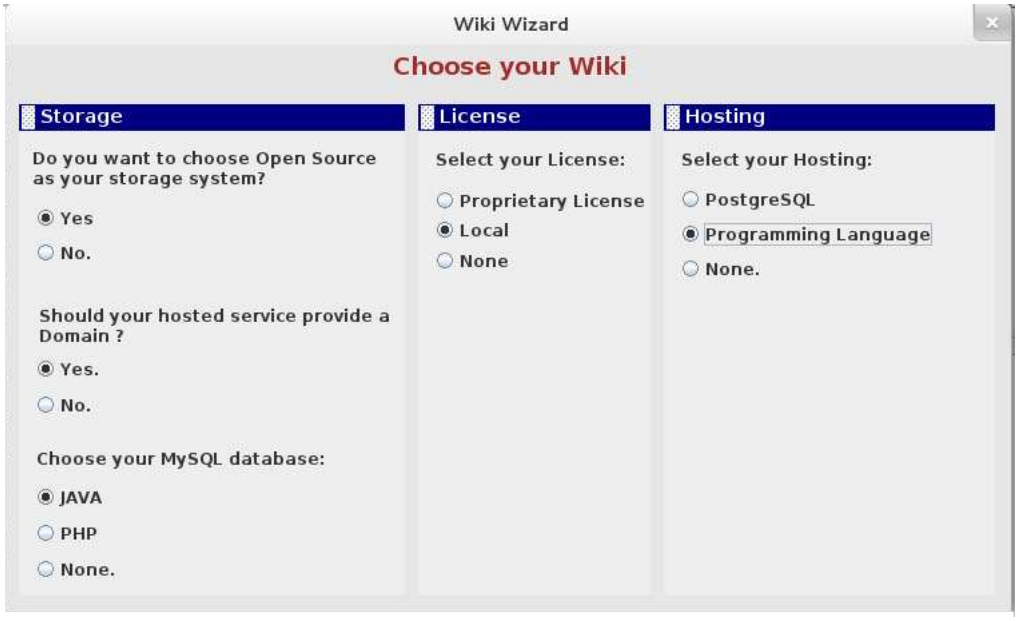
3.3.1 The Importance of Ontological Semantics

Let us first consider the re-engineering scenario mentioned in Section 3.1 (see [1, 16] for more details). After having reverse engineered an FM, a practitioner aims at generating a graphical interface (in the style of Figure 4) for assisting users in customizing the product that best fits their needs. Unfortunately, the ontological semantics of the FM is highly questionable (see Figure 5a). The bad quality of the FM poses two kinds of problems.

Automated exploitation. Transformation tools that operate over the FM will naturally exploit its ontological semantics. Figure 5 gives an example of a possible transformation of an FM into a user interface (UI). The generated UI (see Figure 5a) is as unintuitive as the ontological semantics is: features PHP and Java are below MySQL, instead of being below Programming



(a) fm_u



(b) Configurator UI

Figure 5: An FM fm_u with a doubtful ontological semantics and a non intuitive user interface of a configurator that could have been generated from fm_u.

Language ; Programming Language itself is badly located since below Hosting ; Open Source is below Storage whereas the intended meanings was to state that a Wiki engine has different alternatives of a License. All in all, the series of questions and the organization of the elements in the UI are clearly non exploitable for a customer willing to choose a Wiki.

Human exploitation. One could argue that an automated transformation is not adequate and a developer is more likely to write or tune the transformation. In this case, the developer faces different problems. First, there are evident readability issues when the developer has to understand and operate over the model. Second, when writing the transformation, the ontological semantics cannot be exploited as such and she has to get around the issue, complicating her task. A solution could be to restructure (i.e., refactor) the FM, but the operation is an instance of the synthesis problem (see below). Third, default transformation rules that could reduce her effort are likely to cause problems since based on the ontological semantics.

This scenario illustrates the importance of having FMs with an appropriate ontological semantics for a further exploitation in a forward engineering process. This need is observed in other FM-based activities, e.g., when understanding a domain or a system [3,10], communicating with other stakeholders [41], relating FMs to other artefacts (e.g., models) [17,30,36], or simply generating other artefacts from FMs [27].

Limits of FM management operations. The ontologic-aware synthesis problem not only arises when reverse engineering FMs. It is also apparent when *refactoring* an FM, i.e., producing an FM with the same configuration semantics but with a different ontological semantics. For example, the FM of Figure 5a can be refactored to enhance its quality and make it exploitable.

The operation of *slicing* an FM has numerous practical applications (decomposition of a configuration process in multi-steps, reduction of the variability space to test an SPL, reconciliation of variability views, etc.) [5]. Despite some basic heuristics, Acher et al. observed that the retained hierarchy and feature groups can be inappropriate [6] (the same observation applies for the *merge* and *diff* operators).

All these automated operations are instances of the FM synthesis problem (see Figure 1) and are impacted by the problem.

3.3.2 Limits of Related Work

Dependencies-aware synthesis. Techniques for synthesising an FM from a set of dependencies (e.g., encoded as a propositional formula) have been proposed [6,8,31,34,35,37,52]. An important limitation of prior works is the identification of the feature hierarchy when synthesizing the FM.

In [8,31], the authors calculate a diagrammatic representation of all possible FMs, leaving open the selection of the hierarchy and feature groups.

The algorithms proposed in [34] and [35] do not control the way the feature hierarchy is synthesized in the resulting FM and may generate FM whose configurations violate the input dependencies.

In [6], Acher et al. proposed a synthesis procedure that processes user-specified knowledge for organizing the hierarchy of features. Janota et al. [37] proposed an interactive environment to guide users in synthesizing an FM. In both cases [6,37], the effort may be substantial since users have to review numerous potential parent features (8.5 in average for the FMs of the SPLOT repository, see Section 6).

Ontologic-aware synthesis. Alves et al. [7], Niu et al. [44], Weston et al. [60] and Chen et al. [21] applied information retrieval techniques to abstract requirements from existing specifications, typically expressed in natural language. These works do not consider precise logical dependencies and solely focus on ontological semantics. As a result, users have to manually set the variability information. Moreover, a risk is to build an FM in contradiction with the actual dependencies of a system or a domain.

The previous exposed works overlooked the combination of configuration and ontological aspects when synthesising an FM. We now review other existing works trying to address the two aspects together.

Dependencies and ontologic-aware synthesis. In [4], Acher et al. proposed a semi-automated procedure to support the transition from product descriptions (expressed in a tabular format) to FMs. In [3], Acher et al. combined architectural knowledge, plugins dependencies and the slicing operator to obtain an exploitable and maintainable FM ; in particular the feature hierarchy reflects the hierarchical structure of the system. Ryssel et al. developed methods based on Formal Concept Analysis and analyzed incidence matrices containing matching relations [50].

In the operating system domain, She et al. [52] proposed a procedure to rank the correct parent features in order to reduce the task of a user.

The procedures exposed in [3, 4, 50, 52] are specific to a project or a domain and assume the existence of a certain structure or artefacts (e.g., textual corpus, hierarchical model of the architecture) to organize the features in the FM. We cannot make similar assumptions in the general case.

For example, the list of features may be flattened and no prior ontological knowledge may be inferred as in the matrix of Figure 2 and reported in [34,35]. As another example, ontological knowledge is missing when extracting conditional compilation directives from source code and build files [32]. Finally, the inner structure of the input artefacts of the synthesis may not be the desired one by the user.

Furthermore inferring a complete hierarchy from the analysis of artefacts is hardly conceivable. It is most likely that a partial ontological knowledge [3, 4] or a ranking of candidate parent features [52] will be obtained. Alternative automated strategies for computing such knowledge and interactive support for assisting users are thus worth to consider.

3.3.3 Summary

The FM synthesis problem is at the heart of many reverse engineering and FM management procedures. A naive organization of features in the synthesized FM may dramatically increase the stakeholders' effort when communicating among others or when understanding, maintaining and developing configurable systems.

As we can see in Figure 6, only a few works considered configuration semantics and ontological semantics together and in a comprehensive manner. Finally, state-of-the-art FM synthesis techniques and empirical observations show that the major challenge and time-consuming task is to select a hierarchy. The variability information and other components of an FM come almost for free in the vast majority of cases [6,8]. Therefore, in the reminder, we focus on the challenge of choosing a valid and comprehensible hierarchy.

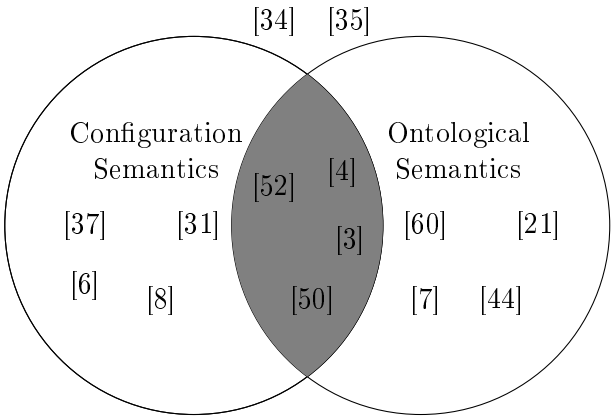


Figure 6: Open issue in FM synthesis: addressing both configuration and ontological semantics

4 Breathing Knowledge into Feature Model Synthesis

Overview. Our ontologic-aware FM synthesis procedure (see Figure 7) essentially relies on a series of heuristics to *i)* rank parent-child relationships and *ii)* compute clusters of conceptually similar features. These two types of information can be interactively complemented or refined by a user, and if needed, an optimum branching algorithm can synthesize a complete FM. The heuristics form the main theoretical contribution of this work. They come in different variants and do not assume any artefacts documenting features.

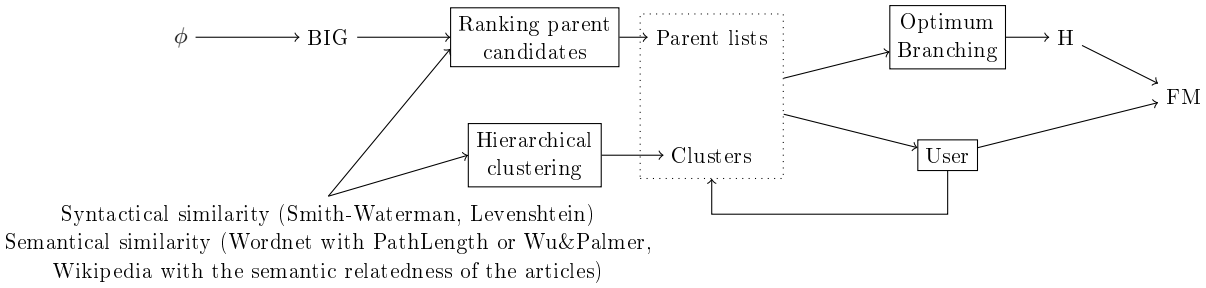


Figure 7: Feature model synthesis

4.1 Selecting a Hierarchy

We recall that the feature hierarchy of an FM defines how features are ontologically related and also participates to the configuration semantics, since each feature logically implies its parent: $\forall (f_1, f_2) \in E, f_1 \Rightarrow f_2$. As a result, the candidate hierarchies, whose parent-child relationships violate the original constraints expressed by ϕ, can be eliminated upfront. For example, the feature Local cannot be a child feature of PostgreSQL since no configuration expressed in Table 2 authorizes such an implication.

We rely on the *Binary Implication Graph* (BIG) of a formula (see Definition 5 and Figure 8) to guide the selection of legal hierarchies. The BIG represents every implication between two variables (resp. features) of a formula (resp. FM), thus representing every possible parent-child relationships a legal hierarchy can have. Selecting a hierarchy now consists in selecting a set of the BIG's edges forming a rooted tree that contains all its vertices. Such a tree represents a branching of the graph (the counterpart of a spanning tree for undirected graphs). The selection

over the BIG is *sound* and *complete* since every branching of the BIG is a possible hierarchy of the FM and every hierarchy is a branching of the BIG.

Definition 5 (Binary Implication Graph)**.** *A binary implication graph of a propositional formula ϕ over $\mathcal{F}$ is a directed graph $BIG = (V_{imp}, E_{imp})$ where $V_{imp} = \mathcal{F}$ and $E_{imp} = \{(f_i, f_j) \mid \phi \wedge f_i \Rightarrow f_j\}$.*

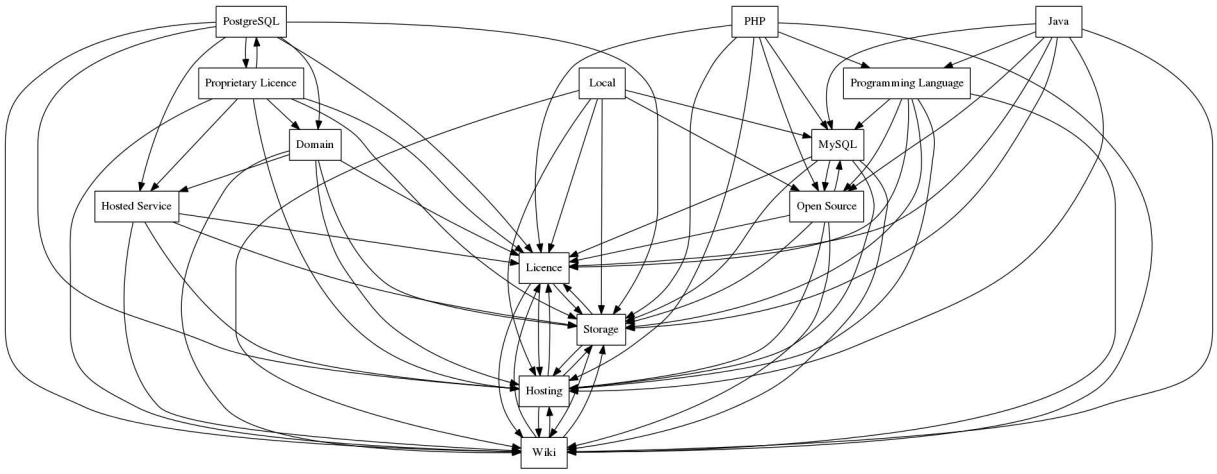


Figure 8: Binary implication graph of the FM in Figure 2

Now that we have a convenient data structure, which captures every possible hierarchy, the whole challenge consists in selecting the most meaningful one.

A first natural strategy is to add a weight on every edge (f_1, f_2) of the BIG, defining the probability of f_2 to be the parent of f_1. The weights are computed by heuristics that directly evaluate the probability of a relationship between a feature and a parent candidate. From a user perspective, the *ranking list* can be consulted and exploited to select or ignore a parent.

In addition, we perform hierarchical *clustering* based on the similarity of the features to compute groups of features. The intuitive idea is that clusters can be exploited to identify *siblings* in the tree since members of the clusters are likely to share a common parent feature. For example, obtained clusters can help tuning the previously computed weights, thus increasing the quality of the previous ranking list. Moreover, users can operate over the clusters to define the parent of a group of features (instead of choosing a parent for every feature of a group).

Once we get the two pieces of information, we select the branching that has the maximum total weight. To compute an optimum branching, we use Tarjan's algorithm [20, 55] whose complexity is $\mathcal{O}(m \log n)$ with n the number of vertices and m the number of edges. The hierarchy is then fed to syntesise a complete FM. The synthesis process is likely to be incremental and interactive. Users can validate and modify the ranking list and the set of clusters. The overall synthesis process is described in Figure 7.

4.2 Heuristics for parent candidates

The design of the heuristics is guided by a simple observation: parent-child relations in a hierarchy often represent a specialization or a composition of a concept (feature). For example, Java is a specialization of a Programming language while a Wiki is composed of a License, a Storage and a Programming Language. As siblings are specializations or parts of the more general concept of their parent, they share the same context. For example, Open source and Commercial are

both referring to permissive rights about the use of a product. The intuitive idea is that sharing the same context tends to make a feature semantically close to its parent and its siblings.

From these observations, we developed several heuristics that exploit the feature names in order to compute the edges' weights of the BIG. We can divide the heuristics in two categories: syntactical heuristics and semantical heuristics.

Syntactical heuristics. Syntactical heuristics use edit distance and other metrics based on words' morphology to determine the similarity of two features. In our example in Figure 2, License is closer to Proprietary License than Storage because the two first features contain the same substring: *License*. We used *Smith-Waterman* algorithm [53] that looks for similar regions between two strings to determine their distance. We also used the so-called *Levenshtein* edit distance [59] that computes the minimal edit operations (renaming, deleting or inserting a symbol) required to transform the first string into the second one.

Semantical heuristics. Syntactical heuristics have some limits, for example, when we consider feature names of Figure 2. In particular, feature names may not be syntactically close but semantically close (in the sense of meaning). Thus, we need to add some semantical knowledge to improve our technique. We explored two general ontologies out of which we built semantical metrics.

First, we explored *WordNet* [42]. This is a structured English dictionary that provides hyperonymy (specialization) and meronymy (composition) relations between word senses.

As we are exclusively using the feature's names in this report, we could not use the most efficient metrics based on a text corpus [18]. Therefore, we selected two metrics named *PathLength* and *Wu and Palmer* that are only based on WordNet's structure. The PathLength metric gives the inverse of the length of the shortest path between two words in WordNet considering only hyperonymy relations. Wu and Palmer described a metric based on the depth of the words and their lowest common ancestor in the tree formed by hyperonymy relations [61]. We exclusively use hyperonymy relations because they are the most represented relations in WordNet and they form trees of word senses that are convenient for determining semantical similarity in our case.

These two previous metrics compute the similarity of two words, however features' name may contain several words. Wulf-Hadash et al. also used the Wu&Palmer metric in order to compute feature similarity [62]. We use the same formula for combining word similarities into sentence similarity:

$$Sim(f1, f2) = \frac{\sum_{i=1}^{m} \max_{j=1..n} wsim_{i,j} + \sum_{j=1}^{n} \max_{i=1..m} wsim_{i,j}}{m+n}$$

where m and n are respectively the number of words in $f1$ and $f2$ and $wsim_{i,j}$ is the similiarity between the i-th word of $f1$ and the j-th word of $f2$.

The main limit of WordNet is the lack of technical words. For example, MySQL and PostgreSQL are missing from WordNet. Thus we explored *Wikipedia* to increase the number of recognized words. The well known online encyclopedia offers a large database containing text articles and links between them. Associating a set of articles to a feature enables the use of techniques that compute semantic similarity of texts. For example we can associate Java and Programming language to their respective articles in Wikipedia. Then we compute their similarity by comparing the links contained in these articles as proposed by Milne et al [43]. They created a model based on the hyperlink structure of Wikipedia in order to compute the semantic similarity of two articles. In this model, an article is represented by a vector containing the occurence of each link found in the article weighted by the probability of the link occuring in

the entire Wikipedia database. Thus, a link pointing to the article about *Science* will have a smaller weight than a link pointing to the article about *Wiki software*. The similarity between two articles is given by the cosine similarity of the vectors representing them.

The weights given by the heuristics can be used to present a sorted list of parent candidates to the user. In the example in Figure 2, our heuristic based on Wikipedia would give the following list for the feature Proprietary License: [License, Wiki, PostgreSQL, Storage, Hosting, Hosted Service, Domain]. It means that License is most likely the parent of Proprietary License and, at the contrary, Domain has a low probability.

4.3 Detecting siblings with hierarchical clustering

Defining weights on the BIG's edges and computing the optimum branching can be summarized as choosing the best parent for a feature. However, it is sometimes easier to detect that a group of features are siblings. To detect such siblings we reuse the previous heuristics to compute the similarity of each pair of features without considering the BIG structure. Then we perform agglomerative hierarchical clustering on these values.

Agglomerative hierarchical clustering consists in putting each feature in a different cluster. Then, the closest clusters according to their similarity are merged. The distance between two clusters is determined according to a user defined metric and linkage. The metric gives the distance between two features (e.g. the Euclidian distance) and the linkage gives the distance between two clusters according to the previous metric. Such linkage can take the minimum distance between two features (single linkage), the maximum distance (complete linkage) or for example the mean distance (average linkage). Finally, the merge operation is repeated until the similarity falls below a user specified threshold.

We use the BIG to separate the obtained clusters in 3 categories. In the following, C represents a cluster and $possibleParents$ gives the parent candidates according to the BIG.

- $\exists f_p \in \mathcal{F}, \forall f_c \in C, f_p \in possibleParents(fc)$, i.e. all the features can be siblings. It remains to find a common parent of the cluster among the other features.
- $\exists P \subset C$, $\forall f_p \in P$, $\forall f_c \in C$, $f_p \neq f_c$, $f_p \in possibleParents(fc)$, i.e. some features could be the parent of the others. It remains to find the parent among the features within the cluster.
- Neither of the previous cases. We must refine the cluster in subclusters to get back to the first two cases. In this step, we use the BIG to help us separating the unrelated features.

Once we have defined the parents of the clusters, we modify the edges' weights of the BIG to consider this new information during the optimum branching algorithm. This modification consists in setting the maximal weight to each edge between a child and its chosen parent. The rationale is that features of the clusters are likely to share the same parent, thus the idea of augmenting the weights.

For example, in Figure 2, we could determine with a heuristic, that Java and PHP are semantically close, thus being in a same cluster. It corresponds to the first category exposed above. The key benefit is that we can now solely focus on their common parents (computed using the BIG). The best parent is chosen according to heuristics for parent candidates. (It should be noted that the heuristic is not necessarily the same one used for the clustering). We obtain Programming Language in the example. As a result, we assign the maximum weight (i.e. 1) for the following BIG's edges: Java → Programming Language and PHP → Programming Language.

4.4 Illustration on the example

Now that we have explained each part of our synthesis procedure, we can illustrate the entire process on the example in Figure 2 on page 7. We choose an heuristic based on Wikipedia and the Smith-Waterman heuristic for the clustering (see Appendix A for the complete configuration and the same example with the environment of Section 5). After extracting the formula from the FM, we compute the parent candidate lists and the clusters. We obtain the following results:

Parent candidates (partial list)

- *PostgreSQL*: Wiki, Proprietary License, Storage, License, Hosting, Domain, Hosted Service
- *MySQL*: Wiki, Open Source, Storage, Hosting, License
- *Proprietary License*: License, Wiki, PostgreSQL, Storage, Hosting, Hosted Service, Domain
- *Programming Language*: Open Source, MySQL, Storage, Wiki, Hosting, License
- *Java*: Open Source, MySQL, Programming Language, Wiki, License, Storage, Hosting
- *Hosting*: License, Storage, Wiki

Clusters

- PostgreSQL, MySQL
- Storage, Programming Language
- License, Proprietary License
- Hosting, Hosted Service

From these information we push further the synthesis and generate a complete hierachy with the optimum branching algorithm.

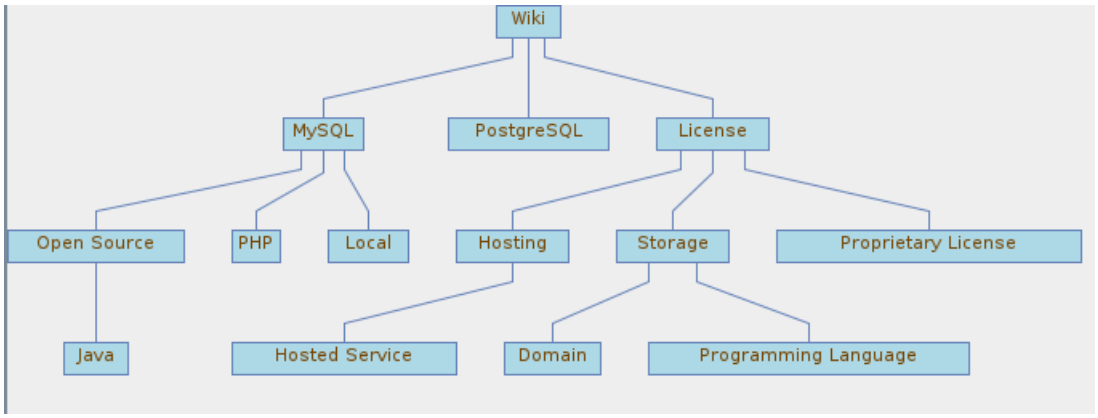


Figure 9: Automatically generated hierarchy from our heuristics

At first sight, the resulting hierarchy in Figure 9 is not understandable but by analyzing the results we can see that the optimum branching algorithm correctly used the generated clusters and the parent candidate lists. Furthermore, the procedure gave us some correct relations and especially correct clusters. Wiki is indeed the root of the FM. License is the parent of Proprietary

License and Hosting is the parent of Hosted Service. Moreover, PostgreSQL and MySQL are indeed siblings.

Finally, the mistakes made by our heuristics can be explained. For example, in the parent candidate list of Java, we find Open Source in first position instead of Programming Language. Java is indeed a programming language but it is also an open source project. If we look at features like Storage or Hosting, their names are very general and have many senses or meanings. To pick the correct sense, we need the context of the FM. However, this information is not available in the input formula of our procedure.

This example show positive results and limits of our synthesis techniques. Our techniques reduce the user effort by providing ranking lists and clusters. The optimum branching can also deduce a part of the hierarchy. However, the synthesis may require arbitrary decisions or additional information like the context of the FM. Only the user can provide this information, thus motivating the need of an environment as presented in Section 5.

5 An Interactive Environment for Feature Model Management

We developed a specific synthesis support and environment[2] on top of the FAMILIAR language [5]. FAMILIAR already includes numerous facilities for managing FMs. A previous work [6], offers a way to include ontological knowledge into FM synthesis. The user can specify a part of the hierarchy or the feature groups and FAMILIAR complete the FM with arbitrary choices while assuring the configuration semantics. However, this is not an interactive process and no assistance is provided to the user.

We equiped the synthesis technique, central to many operations of FAMILIAR, with ontological capabilities (see Section 4) and a graphical and interactive support. The environment raises previous ontological limitations of FAMILIAR when reverse engineering FMs from a set of Boolean constraints, merging a set of FMs, slicing an FM, computing differences of two FMs as an FM, or refactoring an FM.

5.1 Features of the Environment

Specifically, our tool offers an interactive mode where the user can import a formula (e.g., in DIMACS format), synthesizes a complete FM and export the result in different formats. During the FM synthesis, the graphical user interface displays a ranking list of parent candidates for every feature, a list of clusters and a graphical preview of the FM under construction (see Figure 10). During the interactive process, users can:

- select or ignore a parent candidate in the ranking lists
- select a parent for a cluster within the cluster's features or any potential parent feature outside the cluster (the user can also consider a subset of a cluster when selecting the parent)
- undo a previous choice
- define the different heuristics and parameters of the synthesis
- automatically generate a complete FM according to previous choices and selected heuristics.

[2]This section is the base for the following submitted tool demonstration paper (under review): Guillaume Bécan, Sana Ben Nasr, Mathieu Acher and Benoit Baudry. An Ontologic-Aware Feature Modeling Environment. In the *28th IEEE/ACM International Conference on Automated Software Engineering*. 2013.

A typical usage is to perform some choices, generate a complete FM with the heuristics and reiterate until having a satisfactory model. Appendix A illustrate this usage on the running example.

5.2 Representing Information During FM Synthesis

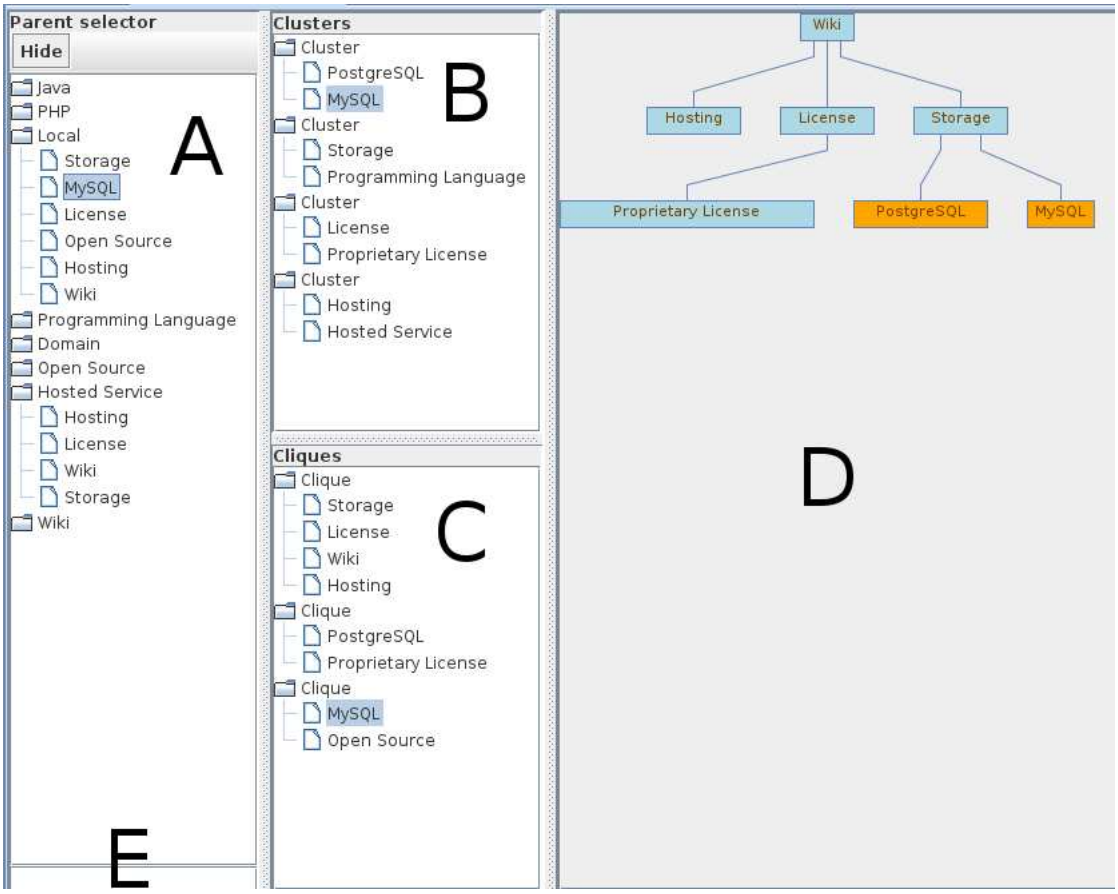


Figure 10: Interface of the environment during synthesis

5.2.1 Parent candidates and clusters

A crucial aspect of the environment is the representation of the results of the heuristics. The ranking lists of parent candidates and the clusters form the main information during the synthesis. As the amount of information can be overwhelming, we decided to use a tree explorer view for both parent candidates and clusters (see Figure 10, A, B and C). A tree explorer view is scalable and allows to focus on specific features or clusters as the information for closed features or clusters in the explorer is hidden.

During the synthesis, the user interacts almost exclusively by right-clicking on the elements of these explorers. We offer a simple way to take into account the 3 categories of clusters (see Section 4.3). For the first category, the user can select the parent of a cluster among the common parent candidates of its features by right-clicking on the cluster (see Figure 11a). For the second category, the user can select the parent among the cluster's features by right-clicking on the desired parent (see Figure 11b). In both cases, the user can deselect some features to consider only a subset of the cluster, thus supporting the third category of clusters (see Figure 11c).

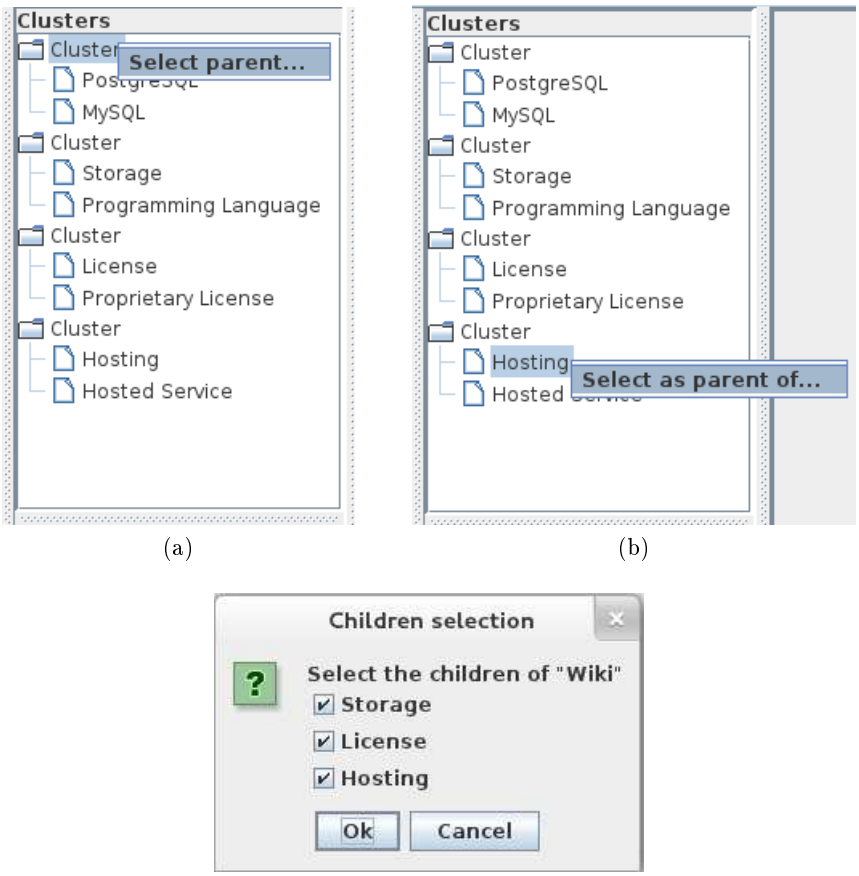


Figure 11: Handling of the 3 categories of clusters in the environment.

5.2.2 Feature model under construction

During the synthesis, each choice from the user operated on the features and the clusters has an impact on the FM. A graphical representation of the FM is crucial to summarize the previous choices and have a global view of the FM. We considered two types of representation: a graph and a tree explorer view.

The graph represents the hierarchy of the FM with boxes for features and lines for parent-child relations (see Figure 10, D). Features are organized hierarchically as in the common representation used for the hierarchy of the FM in Figure 2. Thus, it offers a familiar and clear view of the FM for the user. However with large FMs, the layout of the features and the size of the FM become hard to handle.

Tree explorer view, as used for parent candidates and clusters, do not offer a clear view of the entire FM but scales well with the size of the FM. Such view easily allows to focus on a subset of features and, unlike a graph, no layout algorithm is required.

For now, we only use a graph representation developed with the JGraphX library[3]. We plan to address the problem of scalability by automatically switching to a tree explorer view, zooming on a subset of the graph or slicing the FM.

To encourage users to look at all the information available on screen and to avoid searching a specific feature in the lists, we propagate the selection from the parent candidate lists to the

[3] `http://www.jgraph.com/jgraph.html`

cluster list (see Figure 10, A and B). Moreover we highlight the features of the selected cluster in the graphical representation of the FM (see orange features in Figure 10, D). Finally, the user can filter the features in the parent candidate lists by typing a part of their name in a search bar (see Figure 10, E).

5.3 Implementation of the Heuristics

Hierarchical clustering is performed by the Hac[4] library. As no Java implementation of Tarjan's algorithm was available we developed it from scratch.

We also developed 6 heuristics based on specialized libraries. The syntactical heuristics, *Smith-Waterman* and *Levenshtein*, come from the Simmetrics[5] library. *PathLength* and *Wu and Palmer* rely on extJWNL[6] which handles the communication between WordNet and our tool. Wikipedia Miner[7] offers an API to browse Wikipedia's articles offline and compute their relatedness [43]. We used this tool on the english version of Wikipedia and Wiktionary which form the last two heuristics.

Unlike WordNet, Wikipedia Miner requires a long process before using it. First, it needs a dump of the Wikipedia database. Such dump is contained in a 40GB+ XML file. Then, summaries must be extracted from this dump. Finally, these summaries and the dump are used to form a new database. The whole process took us about 2 days on a 8 core Intel Xeon processor at 2.27GHz.

It is important to note that our implementation provide a common interface for all the heuristics. This interface only requires the development of a function that provide the similarity of two strings: a feature and its parent candidate. The returned value must be between 0 and 1. With this simple interface, we offer a convenient way for adding new heuristics.

5.4 Another type of clusters: the cliques

In Section 4.3, we have used hierarchical clustering to compute clusters of features that are semantically related. FAMILIAR offers another type of clusters which are composed of features that are logically related. These clusters are extracted from the BIG by computing the maximal cliques of the graph. A clique is a set of nodes such that each node has an outgoing edge to every other node of the clique. A clique is maximal when we cannot add another node to form a bigger clique. In the case of the BIG, it means that every feature of a clique implies the other ones. Thus, a clique represent a set of features that are always together in a product. Cliques are important in FM synthesis. They reduce the user effort by allowing him to choose one parent for all the features of the clique. Moreover, we often saw that the cliques in our experiments were formed by the root of the FM and the mandatory from the first or second level of the hierarchy. Thus, it helps structuring the FM.

In our example in Figure 2, there are 3 non-trivial cliques: {Storage, License, Wiki, Hosting}, {PostreSQL, Proprietary License} and {MySQL, Open Source} (see Figure 10, C). As we observed, the first one is indeed formed by the root and the first level of mandatory features. We use the same graphical representation as clusters for the cliques (see Figure 10).

[4] `http://sape.inf.usi.ch/hac`
[5] `http://sourceforge.net/projects/simmetrics`
[6] `http://extjwnl.sourceforge.net`
[7] `http://sourceforge.net/projects/wikipedia-miner`

6 Evaluation

6.1 Goal and Scope of the Evaluation

Our techniques aim at assisting the user during the FM synthesis. We evaluate them through the following research question: How effective are our automated techniques to reduce the user effort when synthesizing FMs?

We focus on evaluating the quality of the parent candidate lists, the clusters and the final FM generated by the optimum branching algorithm. We consider that we reduce the user effort when the parent candidate lists are correctly sorted, the clusters contain siblings and the optimum branching algorithm provides a FM close to a ground truth. We evaluate this reduction only from a theoretical point of view and we do not consider the usability of our tool.

6.2 Experimental Settings

6.2.1 Data

We evaluate our techniques on two data sets. The *SPLOT*[8] repository offers a large set of FMs created by practitioners from different domains. Numerous papers use SPLOT for evaluating their proposals (e.g., [6, 8, 34, 35, 40, 46]). We restricted the repository to English FMs that contains meaningful feature names, resulting in 108 FMs with a total of 2266 features. *Product Comparison Matrices* (PCMs), comparing features of domain specific products, abound on the internet (e.g., on Wikipedia) [4]. For example, the page *Comparison of wiki software*[9] compares the available Wiki software in terms of license, encoding or programming language. We gathered 16 FMs containing 593 features with an automated extraction process[10]. The structure of the web pages and the matrix allows us to extract FMs with realistic hierarchies. Importantly we do not change the original ontological semantics of a PCM, we only tune the extraction when interpreting variability of PCMs data. The dataset is challenging for our procedures since the number of cross-tree constraints can be huge, the number of features is rather important, and feature names are disparate.

To give an overview of the complexity of the synthesis we extract from the BIG of an FM, the number of parent candidates for each feature. We have an average of 8.5 (from 0 to 36) parent candidates in SPLOT FMs and 4.2 (from 0 to 10) for PCM-based FMs. Therefore, the user would have in average the same number of parent candidates to analyze before taking a decision for one feature.

From each FM of these two data sets, we extract its hierarchy representing its ontological semantics and compute its formula ϕ representing its configuration semantics. The formula serves as input of our proposed algorithms and the hierarchy constitutes a ground truth on which we can rely to evaluate the quality of our heuristics and algorithms.

6.2.2 Measurements

To evaluate the quality of the parent candidate lists, we check for each feature that its correct parent appears in the top 2 positions of the parent lists generated by a heuristic. With an average of 8.5 parent candidates per feature for SPLOT FMs, we chose to restrict our evaluation to the top two positions in order to reduce the impact of random choices. It already allows about 25% of probability of having a correct parent list only by chance.

[8] `http://www.splot-research.org`

[9] `http://en.wikipedia.org/wiki/Comparison_of_wiki_software`

[10] A description of the process is available in the following web page: `https://github.com/FAMILIAR-project/familiar-documentation/tree/master/manual/ontologicalTutorial`

To evaluate the quality of the clusters we check that they contain features that are indeed siblings in the original FM or that the clusters contains the parent of their other features. For example in Figure 2, {Hosted Service, Local} is a correct cluster because these features are siblings. {Hosted Service, Local, Hosting} is also a correct cluster because Hosting is the parent of Hosted Service and Local. These two cases reflect our observation on the semantic similarity of siblings and their parents. They also correspond to the first two categories of clusters in Section 4. For each heuristic, we optimized the clustering threshold in order to have the maximum number of features in a correct cluster.

We also compute the number of choices (nc) required to complete all the FMs if we use the correct clusters. A choice consists in choosing the correct parent candidate and each correct cluster represents a single choice for all the features it contains.

$$nc = \text{number of features} - \text{number of features in a correct cluster} + \text{number of correct clusters}$$

To evaluate the optimum branching algorithm, we automatically generate a hierarchy for each FM. Then, we compare this generated hierarchy with the original one by computing the tree edit distance according to Zhang's algorithm [63]. This distance corresponds to the minimal number of edit operations required to transform the first FM in the second one. The edit operations are renaming a node, deleting a node or inserting a node. We divide the distance by the number of features in the FM, thus corresponding to the average number of required edit operations per feature. Thanks to this division we also get comparable values between the studied FMs.

We execute these three tests with each heuristics described in Section 4 and 5 and an heuristic providing random values. The random heuristic gives a reference for an algorithm that would not consider the ontological semantics of an FM.

6.3 Experimental Results

We gather the following information for each heuristic:

- Number of correct parents in the top 2 positions
- Number of clusters
- Mean cluster size
- Number of correct clusters
- Number of features in a correct cluster
- Number of choices required to complete the FM
- Mean tree edit distance between the generated FM and the ground truth

To avoid extreme values with the random heuristic we compute the probability of having a correct parent in the top 2 positions of the parent candidates lists and we take the average over 100 executions for the other values.

Table 3 shows the values of the metrics we computed during the evaluation. For the 2266 features of SPLOT FMs, the random heuristic puts 36.5% of the correct parents in the top 2 positions of the parent candidate lists. The heuristics we developed reach about 46% of the correct parent.

For the 593 features of PCM-based FMs, the random heuristic (resp. the heuristics we developed) puts 56% (rep. 57%) of the correct parents. These closer results are explained by the lower average number of parent candidates which is 4.2. It increases the probability of having

Metric	Data	R	SW	L	W&P	PL	Wiki	Wikt
Top 2	SPLOT	828	1089	1017	1017	1059	1088	1048
	PCM	335	363	322	341	339	357	312
Number of clusters	SPLOT	880	561	555	370	349	453	390
	PCM	242	137	139	47	68	105	103
Mean cluster size	SPLOT	2.20	2.96	2.88	2.67	2.85	3.06	2.78
	PCM	2.26	3.43	2.93	2.83	2.84	4.02	3.02
Number of correct clusters	SPLOT	160	273	290	190	232	249	241
	PCM	50	68	98	34	53	66	73
Number of features in a correct cluster	SPLOT	325	691	753	448	607	646	602
	PCM	104	236	294	95	153	205	208
Number of required choices to complete the FMs	SPLOT	2101	1848	1803	2008	1891	1869	1905
	PCM	539	425	397	532	493	454	458
Mean edit distance	SPLOT	0.638	0.596	0.593	0.584	0.582	0.578	0.576
	PCM	0.605	0.552	0.530	0.498	0.480	0.517	0.546

Table 3: Experimental results

a correct parent for the random heuristic. We note that, *Smith-Waterman* and *Wikipedia* are the best heuristics we considered to find out the correct parent-child relationships for both SPLOT and PCM-based FMs. They put for SPLOT FMs about 48% of the correct parents in the top 2 positions of the parent candidate lists and 60% for PCM-based FMs.

For SPLOT FMs, we have similar results, for the edit distance between the original FM and the FM generated by our optimum branching algorithm. The FMs generated by our random heuristic require 0.638 operations per feature to obtain the original FM. The FMs generated by our other heuristics require about 0.585 operations per feature. As shown in Table 3, with *Wikipedia* and *Wiktionary* we get the lowest mean edit distance (0.57). For PCM-based FMs, we obtained more significant difference between edit distance acheived by random heuristic (0.60) and by our heuristics (0.51). *PathLength* gives the lowest mean edit distance (0.48).

For the clusters of SPLOT FMs, the difference between our heuristics and the random one is more significant compared to the number of correct parents. Our heuristics produce from 49% to 66% of correct clusters while the random heuristic produces 18%. The correct clusters represent more than 600 features for most of our heuristics while the random heuristic reach only 325 features. *PathLength* is the most relevant heuristic for clustering, reaching about 1.74 correct features per cluster. For the clusters of PCM-based FMs, our heuristics produce from 49.6% to 78% of correct clusters while the random heuristic produces 20%. *PathLength* is also the most relevant heuristic for clustering of PCM-based FMs with 2.25 as the average number of correct features per cluster.

As a result, we can notice that there is not a best heuristic. Each one can be the most appropriate for a precise FM. However our ontological heuristics generate FMs that are closer to the ground truth than FMs provided by a random heuristic. They also generate less cluster but they are more accurate. The clusters allow the user to take decisions for several features simultaneously. He has to choose a parent for these features only by looking at a single list that contains their common parent candidates. The user have fewer parent candidates to analyze and, as we see in Table 3, it results in fewer choices. Therefore, the parent candidate lists and the clusters of our heuristics reduce the user effort.

6.4 Qualitative Observations and Discussion

We showed that breathing ontological knowledge into FM synthesis improve the quality of the generated FMs. However, we noticed some limits in our techniques. We report some of them here.

Preprocessing. The heuristics based on WordNet and Wikipedia ontologies are very sensitive to preprocessing. The preprocessing aim at separating words and formatting them in an understandable way for the heuristics. Thus, it increases the number of recognized words and improve the results.

Combination of heuristics. There is no best heuristic among the ones we developed. Each heuristic has its own strengths and weaknesses. As we stated in Section 4.2, WordNet contains few technical words. Furthermore, Wikipedia may not contain very specific words or abbreviations and syntactical heuristics do not handle semantical links between words. Therefore, the scope of our heuristics are different. Combining them could extend their scope and improve the quality of the synthesized FM. However, their values cannot be compared directly.

A very low or high value is the only reliable information. It is significant enough to respectively reject or select a parent candidate. From this observation we compute the BIG's weights with a main heuristic thus having comparable values for all the edges. Then, we compute the weights with a set of complementary heuristics. If the heuristics are not conflicting, the weight of a parent candidate with a very low (resp. high) probability is assigned to 0 (resp. 1). This algorithm allow us to gather information from several heuristics while maintaining comparable values for the optimum branching algorithm.

For example, we use the Smith-Waterman metric as the main heuristic thus having a weight for every edge of the BIG. Then we complete these weights with the information from Wikipedia in order to reveal the semantical relations that were not present in the words' morphology.

We did not investigate the effectiveness of this process. However, we believe that a general ontology like the ones studied in this report could benefit from more specific ontologies using other artefacts.

Role of clusters. We conducted an additional test computing the optimum branching with each heuristic and Levenshtein as a clustering heuristic. Combining the parent candidate lists and the clusters obtained by the additional heuristic barely increase the quality of the FMs. Only the random heuristic benefits from this new ontological information. One might say that the clusters are useless but we proved that they reduce the user effort by reducing the required number of choices to complete the FM. This is why we gave equal importance to the parent candidate lists and the clusters in our interactive environment.

Lessons learned: Subjectivity of dataset and Importance of the user. We saw bad modeling pratices and lots of subjective choices in the FMs of our data sets. Therefore, the data sets considered are said to be realistic but the ground truth can be questionable. Moreover, our heuristics can be used to detect bad smells in FMs and suggest accordingly some refactoring opportunities. In the case of subjective choices, only the user can decide what is the right parent for his final application in the FM. Thus, a completely automated FM synthesis is not possible nor desirable.

6.5 Threats to Validity

A first *internal threat* is that the extraction of FMs from PCMs is a mix between automated techniques and manual directives. Their ontological semantics could be considered as optimized for our techniques. Yet we retain the original structure of PCMs and the manual intervention was done to interpret variability information. Furthermore we confirmed our results with SPLOT FMs that were produced by SPL practitioners.

A second threat comes from the manual optimization of the clustering thresholds for the evaluation of the heuristics. Another set of thresholds could generate less favourable results. It is unclear whether this difference would be significant.

Threats to *external validity* are conditions that limit our ability to generalize the results of our automated techniques to variability systems or forms of dependencies.

As we apply part of our procedures to Wikipedia PCMs dataset, one might perceive that some of the heuristics, based also on Wikipedia, are biased. However the heuristics do not operate over Wikipedia pages where we extracted the PCM. We exploit the Wikipedia as a general ontology and there is no connection with PCMs.

Another threat is that we hypothesize that the user effort is reduced thanks to clusters computation, ranking lists and the branching algorithm. A preliminary pilot study showed the prior importance of the interactive synthesis environment. A bad usability can prevent the usage of the clusters or the lists and alleviate the benefits of our proposals. Thus, our immediate concern is to improve our environment and run user experiments to validate our theoretical results.

7 Conclusion

7.1 Summary

In this paper, we addressed the problem of synthesising a feature model conformant to a set of dependencies and also exhibiting an appropriate ontological semantics as defined by its hierarchy and feature groups. This problem is crucial for many product line engineering scenarios involving operations like reverse engineering, slicing, merging or refactoring of feature models. We developed and evaluated a series of automated techniques to *i)* rank possible parent-child relationships between features and *ii)* compute clusters of conceptually similar features.

Our evaluation on hundreds of feature models showed that the ranking list and the clusters can be exploited to breath ontological knowledge during the synthesis of a feature model. The number of parents to choose decrease by 10% to 20% in the best case and for at least 45% of the features, the user has to consider only 2 parent candidates. The results encouraged us to develop an ontologic-aware synthesis environment and equip important FM automated operations (like refactoring, diff, merging, slicing) with ontological capabilities. Our experiments also demonstrated that the role of the user is crucial since it is neither realistic nor feasible to compute a high quality model without her intervention.

7.2 Future Work

Usability of the tool. Our immediate concern is to improve our tool support and conducts user experiments to evaluate the effort saved by our procedures in practical reverse engineering or maintenance settings. We conducted a preliminary pilot study that showed the prior importance of the interactive synthesis environment, i.e., a bad usability can alleviate the benefits of our proposals.

Specific vs general heuristics. A second research direction is to further exploit *specific* information sources and artefacts (e.g., domain-specific ontologies, textual corpus or requirements, source code) that may be present in software projects to automatically capture and breath ontological knowledge. Additional artefacts are necessary to improve the quality of the generated FMs from realistic projects.

Quality of an FM. A third research direction is to use our heuristics to evaluate the quality of an FM. By now, our heuristics are used for recommending feature relations during the synthesis of an FM. We could also use them on an existing FM to detect bad modeling constructs or inconsistencies in its hierarchy. Futhermore, we could recommend refactoring operations to improve the FM quality.

Extraction and evolution in real projects. Our last research direction is to develop automated techniques that helps understanding and maintaining the evolution of a project. No studies were conducted on multiple versions of a same project. However, each modification in the requirements, the documentation or the source code may change the variability. Therefore, monitoring variability over different versions is crucial for maintaining the quality of a project. We plan to evaluate these techniques on large open source projects like Eclipse, Linux, Emacs, ffmpeg or BusyBox.

We believe the combined exploitation of artefacts, user knowledge and automated techniques is the key to re-engineer feature-rich systems more and more reported in the industry or observed in the open source community.

A Illustration of the environment on the running example

We illustrate a typical usage of the environment[11] and the techniques from Section 4 on the running example in Figure 2.

First, we start FAMILIAR and load a formula or an FM. Then, we execute the command *ksynthesis --interactive fm* (with fm replaced by the name of the loaded FM). This command starts the synthesis environment and the user can see the interface in Figure 12.

The next step consists in setting the parameters of the synthesis in the *synthesis* menu (see Figure 12). For this example, we choose Wikipedia Miner with the Wikipedia database as the heuristic for parent candidates, Smith-Waterman as the clustering metric and we set the threshold at 0.6. After each change, the parent candidates lists and the clusters are updated.

We start synthesizing the FM by choosing a parent for the feature Proprietary License. We open the list of parent candidates for this feature and see that License is the best parent according to our heuristic which seems correct (see Figure 13a). Thus, we right-click on License and select the option *Select this parent*. As a result, the features and their new relation appears on the FM overview (see Figure 13b).

We continue our synthesis by choosing a parent for the cluster {PostgreSQL, MySQL}. We know by experience that these features are two types of database and should be siblings. We right-click on the cluster and select the only option available (see Figure 14a). A popup window appears asking for the parent of the selected cluster (see Figure 14b). At this point, we can deselect some features that should not be part of this clusters. The list of common parent that appears below is automatically updated and we can choose the parent for this subcluster. In our case we keep all the features selected and we choose Storage as the parent. Once again, the FM is updated and as the two previous choices do not form a single tree, two trees are displayed side by side (see Figure 14c).

The other available operation on a cluster is to select its parent within the cluster's features. The same operation is available on cliques. We consider the clique {Storage, License, Wiki, Hosting}. This time, to choose Wiki as the parent of the clique, we right-click on the feature and select the only option displayed (see Figure 15a). A popup window appears asking for the children of Wiki (see Figure 15b). We confirm the choice and the FM is updated resulting in a single tree (see Figure 15c).

At this point of the synthesis, we start to recognize the hierarchy of the desired FM in Figure 2. Thus, we try to generate the rest of the hierarchy by selecting the *Complete FM* option in the synthesis menu (see Figure 12). The result in Figure 16 is not satisfactory. The heuristics we set at the beginning of the synthesis do not perform well on the other features. For example they propose Programming Language as the child of Storage instead of Wiki.

In that case, we can change the heuristics and the clustering threshold to influence the automated synthesis or we can continue to provide information by choosing or ignoring a parent in the remaining features. This example illustrates that synthesizing a FM is an iterative process. Moreover the order of the choices may differ from one user to another. We could adopt a top down approach by first defining the root and its descendants or a bottom up approach by first defining the leaves of the hierarchy and finally setting the root. We can also adopt an unordered approach like in our example. This diversity of approaches forces us to present all the pieces of information at the same time instead of presenting one feature at a time.

[11] A complete tutorial of the environment is available in the following web page: `https://github.com/FAMILIAR-project/familiar-documentation/tree/master/manual/ontologicalTutorial`

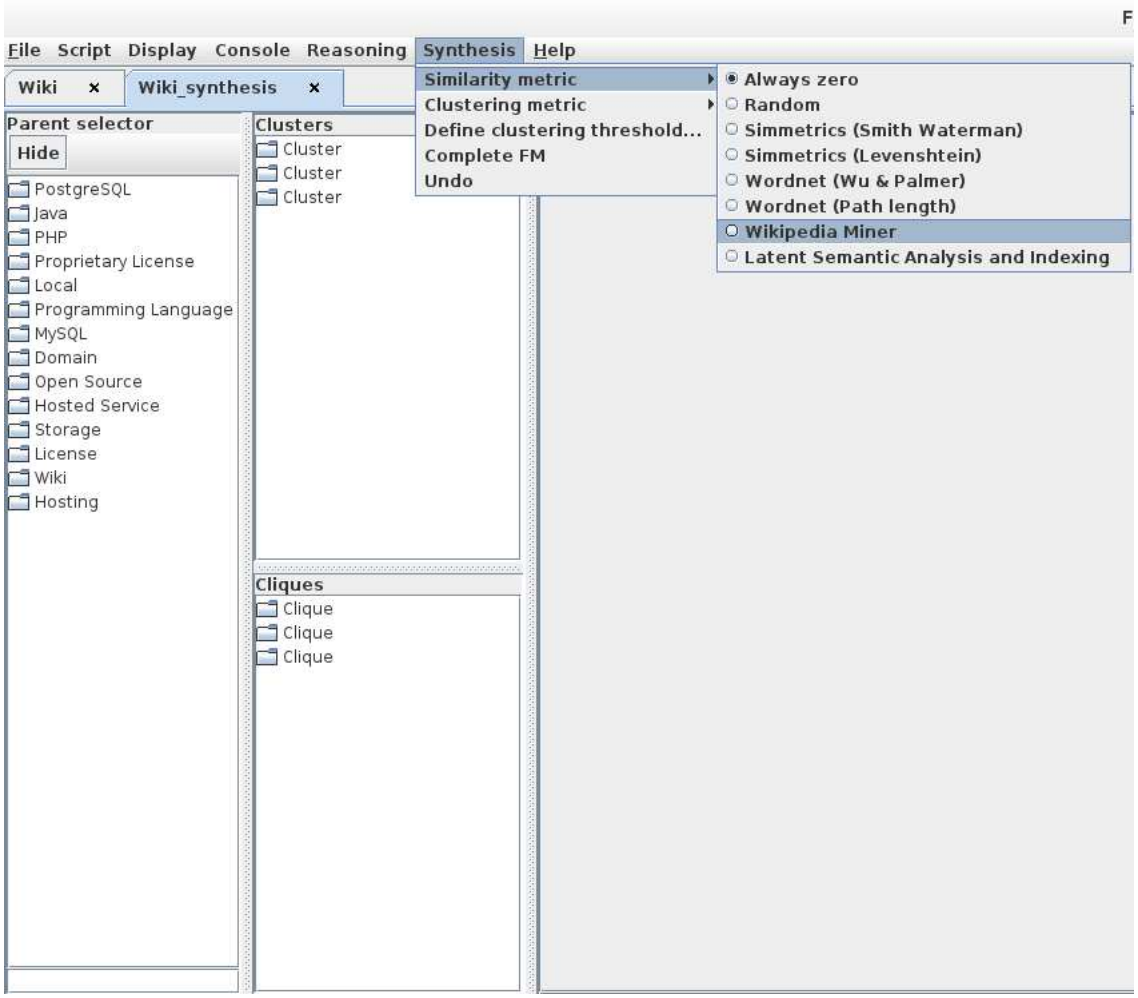
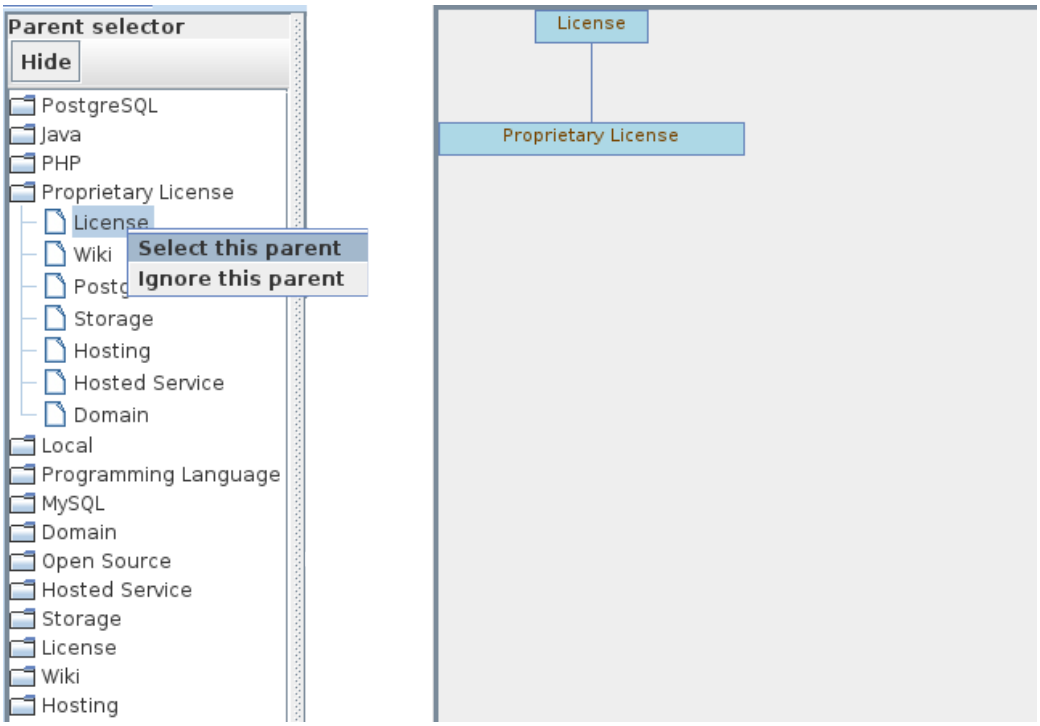
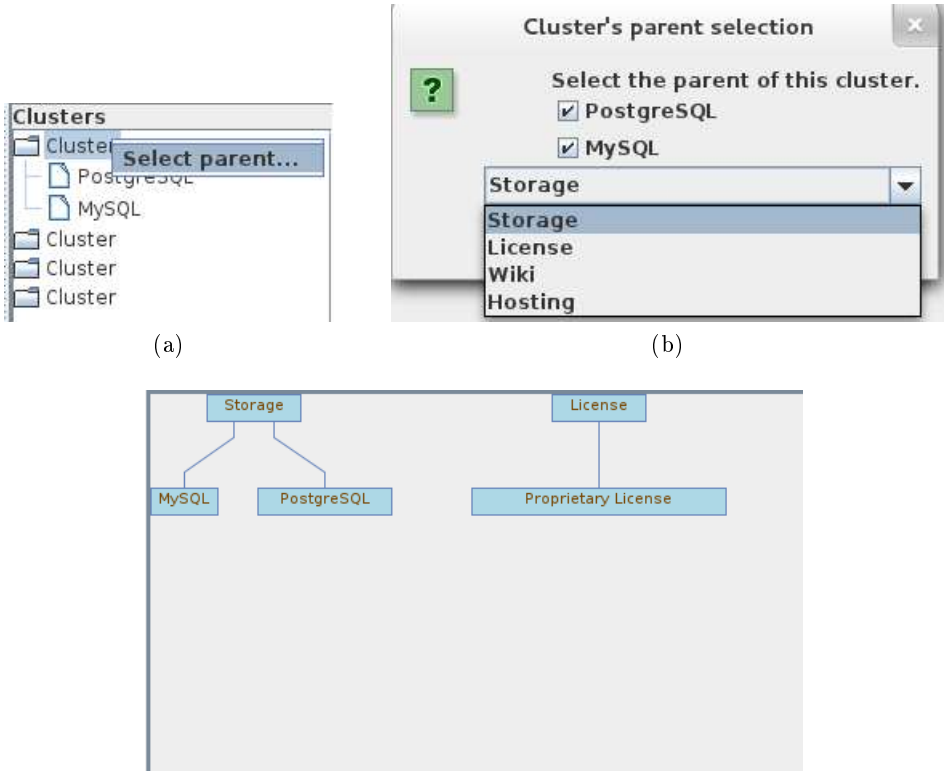


Figure 12: The environment at the beginning of the synthesis and the synthesis menu.



(a) (b)

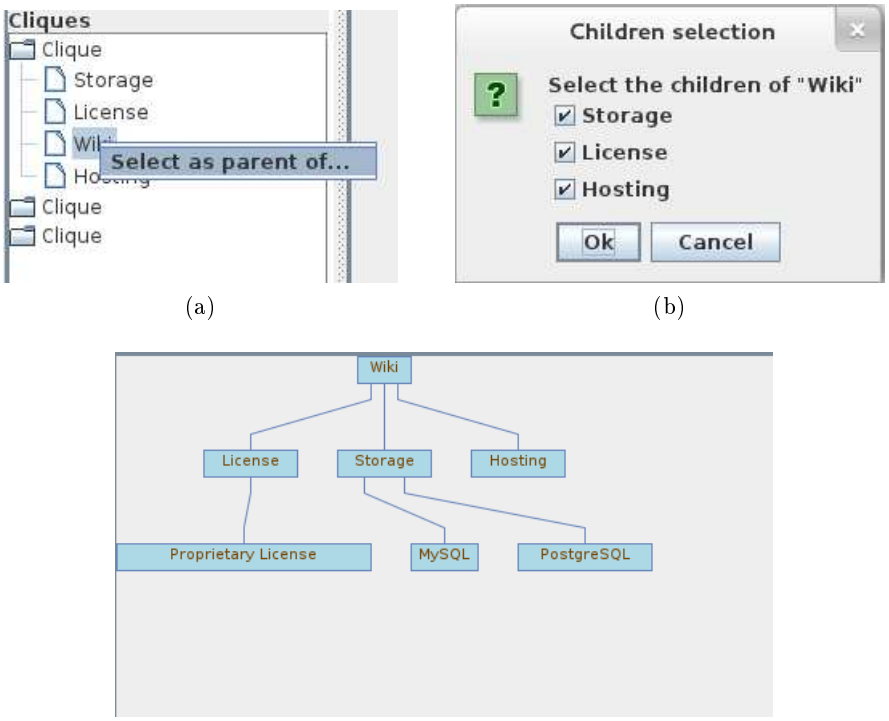
Figure 13: Selection of a parent for a feature and its effect on the FM.



(a) (b)

(c)

Figure 14: Selection of a parent for a cluster and its effect on the FM.



(a) (b)

(c)

Figure 15: Selection of a parent for a clique and its effect on the FM.

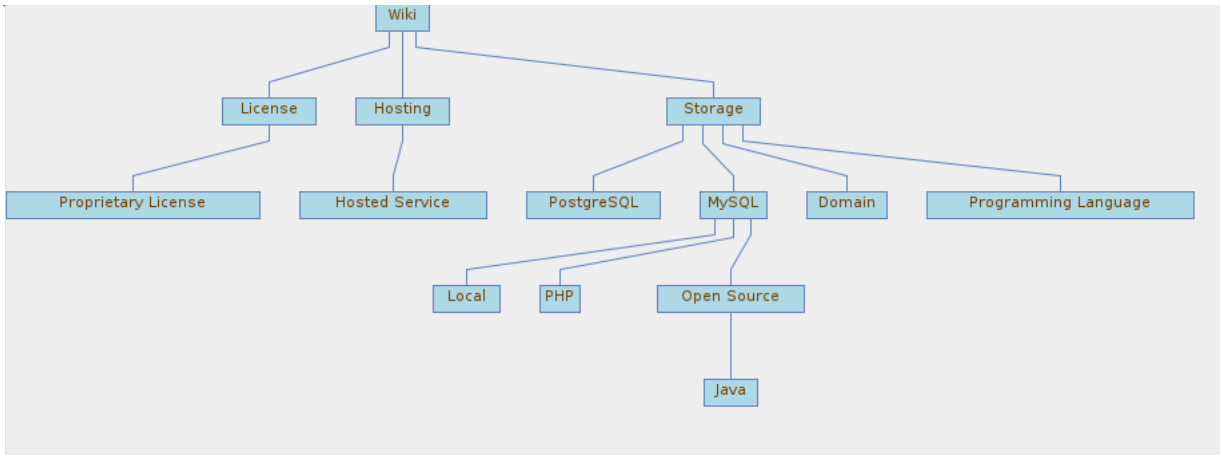


Figure 16: Generated FM according to previous choices and heuristics.

References

[1] The anatomy of a sales configurator: An empirical study of 111 cases. In *CAiSE'13*.

[2] Sei's software product line hall of fame.

[3] Mathieu Acher, Anthony Cleve, Philippe Collet, Philippe Merle, Laurence Duchien, and Philippe Lahire. Reverse engineering architectural feature models. In *ECSA'11*, volume 6903 of *LNCS*, pages 220–235, 2011.

[4] Mathieu Acher, Anthony Cleve, Gilles Perrouin, Patrick Heymans, Charles Vanbeneden, Philippe Collet, and Philippe Lahire. On extracting feature models from product descriptions. In *VaMoS'12*, pages 45–54. ACM, 2012.

[5] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert France. Familiar: A domain-specific language for large scale management of feature models. *Science of Computer Programming (SCP) Special issue on programming languages*, page 22, 2013.

[6] Mathieu Acher, Patrick Heymans, Anthony Cleve, Jean-Luc Hainaut, and Benoit Baudry. Support for reverse engineering and maintaining feature models. In *VaMoS'13*, Pisa, Italie, jan 2013. ACM.

[7] Vander Alves, Christa Schwanninger, Luciano Barbosa, Awais Rashid, Peter Sawyer, Paul Rayson, Christoph Pohl, and Andreas Rummler. An exploratory study of information retrieval techniques in domain analysis. In *SPLC*, pages 67–76. IEEE Computer Society, 2008.

[8] Nele Andersen, Krzysztof Czarnecki, Steven She, and Andrzej Wasowski. Efficient synthesis of feature models. In *Proceedings of SPLC'12*, pages 97–106. ACM Press.

[9] Sven Apel and Christian Kästner. An overview of feature-oriented software development. *Journal of Object Technology (JOT)*, 8(5):49–84, July/August 2009.

[10] Sven Apel, Christian Kästner, and Christian Lengauer. FeatureHouse: Language-independent, automated software composition. In *ICSE'09*. IEEE Computer Society, May 2009.

[11] Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Größlinger, and Dirk Beyer. Strategies for product-line verification: Case studies and experiments. In D. Notkin, B. H. C. Cheng, and K. Pohl, editors, *ICSE 2013*. IEEE, 2013.

[12] D. Batory. Feature models, grammars, and propositional formulas. *Software Product Lines*, pages 7–20, 2005.

[13] D. Batory. Feature modularity for product-lines. *Tutorial at: OOPSLA*, 6, 2006.

[14] D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Automated reasoning on feature models. *LNCS, Advanced Information Systems Engineering: 17th International Conference, CAiSE 2005*, 3520:491–503, 2005.

[15] Berger, Thorsten and Rublack, Ralf and Nair, Divya and Atlee, Joanne M. and Becker, Martin and Czarnecki, Krzysztof and Wasowski, Andrzej. A survey of variability modeling in industrial practice. In *VaMoS'13*. ACM, 2013.

[16] Quentin Boucher, Ebrahim Abbasi, Arnaud Hubaux, Gilles Perrouin, Mathieu Acher, and Patrick Heymans. Towards more reliable configurators: A re-engineering perspective. In *Third International Workshop on Product LinE Approaches in Software Engineering at ICSE 2012 (PLEASE'12)*, , Zurich, jun 2012.

[17] Quentin Boucher, Andreas Classen, Paul Faber, and Patrick Heymans. Introducing TVL, a text-based feature modelling language. In *Int'l Workshop on Variability Modelling of Software-intensive Systems (VaMoS'10)*, pages 159–162, 2010.

[18] Alexander Budanitsky and Graeme Hirst. Evaluating wordnet-based measures of lexical semantic relatedness. *Computational Linguistics*, 32(1):13–47, 2006.

[19] Guillaume Bécan. Reverse engineering feature models in the real. *Bibliographic report of master research internship*, 2013.

[20] PM Camerini, L Fratta, and F Maffioli. A note on finding optimum branchings. *Networks*, 9(4):309–312, 1979.

[21] Kun Chen, Wei Zhang, Haiyan Zhao, and Hong Mei. An approach to constructing feature models based on requirements clustering. In *RE'05*, pages 31–40.

[22] A. Classen, P. Heymans, and P.Y. Schobbens. What's in a feature: A requirements engineering perspective. *Fundamental Approaches to Software Engineering*, pages 16–30, 2008.

[23] Andreas Classen, Quentin Boucher, and Patrick Heymans. A text-based approach to feature modelling: Syntax and semantics of TVL. *Science of Computer Programming, Special Issue on Software Evolution, Adaptability and Variability*, 76(12):1130–1143, 2011.

[24] Paul Clements and Linda M. Northrop. *Software Product Lines : Practices and Patterns*. Addison-Wesley Professional, 2001.

[25] K. Czarnecki, P. Grünbacher, R. Rabiser, K. Schmid, and A. Wąsowski. Cool features and tough decisions: A comparison of variability modeling approaches. In *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*, pages 173–182. ACM, 2012.

[26] K. Czarnecki, S. She, and A. Wasowski. Sample spaces and feature models: There and back again. In *Software Product Line Conference, 2008. SPLC'08. 12th International*, pages 22–31. IEEE, 2008.

[27] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[28] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Formalizing Cardinality-based Feature Models and their Specialization. In *Software Process Improvement and Practice*, pages 7–29, 2005.

[29] Krzysztof Czarnecki, Chang Hwan Peter Kim, and Karl Trygve Kalleberg. Feature models are views on ontologies. In *SPLC '06*, pages 41–51. IEEE, 2006.

[30] Krzysztof Czarnecki and Krzysztof Pietroszek. Verifying feature-based model templates against well-formedness ocl constraints. In *GPCE'06*, pages 211–220. ACM, 2006.

[31] Krzysztof Czarnecki and Andrzej Wasowski. Feature diagrams and logics: There and back again. In *SPLC'07*, pages 23–34. IEEE, 2007.

[32] Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. A robust approach for variability extraction from the linux build system. In Eduardo Santana de Almeida, Christa Schwanninger, and David Benavides, editors, *SPLC (1)*, pages 21–30. ACM, 2012.

[33] H. Dumitru, M. Gibiec, N. Hariri, J. Cleland-Huang, B. Mobasher, C. Castro-Herrera, and M. Mirakhorli. On-demand feature recommendations derived from mining public product descriptions. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 181–190. IEEE, 2011.

[34] Evelyn Nicole Haslinger, Roberto E. Lopez-Herrejon, and Alexander Egyed. Reverse engineering feature models from programs' feature sets. In *WCRE'11*, pages 308–312. IEEE CS, 2011.

[35] Evelyn Nicole Haslinger, Roberto Erick Lopez-Herrejon, and Alexander Egyed. On extracting feature models from sets of valid feature combinations. In Vittorio Cortellessa and Dániel Varró, editors, *FASE*, volume 7793 of *Lecture Notes in Computer Science*, pages 53–67. Springer, 2013.

[36] Florian Heidenreich, Pablo Sanchez, Joao Santos, Steffen Zschaler, Mauricio Alferez, Joao Araujo, Lidia Fuentes, Uira Kulesza amd Ana Moreira, and Awais Rashid. Relating feature models to other models of a software product line: A comparative study of featuremapper and vml*. *Transactions on Aspect-Oriented Software Development VII, Special Issue on A Common Case Study for Aspect-Oriented Modeling*, 6210:69–114, 2010.

[37] Mikolás Janota, Victoria Kuzina, and Andrzej Wasowski. Model construction with external constraints: An interactive journey from semantics to syntax. In *MODELS'08*, volume 5301 of *LNCS*, pages 431–445, 2008.

[38] 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, DTIC Document, 1990.

[39] Frank J. van der Linden, Klaus Schmid, and Eelco Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[40] Marcilio Mendonca, Andrzej Wasowski, and Krzysztof Czarnecki. SAT-based analysis of feature models is easy. In *SPLC'09*, pages 231–240. IEEE, 2009.

[41] Andreas Metzger, Klaus Pohl, Patrick Heymans, Pierre-Yves Schobbens, and Germain Saval. Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. In *RE'07*, pages 243–253, 2007.

[42] George A Miller. Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.

[43] David Milne. Computing semantic relatedness using wikipedia link structure. In *the new zealand computer science research student conference.* Citeseer, 2007.

[44] Nan Niu and Steve M. Easterbrook. Concept analysis for product line requirements. In Kevin J. Sullivan, Ana Moreira, Christa Schwanninger, and Jeff Gray, editors, *AOSD*, pages 137–148. ACM, 2009.

[45] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer-Verlag, 2005.

[46] Richard Pohl, Kim Lauenroth, and Klaus Pohl. A performance comparison of contemporary algorithmic approaches for automated analysis operations on feature models. In Perry Alexander, Corina S. Pasareanu, and John G. Hosking, editors, *ASE*, pages 313–322. IEEE, 2011.

[47] Ariel Rabkin and Randy Katz. Static extraction of program configuration options. In *ICSE'11*, pages 131–140. ACM, 2011.

[48] Julia Rubin and Marsha Chechik. Locating distinguishing features using diff sets. In *the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 242–245, New York, NY, USA, 2012. ACM.

[49] Julia Rubin and Marsha Chechik. *Domain Engineering: Product Lines, Conceptual Models, and Languages (editors: Reinhartz-Berger,I. and Sturm, A. and Clark, T. and Bettin, J. and Cohen, S.)*, chapter A Survey of Feature Location Techniques. Springer, 2013.

[50] Uwe Ryssel, Joern Ploennigs, and Klaus Kabitzsch. Extraction of feature models from formal contexts. In *FOSD'11*, pages 1–8, 2011.

[51] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. Generic semantics of feature diagrams. *Comput. Netw.*, 51(2):456–479, 2007.

[52] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. Reverse engineering feature models. In *ICSE'11*, pages 461–470. ACM, 2011.

[53] TF Smith and MS Waterman. Identification of common molecular subsequences. *Molecular Biology*, 147:195–197, 1981.

[54] M. Svahnberg, J. Van Gurp, and J. Bosch. A taxonomy of variability realization techniques. *Software: Practice and Experience*, 35(8):705–754, 2005.

[55] Robert E Tarjan. Finding optimum branchings. *Networks*, 7(1):25–35, 1977.

[56] Sahil Thaker, Don Batory, David Kitchin, and William Cook. Safe composition of product lines. In *GPCE '07*, pages 95–104, New York, NY, USA, 2007. ACM.

[57] Thomas Thüm, Don Batory, and Christian Kästner. Reasoning about edits to feature models. In *ICSE'09*, pages 254–264. ACM, 2009.

[58] Marco Tulio Valente, Virgilio Borges, and Leonardo Passos. A semi-automatic approach for extracting software product lines. *IEEE Transactions on Software Engineering*, 38(4):737–754, 2012.

[59] Robert A Wagner and Michael J Fischer. The string-to-string correction problem. *Journal of the ACM (JACM)*, 21(1):168–173, 1974.

[60] N. Weston, R. Chitchyan, and A. Rashid. A framework for constructing semantically composable feature models from natural language requirements. In *SPLC'09*, pages 211–220. ACM, 2009.

[61] Zhibiao Wu and Martha Palmer. Verbs semantics and lexical selection. In *the 32nd annual meeting on Association for Computational Linguistics*, pages 133–138. Association for Computational Linguistics, 1994.

[62] Ora Wulf-Hadash and Iris Reinhartz-Berger. Cross product line analysis. In *the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, VaMoS '13, pages 21:1–21:8, 2013.

[63] Kaizhong Zhang and Dennis Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM journal on computing*, 18(6):1245–1262, 1989.