Feature Model Differences

Mathieu Acher[1], Patrick Heymans[1,2], Philippe Lahire[3], Clément Quinton[2], Philippe Collet[3], and Philippe Merle[2]

[1] PReCISE Research Centre, University of Namur, Belgium
{mac, phe}@info.fundp.ac.be
[2] INRIA Lille-Nord Europe, LIFL CNRS UMR 8022, University of Lille 1, France
{clement.quinton, philippe.merle}@inria.fr
[3] I3S – CNRS UMR 6070 University of Nice Sophia Antipolis, France
{collet, lahire}@i3s.unice.fr

Abstract. Feature models are a widespread means to represent commonality and variability in software product lines. As is the case for other kinds of models, computing and managing feature model differences is useful in various real-world situations. In this paper, we propose a set of novel differencing techniques that combine syntactic and semantic mechanisms, and automatically produce meaningful differences. Practitioners can exploit our results in various ways: to understand, manipulate, visualize and reason about differences. They can also combine them with existing feature model composition and decomposition operators. The proposed automations rely on satisfiability algorithms. They come with a dedicated language and a comprehensive environment. We illustrate and evaluate the practical usage of our techniques through a case study dealing with a configurable component framework.

Keywords: Feature Models, Model Differences, Model Management

1 Introduction

Software product line (SPL) engineering aims at generating software variants tailored to the needs of particular customers or market segments [24]. SPL principles, formalisms and techniques are gaining more and more attention in different application domains to efficiently produce and maintain multiple similar software products. Central to SPL engineering is the modeling and management of variability: exploiting what variants have in common and managing what varies among them. In this context, *feature models* (FMs) are widely used to model the variability of a system in terms of mandatory, optional and exclusive features as well as propositional constraints over the features [25,28]. The primary purpose and semantics of FMs is to characterize the combinations of features (called configurations) supported by a system. A number of formalizations (e.g., [25,10]), automated reasoning operations [7] and tools (e.g., [20,4,27]) have been developed to address this issue. The formalism of FMs is now at the core of many generative or model-based approaches [9,24,23,11].

When managing the variability of an SPL, reasoning about the *differences* of two FMs is a prime concern: for instance, when an SPL, and therefore its FM, evolve over time. Even small edits to an FM, like moving a feature from one branch to another, can unintentionally change the set of valid feature combinations. Understanding the impact of the evolution of an FM, and thus the differences between two versions, is known to be impractical to determine manually [28]. Consequently tool support is required to assist practitioners in computing and understanding FM differences. In practice, the need to support FM differences has been observed in different domains and for different purposes.

Evolution of an FM. In [11,23], the authors report that evolution support becomes particularly important for engineering SPLs and other variability-intensive systems. They propose model-driven support at the feature level, using FM concepts [23]. Lotufo *et al.* study the evolution of the Linux kernel variability model [16]. They identify edit operations applied in practice and new automation challenges, including the detection of edits that break existing configurations. Differencing techniques are thus needed, for example, to identify what are the added and removed configurations. In [2], we developed an automated procedure to extract the FM of FraSCAti, a large component and plugin-based system. The handling of FM differences was needed to compare the automatically extracted FM with one that was elaborated manually by the main developer of FraSCAti. It is also needed to understand and validate the evolution of the FMs for different versions of FraSCAti.

Management of a product line (PL) offering. Two kinds of variability are usually distinguished in SPL engineering [24,21]: software variability, hidden from customers, as opposed to PL variability, visible to them. An important property of an SPL is *realizability*, that is, whether the set of products that the PL management decides to offer is fully covered by the set of products that the software platform allows to build. Symmetrically, the *usefulness* property is interesting for a product manager to identify unused flexibility of the software platform. In this case, product managers and software engineers need to precisely understand what are the products supported by the platform but not offered to customers (it can be on purpose and justified by future market extensions). The differencing information is then exploited by product managers and software engineers to validate or evolve the FMs documenting the two kinds of variability. In the development of a video surveillance SPL and medical imaging workflows, we observed similar differentiation needs [6].

Until now, the problem of FM differences has neither been recognized nor comprehensively addressed by existing approaches. Model-based approaches mostly rely on syntactic mechanisms, basing their heuristics on the names and structure of model elements [22]. While showing some success, there are serious limitations. Models that are syntactically very similar may actually have very different semantics (intended meaning), and vice versa, models that describe the same system may have very different syntactic representations [19,17,13]. This observation also applies to FMs [27]. As a result, a list of syntactic differences, although accurate and useful, may not be able to reveal the real and meaning-

ful implications these differences have on the models involved. Some voices are calling for more *semantic differencing* [19,13] and such techniques have recently emerged for specific modeling formalisms [17,18]. In the feature modeling community, Benavides *et al.* do not report any differencing support in their survey, despite the impressive research effort on FM automations [7]. Only a few recent works have specifically considered the problem of FM differences at the syntactic [26] or at the semantic level [28,13] but have limitations to compute and present exploitable differences. In previous work, we developed a set of semantic techniques for FMs [3,2,4,6] but not yet for differences.

In this paper we first present and illustrate the problem of FM differences (Section 2). We develop a set of syntactic and semantic techniques to compute, reason about and present differences (Section 3). Thanks to these techniques, a practitioner can understand differences in a fine-grained way, visualize and manage a model of differences, augment an existing FM with the differences or compute the differences only on some parts of the two FMs. The techniques are automated using satisfiability algorithms and come with a dedicated language (Section 3.4). We also report on a practical usage and evaluation of the differencing techniques through a case study (Section 4). We discuss related work (Section 5) and conclude the paper (Section 6).

2 The Problem of Feature Model Differences

2.1 Background

FMs hierarchically structure application features into multiple levels of increasing detail. When decomposing a feature into subfeatures, the subfeatures may be optional or mandatory or may form *Xor*- or *Or*-groups (see Figure 1(a) for a visual representation of an FM). The terms FM and *feature diagram* are employed in the literature, usually to denote the same concept. In this paper, we consider that a feature diagram (see Definition 1) includes a feature hierarchy (tree), a set of feature groups, as well as human readable constraints (implies, excludes). The formalism of FMs considered is among the most popular in use.

Definition 1 (Feature Diagram) *A feature diagram* $FD = \langle G, E_{MAND}, G_{XOR}, G_{OR}, I, EX \rangle$ *is defined as follows:*

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

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. It should be noted that a feature diagram, as defined above, is not expressively complete with respect to propositional logics. Similar to [27], we thus consider that an FM is composed of a feature diagram *plus* a propositional formula ψ_{cst} (see Definition 2).

Definition 2 (Feature Model) *An FM is a tuple* $\langle FD, \psi_{cst} \rangle$ *where* FD *is a feature diagram and* ψ_{cst} *is a propositional formula over the set of features* $\mathcal{F}$*.*

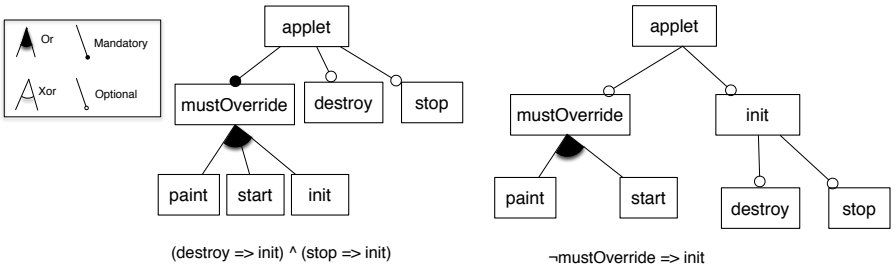
Not all combinations of features (*configurations*) are authorized by an FM. A *valid* (or legal) configuration is obtained by selecting features in a manner that respects the following rules: *i)* If a feature is selected, its parent must also be selected; *ii)* If a parent is selected, the following features must also be selected - all the mandatory subfeatures, exactly one subfeature in each of its Alternative groups, and at least one of its subfeatures in each of its Or groups; *iii)* propositional constraints must hold. An FM thus defines a set of valid feature configurations (see Definition 3).

Definition 3 (Configuration Semantics) *A configuration of an FM* fm_1 *is defined as a set of selected features.* $[\![fm_1]\!]$ *denotes the set of valid configurations of* fm_1 *and is a set of sets of features.*

An FM is usually encoded as a propositional formula, denoted ϕ, and defined over a set of Boolean variables, where each variable corresponds to a feature [10]. The translation to propositional logic is well known and off-the-shelf SAT solvers or *binary decisions diagrams* (BDDs) can be used to automatically reason about properties of an FM and its configurations [7,25,10].

2.2 Diff: A Running Example

We now illustrate with a simple example, extracted from [13], the problem of FM differences. Let us consider $applet_1$ (see Figure 1(a)) and $applet_2$ (see Figure 1(b)). Obviously, the two FMs differ, for example, feature init is a child feature of mustOverride in $applet_1$ whereas feature init is a child feature of the root feature applet in $applet_2$. This difference may occur in many scenarios already described in the introduction: the FM $applet_1$ has evolved over time, leading to $applet_2$; the two FMs have been reverse engineered from two existing frameworks ; $applet_1$ is a requirements model specified by a customer whereas $applet_2$ is the actual implementation model supported by an application programming interface, etc. A few questions arise: How do these two FMs, $applet_1$ and $applet_2$, differ? Are they equivalent? If not, what is the actual difference? Several applications of FM differences are conceivable. First, the difference may serve as a debugging information. For example, the four configurations satisfying $applet_1$ but not $applet_2$
$\{\{applet, destroy, init, mustOverride, stop\}$,
$\{applet, init, mustOverride, stop\}, \{applet, init, mustOverride\}$,



(a) $applet_1$ (b) $applet_2$

Fig. 1. Two example FMs extracted from [13]

$\{applet, destroy, init, mustOverride\}\}$ can be considered as a specification error. Second, examples of configurations allowed by $applet_1$ but not in $applet_2$ might be used to expand configurations of $applet_2$. Based on the analysis of differences, a practitioner may correct the error previously identified and relax some variability constraints and edit $applet_2$ (e.g., features paint and start become optional features and are no longer forming an Or-group). The example shows that a crucial issue for a practitioner, being modeler, product line manager, software engineer or architect, is to precisely understand and reason about differences of two FMs. Based on this information, a practitioner can validate the differences and/or evolve the FMs.

3 Set of Operators for Differencing of Feature Models

Let $fm_1 = \langle FD_1, \psi_{cst_1} \rangle$ with $FD_1 = \langle G_1 = (\mathcal{F}_1, E_1, r_1), E_{MAND_1}, G_{XOR_1}, G_{OR_1}, I_1, EX_1 \rangle$ and $fm_2 = \langle FD_2, \psi_{cst_2} \rangle$ with $FD_2 = \langle G_2 = (\mathcal{F}_2, E_2, r_2), E_{MAND_2}, G_{XOR_2}, G_{OR_2}, I_2, EX_2 \rangle$ be two FMs. In the reminder of this section, we consider the difference (*diff* for short) between fm_1 and fm_2 – in this particular order. We will use the example of Figure 1 to illustrate our contributions (i.e., $fm_1 = applet_1$ and $fm_2 = applet_2$).

Rationale and overview. Several techniques for the diff of FMs can be considered. Roughly, the diff between fm_1 and fm_2 is the set of elements in fm_1 but not in fm_2. From a syntactical perspective, the elements to be considered in the diff may be features, feature hierarchies, feature groups or implies / excludes. We present *syntactic differencing* techniques in Section 3.1. Though the syntactic diff might be useful, we believe that a semantic diff for FMs should also be developed and possibly be combined with syntactic differencing. Many researchers share this vision (e.g., see [28,19,13]) and semantic differences have already been developed specifically for other kinds of models (class diagrams [17] and activity diagrams [18]). We present *semantic differencing* techniques in Section 3.2. Rather than directly computing and reasoning about differences, it can be useful to focus on the common parts of two FMs. Another developed technique is to provide the means to focus on a specific part, typically smaller, of

the two FMs in order to facilitate the understanding of their differences. We describe a set of syntactic and semantic *composition* and *decomposition* techniques in Section 3.3 that comes in complement to the differencing techniques.

3.1 Syntactic Diff

A general approach to model differencing is to concentrate on matching between model elements using different heuristics related to their names and structure and on finding and presenting differences at a concrete or abstract syntactic level. Matching algorithms are out of the scope of this paper, surveys of different approaches can be found in [12,15]. We assume that features are identified by an unique label (i.e., name) in an FM and that two features of two FMs match if and only if they have the same name[4].

In terms of feature modeling, elements of interest are features, variability information (mandatory features, feature groups, and propositional constraints) and feature hierarchy (see Definition 1 and 2). We thus consider the diff of these model elements.

Diff of features. $\mathcal{F}_{diff}$ is the set of features that are in fm_1 but not in fm_2, i.e., $\mathcal{F}_{diff} = \mathcal{F}_1 \setminus \mathcal{F}_2$. In the example of Figure 1, $\mathcal{F}_{diff} = \emptyset$.

Diff of feature hierarchies. Several techniques can be considered (e.g., tree edit distance [8]), including the computation of E_{diff} the set of edges modeling parent-child relationships in fm_1 but not in fm_2. Formally: $E_{diff} = E_1 \setminus E_2$. In the example of Figure 1, $E_{diff} = \{\{mustOverride, init\}, \{applet, destroy\}, \{applet, stop\}\}$.

Diff of mandatory features. A syntactic diff of mandatory features produces $E_{MAND_{diff}} = E_{MAND_1} \setminus E_{MAND_2}$. In the example of Figure 1, $E_{MAND_{diff}} = \{\{applet, mustOverride\}\}$, showing that the feature mustOverride is mandatory in fm_1, which is not the case in fm_2.

Diff of feature groups. It is useful to determine feature groups (Xor and Or) that are in fm_1 but not in fm_2, including $G_{XOR_{diff}} = G_{XOR_1} \setminus G_{XOR_2}$ and $G_{OR_{diff}} = G_{OR_1} \setminus G_{OR_2}$. We consider that two feature groups are equal if and only if their parent features match and their child features match. In the example of Figure 1, $G_{XOR_{diff}} = \emptyset$ and $G_{OR_{diff}} = \{\{\{init, start, paint\}, mustOverride\}\}$

Diff of implies and excludes. A syntactic diff of implies (resp. excludes) constraints produces I_{diff} (resp. EX_{diff}) so that $I_{diff} = I_1 \setminus I_2$ (resp. $EX_{diff} = EX_1 \setminus EX_2$). In Figure 1, $I_{diff} = \{\{destroy \Rightarrow init\}, \{stop \Rightarrow init\}\}$ and $EX_{diff} = \emptyset$. It should be noted that I_{diff} is not semantically correct in this example: the features destroy and stop do imply the feature init in fm_2, owing to the parent-child relationships in fm_2.

3.2 Semantic Diff

Syntactic differences are useful for the example of Figure 1. Using the list of differences, a modeler can identify that the feature init has been moved or that

[4] This assumption is shared by [26,28,13] and holds for all the case studies presented in the introduction. The problem of *FM matching* is discussed in Section 4.2

the feature mustOverride becomes a mandatory feature in fm_2. However, a practitioner rather wants to understand the difference between the two FMs in terms of *configuration semantics* (i.e., in terms of sets of configurations). For this purpose, the list of syntactic differences fails to produce some differences, among others: *i)* the four configurations authorized by fm_1 but not by fm_2 are not identified ; *ii)* I_{diff} does not report that $stop \Rightarrow mustOverride$ holds in fm_1 but not in fm_2. With this information, a practitioner could learn that the feature stop does not imply the selection of the feature mustOverride in fm_2 whereas it is the case in fm_1.

To raise the limits of a syntactic diff, we address semantically the list of differences. We translate fm_1 and fm_2 into two formula ϕ_1 and ϕ_2. Nevertheless, performing at the level of abstraction of Boolean variables may produce unexploitable results. Stated differently, a practitioner wants to understand differences in terms of feature modeling concepts rather than in terms of a propositional formula. As a result, we take care of producing meaningful information based on the analysis of the two formula.

Diff of information extracted from the two formula. A first general strategy consists in analyzing separately each formula and then performs the differences of the information produced.

Diff of binary implication (resp. exclusion) graphs. We consider a binary implication graph of an FM and its propositional formula ϕ as a directed graph $BIG = (V_{imp}, E_{imp})$ formally defined as follows:

$$V_{imp} = \mathcal{F} \qquad E_{imp} = \{(f_i, f_j) \mid \phi \wedge f_i \Rightarrow f_j\} \tag{1}$$

Each binary, directed edge from feature f_i to feature f_j represents a binary implication. Based on the analysis of ϕ_1 and ϕ_2, we can produce BIG_1 and BIG_2 and then compute $BIG_{diff} = BIG_1 \setminus BIG_2$. It is then straightforward to compute the set of binary implications expressed in fm_1 but not in fm_2. In the example of Figure 1, $E_{impl_{diff}} = \{\{destroy \Rightarrow mustOverride\}, \{applet \Rightarrow mustOverride\}, \{stop \Rightarrow mustOverride\}, \{init \Rightarrow mustOverride\}\}$. As we support arbitrary propositional constraints in an FM, it should be noted that BIG_{diff} cannot be produced syntactically in the general case. Furthermore, the binary implication graph structure, reified from the propositional formula, has the advantage of exposing an information than can be directly translated in terms of feature modeling (i.e., either as a binary implication between a child feature and a parent feature or simply as an implies constraint). As a more general and powerful technique, the computation of BIG_{diff} should be used in favour of the syntactic diff of implies constraints.

Diff of binary exclusion graphs. Similarly, we can compute the set of binary exclusions expressed in fm_1 but not in fm_2. We consider a binary exclusion graph of an FM and its formula ϕ as an undirected graph, denoted BEG, consisting of vertices being features and edges denoting a mutual exclusion between two features f_i and f_j such that its formula ϕ entails $f_i \Rightarrow \neg f_j$. Then, $BEG_{diff} = BEG_1 \setminus BEG_2$. In the example of Figure 1, the set of edges of BEG_{diff} is empty.

Diff of cliques in implication and exclusion graphs. We extend the previous technique to n-ary bi-implications and n-ary mutual exclusions. A n-ary bi-implication involves n features such that $f_i \Rightarrow f_j$ for any $i, j = 1 \ldots n$. It can be obtained by computing cliques in BIG. (A clique in the implication graph is a subgraph in which any two vertices are connected by an edge). A n-ary mutual exclusion involves n features $f_1, \ldots, f_n$ and is detected if there exists a clique between $f_1, \ldots, f_n$ in BEG. (A clique in the exclusion graph requires each member to have an exclusion to every other member.) For the purpose of conciseness (no set of features is subsumed by other), we compute *maximal* cliques in BIG and BEG. In the example of Figure 1, we detect that features applet and mustOverride are bi-implied. We do not learn additional difference for this specific example as a syntactic technique already produces such information. Nevertheless the semantic technique is particularly suitable when there are complex cross-tree constraints in an FM.

Semantic diff of feature groups. Reasoning techniques can be performed on ϕ to detect candidate feature groups (Xor and Or-groups). It is based on an important property: several FMs can represent the same set of configurations while having different hierarchies and feature groups. As a result some feature groups are not syntactically restituted in a feature diagram[5]. For example, in fm_1, there are two candidate Or-groups $\{\{init,\ start,\ paint\}, mustOverride\}$, $\{\{init,\ start,\ paint\}, applet\}\}$, but only $\{\{init,\ start,\ paint\}, applet\}$ is included in G_{OR_1}. To compute candidate feature groups, we rely on techniques exposed in [10,27] that perform over a propositional formula.

Reasoning about the two formula. A second general strategy consists in producing relevant information based on the logical combinations of the two formula. We first describe two existing techniques [28,13] relevant for FM differences.

Relationship between two FMs. Thüm *et al.* [28] reason on the nature of FM edits, for example, when fm_1 is edited (e.g., some features are moved, added, or removed), giving fm_2. They provide a classification (see Definition 4) and an efficient algorithm to compute the kind of relationship between two FMs. In case the relationship is not a refactoring, the authors propose a technique to generate an example of configuration authorized in one but not in another.

Definition 4 (Edits) *fm_1 is a specialization of fm_2 if $[\![fm_1]\!] \subset [\![fm_2]\!]$; fm_1 is a generalization of fm_2 if $[\![fm_1]\!] \subset [\![fm_2]\!]$; fm_1 is a refactoring of fm_2 if $[\![fm_1]\!] = [\![fm_2]\!]$; fm_1 is an arbitrary edit of fm_2 in other cases.*

Quotient. In [13], an algorithm is presented that takes as input two formula ϕ_1 and ϕ_2 in *conjunctive normal form* (CNF) – FMs are easily converted to

[5] In fm_2, the features applet, mustOverride and init are semantically forming an Or-group. This is not syntactically restituted in the feature diagram (see Figure 1(b)) and is considered as an *anomaly* in the literature [7]. This example, extracted from [13], can be seen as an additional argument in favour of a diff performing at the semantic level.

CNF. The algorithm finds for the quotient (i.e., difference) all clauses in ϕ_1 which are not entailed by ϕ_2 through the satisfiability checks of $\phi_2 \wedge \neg c$ (c being a clause of ϕ_1). As recognized, this is clearly an over-approximation of the difference, but might fail at maximality. In the example of Figure 1, the quotient is $\{\{init \Rightarrow mustOverride\}, \{applet \Rightarrow mustOverride\}\}$

Diff of Formula. The two previous techniques fail to comprehensively represent the difference of the two configuration sets. To raise the limitations, we develop a diff operator, noted $\oplus_\setminus$, that takes as input two FMs and produces a diff FM (i.e., $fm_{diff} = fm_1 \oplus_\setminus fm_2$). fm_{diff} is depicted in Figure 2(a). The following defines the semantics of this operator:

$$[\![fm_1]\!] \setminus [\![fm_2]\!] = \{x \in [\![fm_1]\!] \mid x \notin [\![fm_2]\!]\} = [\![fm_{diff}]\!] \qquad (M_1)$$

Computing the diff formula that encodes $[\![fm_{diff}]\!]$ is as follows:

$$\phi_{diff} = (\phi_1 \wedge not(\mathcal{F}_2 \setminus \mathcal{F}_1)) \wedge \neg(\phi_2 \wedge not(\mathcal{F}_1 \setminus \mathcal{F}_2))$$

not is a function that, given a non-empty set of features, returns the Boolean conjunction of all negated variables corresponding to features:

$$not(\{f_1, f_2, ..., f_n\}) = \bigwedge_{i=1..n} \neg f_i$$

The presence of negated variables is needed since we need to emulate the deselection of features that are in fm_1 (resp. fm_2) but not in fm_2 (resp. fm_1). Otherwise, two features, say $f_1 \in \mathcal{F}_1$ and $f_2 \in \mathcal{F}_2$ such that $f_1 \neq f_2$ (i.e., f_1 does not match f_2), can be combined to form a configuration, thereby violating the configuration semantics of Definition M_1. An important property of ϕ_{diff} is that each valid assignment (true/false values assigned to variables) corresponds to a valid configuration of $[\![fm_{diff}]\!]$. With regards to maximality, it thus outperforms the quotient technique.

The connection between the characterization of edits (see Definition 4) and the diff operator is expressed by Lemma 1 (according to set theory, $[\![fm_1]\!] \subseteq [\![fm_2]\!]$ is equivalent to $[\![fm_1]\!] \setminus [\![fm_2]\!] = \emptyset$).

Lemma 1 (Diff and Specialization/Refactoring) *fm_1 is a specialization or a refactoring of fm_2 if $(fm_1 \oplus_\setminus fm_2)$ characterizes no valid configurations.*

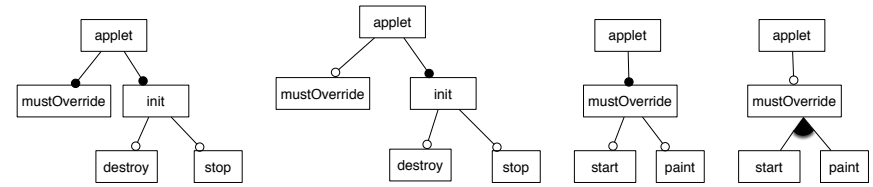
As a result, the satisfiability of ϕ_{diff} can be checked to determine the kind of relationship between fm_1 and fm_2. In the example of Figure 1, fm_1 is an arbitrary edit of fm_2 since $[\![fm_1]\!] \setminus [\![fm_2]\!] \neq \emptyset$ and $[\![fm_2]\!] \setminus [\![fm_1]\!] \neq \emptyset$.

From formula to an FM. Though the diff formula is useful for *reasoning*, we cannot render the formula "as is". Producing a complete FM (including the feature hierarchy, feature groups, etc.) is needed typically when an FM is visualized, serialized to a given format or when syntactic differencing techniques are applied. In this case, we need to transform the diff formula ϕ_{diff} as an FM. As stated above, several FMs can represent the same set of configurations [28,7,27]. Intuitively, we want to maximize the parent-child relations that occur in the two

input FMs. Furthermore not all feature hierarchies can be chosen (e.g., a hierarchy with start as a parent feature of mustOverride is too logically restrictive). The problem of choosing a hierarchy from amongst a set of hierarchies can be formulated as a minimum spanning tree problem over the binary implication graph of ϕ_{diff}. Based on the formula and the hierarchy, propositional logic techniques are applied to synthesize a complete FM (see [4]).

3.3 Composition and Decomposition of FMs

Computing the commonality and the union. It is interesting for a practitioner to determine the common properties of two FMs (rather than the differences). Syntactical techniques can be naturally applied (e.g., to determine common features). In previous work [3], we propose a semantic operator, called *merge*, that computes an FM whose set of configurations is the intersection of the two sets of configurations. Using the merged FM, a practitioner can enumerate or count the common set of configurations or simply visualize it (see below for more details). In the example of Figure 1, there are 15 common configurations. Another variant of the merge operator, denoted $\oplus_{\cup}$, can compute an FM representing the *union* of the two sets of configurations.



(a) $fm_{diff} = fm_1 \oplus_{\setminus} fm_2$ (b) $fm_{12_{comp}}$ (c) $\Pi_{\mathcal{F}_{foo}}(fm_1)$ (d) $\Pi_{\mathcal{F}_{foo}}(fm_2)$

Fig. 2. Diff FM, merge in union mode of two FMs, two slice FMs

Complementing FMs. The need to complement an existing FM, say fm_2, typically occurs when fm_2 is an underapproximation of fm_1 whereas it should not be the case. Therefore the differencing techniques are not only useful to *detect* an underapproximation but also to *compute* it and then *integrate* it in another model. Interestingly, we can combine the diff operator and the merge operators (in intersection or union mode). A possible application is, for instance, to compute an FM representing both configurations included in fm_2 but not in fm_1 and configurations included in fm_1 but not in fm_2. It can be formalized in set theory and therefore automated using the techniques previously described: $fm_{12_{comp}} = (fm_2 \oplus_{\setminus} fm_1) \oplus_{\cup} (fm_1 \oplus_{\setminus} fm_2)$. Such an FM can be used by practitioners to develop products not yet supported by existing solutions (i.e., neither by fm_1 not fm_2). $fm_{12_{comp}}$ is depicted in Figure 2(b).

Decomposing the problem of FM differences. Understanding and managing FM differences is a manual process and can quickly become difficult for a practitioner when a large number of features and constraints are involved. We thus propose to decompose the problem into subproblems. Intuitively, it is easier to focus on some parts of the two FMs rather than considering the two FMs in their entire form. In [5], we propose a semantic operator, called *slice*, that produces a projection of a FM (a slice) with respect to a set of selected features (slicing criterion). We define slicing as a unary operation on FM, denoted $\Pi_{\mathcal{F}_{slice}}\ (fm)$ where $\mathcal{F}_{slice} = \{ft_1, ft_2, ..., ft_n\} \subseteq \mathcal{F}$. The result of the slicing operation is a new FM, fm_{slice}, semantically defined in terms of configuration set: $[\![fm_{slice}]\!] = \{\ x\ \cap\ \mathcal{F}_{slice} \mid x \in [\![fm]\!]\}$ (called the *projected* set of configurations). In the context of FM differences, the slice operator can be applied on fm_1 and fm_2 using the same slicing criterion. For example, we apply the slice on fm_1 (the resulting FM is depicted in Figure 2(c)) and on fm_2 (the resulting FM is depicted in Figure 2(d)) using the slicing criterion $\mathcal{F}_{foo} = \{applet, mustOverride, start, paint\}$. The two slice FMs can then be analyzed using the differencing techniques exposed throughout the section.

3.4 Tool Support: Language and Environment

We rely on FAMILIAR (for *FeAture Model scrIpt Language for manIpulation and Automatic Reasoning*) a domain-specific language for FMs [4]. The language already includes facilities for composing/decomposing FMs, editing FMs (e.g., renaming and removal of features), reasoning about FMs (e.g., validity, comparison of FMs) and their configurations (e.g., counting or enumerating the configurations in an FM). FMs and other types (configuration, set, etc.) are manipulated using variables. Compared to [4], we extend the language and integrate the differencing techniques developed in the paper through the form of operations over FMs (quotient, computation of candidate feature groups and implication / exclusion graphs, etc.). Basic operators to perform the union, intersection or difference of variables are also provided. Two reasoning back-ends (SAT solvers using SAT4J and BDDs using JavaBDD) are internally used and perform over propositional formula to implement the semantic operators. An important property of the language is that operations can be sequentially executed while properties of the variables can be observed. Hence, complex management scenarios can be applied using FAMILIAR environment. A practitioner can decompose the two FMs, then apply some techniques to understand local differences, edit the FMs, and reiterate the process. At the end, the FMs including their differences can be serialized in different formats and be used to pilot configuration or a generative approach.

4 Evaluation

4.1 Complexity

Performing differences at the semantic level, though more powerful, has a cost. The computation of BIG and BEG heavily depends on *satisfiability* checks of

implications and exclusions. In practice, the computation of BIG and BEG scales for thousands of features and can be realized using SAT solvers or BDDs [10,27]. Due to transitivity of implication, maximal cliques (see Section 3.2) are actually strongly connected components in BIG, which can be found efficiently by graph traversal. Furthermore, the computation of cliques and Xor-groups scales for thousands of features [27]. To the best of our knowledge, the computation of Or-groups has only been implemented using BDDs. We rely on BDDs to synthesize a complete FM from a formula. (SAT solvers can be used but in this case we do not restitute Or-groups). The cost of feature diagram construction is polynomial regarding the size of the BDD [10]. We reuse the heuristics developed in [20] to reduce the size of the BDD – they scale up to 2000 features.

SAT solvers require a formula to be in CNF. Converting ϕ_{diff} into CNF requires ϕ_2 to be negated (see Equation M_1). As argued in [28], an exponential explosion of clauses occurs when ϕ_2 is negated, even for a small number of features. To avoid explosion, we rely on BDDs for computing and reasoning about ϕ_{diff}, since computing the disjunction, conjunction and negation of BDDs can be performed in at most polynomial time with respect to their sizes.

4.2 Applying differencing techniques: co-evolution of FMs

Case study. In [2], we presented a process for reverse engineering the FM of FraSCAti, a large and highly configurable component and plugin-based system. The overall challenge is to derive an FM so that its scope is not too large (otherwise some unsafe compositions are authorized) or too narrow (otherwise it is a symptom of unused flexibility). On the one hand, the FM produced by an automated procedure may not be an accurate representation, typically when FraSCAti artefacts do not correctly document the variability of the system. On the other hand, a software architect, while manually elaborating an intentional variability model of FraSCAti, may forget to specify some features or constraints. In order to manage (e.g., understand) the differences between the two FMs, we applied the techniques presented in the paper and the tool support.

Results and lessons learned. We now report *what* techniques have been used and *how* they helped to manage differences between the FMs. Further details and material (e.g., FAMILIAR scripts) about the case study are available in [1].

Implications. We first observed that the FMs involved have the following properties: an average of ≈ 50 features and $\approx 10^6$ configurations per FMs, and a large number of cross-tree constraints (implies). Therefore we made an extensive use of the diff between binary implication graphs. It allowed one to identify dozens of implies constraints expressed in one FM but not in another (and vice-versa). Implies constraints are very important in the FraSCAti case study. First, the software architect elaborates the FM and specifies an important number of binary implications. Second, the extraction procedure combines different sources of information, including plugin dependencies. These dependencies are essentially expressed through implies constraints. The diff between binary implication graphs has the merit of reifying the differences of the two FMs *in*

terms of implies constraints. It is then easier for a software architect to understand the impact of the difference: it is either an implication unintentionally not specified or an implication not documented by plugin dependencies. Compared to a syntactic diff, the major advantage of the structure of binary implication graph is the ability to derive *transitive* implications. The method of quotient produces some disjunctive clauses that can be transformed into implications. Nevertheless, we observed many times that the method suffers from a lack of completeness regarding the diff of implies constraints.

Or-groups vs Optional. We use a syntactic diff for computing feature groups expressed in one FM but not in another. A semantic diff for computing candidate feature groups, though more powerful in theory, does not produce additional information in this specific case study. The difference between feature groups concerns *Or-groups*. Indeed some features were modeled as optional features in the FM of the software architect whereas corresponding features formed an Or-group in the FM produced by the automated procedure. This difference, though subtle, occurs for three Or-Groups and has to be identified and managed (since in one case it is possible to not select any feature).

Decomposition. The compared FMs did not necessary have the same set of features (e.g., some features are added when a new version of FraSCAti is released). The features included in one FM but not in another disturb the management of differences. Intuitively, such features produce new configurations but we were mostly interested by the evolution of the common subset. Therefore we made an extensive use of the *decomposition* operator (i.e., slice, see Section 3.3) by focusing only on features commonly shared by the two FMs. We then applied the differencing techniques on the decomposed FMs. In particular, it made possible to determine if the common subset has correctly evolved and that no configuration has been broken (i.e., the refactoring or generalization property holds, see Definition 4).

Interactive process. Managing differences is not a simple one step-process. Once differences have been identified and understood, we edited FMs accordingly and reiterated the process until having satisfying FMs. Therefore the process is rather incremental and interactive. Automation and reproducibility of the operations are indeed crucial.

Opportunities for future work. The extracted FM and the FM elaborated by the software architect use different names for features that are actually similar. To avoid unexploitable differencing results, some predirectives were needed and consist in manually renaming features. More automated support seems desirable and *matching* techniques already integrated in model-based tools (e.g., see [12,15]) are good candidates. At the current state of the research, it is difficult to assess the significance of the FM matching problem in practice. Many techniques exposed in this paper can be combined on demand to manage differences. We observed that some information produced are sometimes redundant (e.g., the method of quotient detects a binary implication, already detected by the diff of binary implication graphs). To reduce the cognitive process of a practitioner, an interesting perspective is to *summarize* the differences (by aggregating informa-

tion produced by the differencing techniques so that there is no redundancy). We leave it as future work. Another direction for future work is to provide *guidelines* and a *methodological* proposal that could help non-experts to apply all these operators in practice. The usability of the approach (e.g., whether the produced information is understandable enough) should be *evaluated* accordingly using other case studies.

5 Related Work

Model differencing has attracted research efforts in recent years, including the development of tools (e.g., see [22,15,19,19,17,18]). The bibliography [22] compiles about 300 publications in this field. Existing approaches mainly focus on *syntactical* differences. As argued in [19,13], models (e.g., FMs) that are syntactically very similar may induce very different semantics and a list of differences should be best addressed *semantically*. Maoz *et al.* define a semantic diff as an operator that takes as input two models and outputs a set of diff witnesses, i.e., instances of one model that are not instances of the other [19]. In our context, instances are configurations and the set of witnesses is finite and can be enumerated if needs be. Fahrenberg *et al.* propose an alternative definition of a semantic diff and argue that a difference between models should be a model [13]. One contribution of our work is precisely to compute a diff FM.

Recently, Maoz *et al.* tackled the problem of semantic model differencing, specifically for class and activity diagrams [17,18]. They defined and implemented two versions of semantic diff operator, *cddiff* and *addiff*. The *cddiff* operator [18] takes as inputs two class diagrams and computes diff witnesses using Alloy Analyzer, a solver for first-order logic. For the *addiff* operator, they presented algorithms that take as input activity diagrams [17]. These two contributions, as ours, are specific to a given formalism and its associated semantics. A few works consider semantic diff between programs, e.g., Jackson and Ladd summarize the semantic diff between two procedures in terms of observable input-output behaviors [14]. We focus on model comparison and not on program comparison.

In the field of feature modeling, Benavides et al. [7] survey a set of operations and techniques proposed for automated analysis of FMs. No automated techniques have been reported to reason about or compute differences. A notable exception is the algorithm described in [28], which classifies the evolution of an FM via modifications (see Section 3.2). The algorithm can be used in the context of FM differences but have two limitations. First, the kind of relationship between two FMs does not help to precisely *understand* the impact of a change, e.g., what implies or excludes constraints have been removed and added. Second, the technique does not compute *all* added and removed configurations. In [13], the authors illustrate their vision and theory of semantic model differences using FMs. They propose an algorithm to compute the *quotient* (see Section 3.2). As recognized in [13], the quotient is an approximation of the differences between two FMs whereas the diff FM is not. Another limitation is that the quotient is a set of disjunctive clauses that are difficult to understand for a practitioner (see Section 4.2). In practice, an additional step seems necessary to transform

these clauses into a more readable and manageable information, closer to FM constructs. Segura et al. propose a catalog of rules for merging FMs (union and intersection) [26]. They present syntactic mechanisms (see a comparison in [3]) and no diff operator is considered. The works exposed in [23,11] developed a set of operators to make evolve FMs but no differencing technique is proposed to control the evolution of the FMs.

6 Conclusion

Feature models (FMs) are widely used to compactly represent the valid combinations of features (i.e., configurations) supported by a given system. In several application domains and contexts (e.g., software evolution), differences between two FMs should be managed, for example, to identify what are the configurations of an FM that are not included in another. We presented a set of techniques to understand, compute and reason about such differences. The techniques perform at the semantic level (i.e., in terms of sets of configurations) and present relevant information reified from the analysis of propositional formula. A practitioner can detect a difference (e.g., for the purpose of debugging), compute the differences as an FM and then integrate the differences into an existing FM. The tool-supported techniques overcome limitations of earlier attempts and are proved to be essential in a case study.

As future work, we plan to evaluate further the practicality and usefulness of the proposed solution. We hope these insights can contribute to a methodology that guide practitioners in managing FM differences.

Acknowledgement. This work was supported by the FNRS, the University of Namur, the FP7 Marie-Curie COFUND program, the Interuniversity Attraction Poles Programme, Belgian Science Policy (MoVES), the BNP, the french ANR SALTY project (`https://salty.unice.fr`) under contract ANR-09-SEGI-012, and the French Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council and FEDER through the CPER-CIA 2007-2013.

References

1. `https://nyx.unice.fr/projects/familiar/wiki/DiffFMs`.
2. M. Acher, A. Cleve, P. Collet, P. Merle, L. Duchien, and P. Lahire. Reverse Engineering Architectural Feature Models. In *Proc. of ECSA'11*, volume 6903 of *LNCS*, pages 220–235. Springer, 2011.
3. M. Acher, P. Collet, P. Lahire, and R. France. Comparing Approaches to Implement Feature Model Composition. In *Proc. of ECMFA'10*, volume 6138 of *LNCS*, pages 3–19, 2010.
4. M. Acher, P. Collet, P. Lahire, and R. France. A Domain-Specific Language for Managing Feature Models. In *Proc. of SAC'11*, pages 1333–1340. ACM, 2011.
5. M. Acher, P. Collet, P. Lahire, and R. France. Slicing Feature Models. In *Proc. of ASE'11*, pages 424–427. ACM, 2011.
6. M. Acher, P. Collet, P. Lahire, A. Gaignard, R. France, and J. Montagnat. Composing Multiple Variability Artifacts to Assemble Coherent Workflows. *Software Quality Journal (Special issue on Quality Engineering for SPLs)*, 2011.

7. David Benavides, Sergio Segura, and Antonio Ruiz Cortés. Automated analysis of feature models 20 years later: A literature review. *Inf. Syst.*, 35(6):615–636, 2010.
8. P. Bille. A survey on tree edit distance and related problems. *Theor. Comput. Sci.*, 337(1-3):217–239, 2005.
9. K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
10. K. Czarnecki and A. Wąsowski. Feature diagrams and logics: There and back again. In *Proc. of SPLC'07*, pages 23–34, 2007.
11. D. Dhungana, P. Grünbacher, R. Rabiser, and T. Neumayer. Structuring the modeling space and supporting evolution in software product line engineering. *Journal of Systems and Software*, 83(7):1108–1122, 2010.
12. J. Euzenat and P. Shvaiko. *Ontology matching*. Springer-Verlag, Heidelberg, 2007.
13. U. Fahrenberg, A. Legay, and A. Wasowski. Vision paper: Make a difference! (semantically). In *Proc. of MoDELS'11*, pages 490–500, 2011.
14. D. Jackson and D. A. Ladd. Semantic diff: A tool for summarizing the effects of modifications. In *Proc. of ICSM '94*, pages 243–252, 1994.
15. D.S. Kolovos, D. Di Ruscio, A. Pierantonio, and R.F. Paige. Different models for model matching: An analysis of approaches to support model differencing. In *Proc. of CVSM'09 (ICSE Workshop)*, pages 1 –6, may 2009.
16. R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wasowski. Evolution of the linux kernel variability model. In *Proc. of SPLC'10*, volume 6287 of *LNCS*, pages 136–150, 2010.
17. S. Maoz, J. O. Ringert, and B. Rumpe. Addiff: semantic differencing for activity diagrams. In *Proc. of ESEC/FSE'11*, pages 179–189. ACM, 2011.
18. S. Maoz, J. O. Ringert, and B. Rumpe. Cddiff: semantic differencing for class diagrams. In *Proc. of ECOOP'11*, pages 230–254. Springer, 2011.
19. S. Maoz, J. O. Ringert, and B. Rumpe. A manifesto for semantic model differencing. In *Proc. of MODELS'10*, pages 194–203. Springer, 2011.
20. M. Mendonca, A. Wąsowski, K. Czarnecki, and D. Cowan. Efficient compilation techniques for large scale feature models. In *Proc. of GPCE'08*, pages 13–22. ACM, 2008.
21. A. Metzger, K. Pohl, P. Heymans, P-Y. Schobbens, and G. Saval. Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. In *Proc. of RE'07*, pages 243–253, 2007.
22. Bibliography on Comparison and Versioning of Software Models. `http://pi.informatik.uni-siegen.de/CVSM`.
23. A. Pleuss, G. Botterweck, D. Dhungana, A. Polzer, and S. Kowalewski. Model-driven support for product line evolution on feature level. *Journal of Systems and Software*, 2011.
24. K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, 2005.
25. P-Y. Schobbens, P. Heymans, J-C. Trigaux, and Y. Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 51(2):456–479, 2007.
26. S. Segura, D. Benavides, A. Ruiz-Cortes, and P. Trinidad. Automated merging of feature models using graph transformations. *Post-proceedings of the Second Summer School on GTTSE*, 5235:489–505, 2008.
27. S. She, R. Lotufo, T. Berger, A. Wąsowski, and K. Czarnecki. Reverse engineering feature models. In *Proc. of ICSE'11*, pages 461–470. ACM, 2011.
28. T. Thüm, D. Batory, and C. Kästner. Reasoning about edits to feature models. In *Proc. of ICSE'09*, pages 254–264. ACM/IEEE, 2009.