Support for Reverse Engineering and Maintaining Feature Models

Mathieu Acher, Patrick Heymans, Anthony Cleve, Jean-Luc Hainaut, Benoit Baudry

Support for Reverse Engineering and Maintaining Feature Models

Mathieu Acher, Benoit Baudry
University of Rennes 1
Irisa / INRIA, France
{macher,bbaudry}@irisa.fr

Patrick Heymans, Anthony Cleve, Jean-Luc Hainaut
PReCISE Research Centre
University of Namur, Belgium
{phe,acl,jlh}@fundp.ac.be

ABSTRACT

Feature Models (FMs) are a popular formalism for modelling and reasoning about commonality and variability of a system. In essence, FMs aim to define a set of valid combinations of features, also called configurations. In this paper, we tackle the problem of synthesising an FM from a set of configurations. The main challenge is that numerous candidate FMs can be extracted from the same input configurations, yet only a few of them are meaningful and maintainable. We first characterise the different meanings of FMs and identify the key properties allowing to discriminate between them. We then develop a generic synthesis procedure capable of restituting the intended meanings of FMs based on inferred or user-specified knowledge. Using tool support, we show how the integration of knowledge into FM synthesis can be realized in different practical application scenarios that involve reverse engineering and maintaining FMs.

1. INTRODUCTION

In an increasingly competitive market, designing, developing and maintaining software for one customer, one hardware device, one operating system or one country is no longer a viable alternative. Numerous organisations, in many domains, rather efficiently produce a large variety of similar software products, e.g., following a *software product line* (SPL) approach [1]. In this context, *variability*, defined as *"the ability of a software system or artifact to be efficiently extended, changed, customized or configured for use in a particular context"*, should be properly managed [2].

Feature models (FMs) are one of the most popular formalism for modelling and reasoning about variability of a system [3, 4, 5, 6, 7]. Numerous approaches rely on FMs for various purposes (see Figure 1, right) such as generation of product configurators, customisation of other artifacts (e.g., code, models) to derive members of an SPL, or automated reasoning about properties of an SPL through testing or model checking techniques [1, 8, 9, 10, 11].

From Configurations to FMs. The primary purpose of FMs is to represent a set of valid *configurations*, i.e., a set of sets of features. For this purpose, syntactical mechanisms (feature hierarchy, notion of optional and mandatory features, feature groups, and cross-tree constraints) are provided to define what the allowed combinations of features are. Importantly, the valid configurations characterised by an FM should not be too large (otherwise some unsafe composition of respective artifacts are authorised) or too narrow (otherwise it is a symptom of unused flexibility of the system) [12]. Numerous approaches assume that the configuration space is accurately represented by an FM, authorizing an automated and safe composition of associated artefacts. A badly designed FM may have severe consequences on the forward engineering process (see Figure 1). For example, the configurator generated from the FM may expose to customers configurations that actually correspond to any existing product ; the build system can assemble software code that is not consistent with the project's capabilities and violates the property of safe composition, etc. As a result, the construction of an FM representing a given configurations set is of prior importance in many scenarios[1].

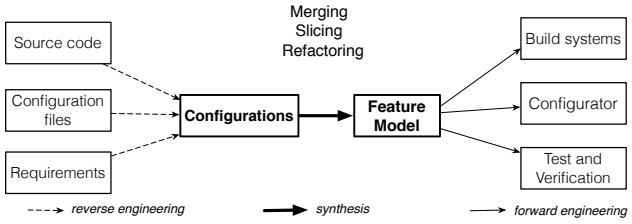


Figure 1: From configurations to a feature model

Nevertheless a manual construction of an FM is both time-consuming and error-prone. Even for a small set of configurations (see Figure 2a), elaborating an accurate FM – specifying its hierarchy, its variability information and its constraints – is known to be impractical manually [4, 7, 13]. Other works propose the use of automated procedures to extract logical dependencies and features' sets from existing software artefacts – being source code, configuration files or requirements [6, 14, 15, 16, 17, 18, 19] (see Figure 1, left). This is a first step but there is still a need to construct an actual and comprehensive FM (see Figure 1, center).

The problem is that numerous FMs can represent a given set of configurations, out of which numerous candidates are

[1]She et al. [13] discussed other scenarios and works in which the configuration space is over or under approximated by an FM. This is out of the scope of this paper.

VaMoS '13, January 23-25, 2013, Pisa, Italy

not maintainable. Obtaining FMs with an unmeaningful feature hierarchy or set of cross-tree constraints may increase the cognitive effort of users involved in the engineering of variability-intensive systems.

Contributions and Outline. In this paper, we present a practical support for *synthesising* an FM from a set of configurations (typically encoded as a Boolean formula). Using our tooling support, a user can breath knowledge into the FM synthesis procedure in order to obtain an FM that is both *accurate* (w.r.t. the set of configurations), *meaningful* (i.e., feature hierarchy, feature groups and cross-tree constraints are maintainable since conformant to user's intention) and *unique*. As we will show, the tool-supported procedure can be applied in many application scenarios such as reverse engineering, refactoring, merging or slicing FMs.

The remainder of the paper is organized as follows. In Section 2, we clarify the different meanings of an FM – beyond its configuration semantics – and identify what *knowledge* is needed by a synthesis procedure to produce a unique and accurate FM. In Section 3, we describe a sound FM synthesis procedure that takes as inputs a Boolean formula and a user-specified knowledge describing the intended properties of the FM. The procedure controls that the specification is consistent (e.g., the specified feature hierarchy is possible) and complete (i.e., no additional knowledge is neededed to produce an unique FM). In Section 4, we integrate our results in a dedicated language and environment, giving a concrete syntax to the specification and allowing users to reuse other FM management facilities. In Section 5, we revisit existing works and show that our tool support reaches better results in terms of user effort and output FM quality. We also report on our experience and identify opportunities for future work.

2. FROM SYNTAX TO SEMANTICS

2.1 Syntax

FMs hierarchically organise a potentially large number of concepts (features) into multiple levels of increasing detail, typically using a tree. Depending on the level of abstraction and artefact described, features in FMs are referred to as high-level decisions taken by stakeholders or functionalities of a software system [20].

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 2b describes a medical image that has two *mandatory* features, Modality and Format. There are two alternatives for Modality acquisition: MRI and CT features form an *Xor*-group (i.e., at least and at most one feature must be selected). Similarly, Nifti and DICOM features form an *Xor*-group of Format. An *optional* feature of Format is Anonymized. Two cross-tree constraints, involving features DICOM, MRI and Anonymized, are also specified in order to restrict their valid combinations.

The terms FM and *feature diagram* are employed in the literature, usually to denote the same thing. 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 (bi-implies, implies, excludes).

Definition 1 (Feature Diagram) *A feature diagram* $FD = \langle G, E_{MAND}, G_{MUTEX}, G_{XOR}, G_{OR}, EQ, RE, EX \rangle$ *is defined as follows:* $G = (\mathcal{F}, E, r)$ *is a rooted 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 ;* $G_{MUTEX} \subseteq \mathcal{P}(\mathcal{F}) \times \mathcal{F}$, $G_{XOR} \subseteq \mathcal{P}(\mathcal{F}) \times \mathcal{F}$ *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 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* $A \Rightarrow \neg B$ *(*$A \in \mathcal{F}$ *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. A similar abstract syntax is used in [3, 6] while other existing dialects slightly differ [21].

2.2 Meanings

The essence of an FM is its *configuration semantics* (see Definition 3). The syntactical mechanisms 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. As shown in [3], feature diagrams are not expressively complete regarding propositional logics and thus cannot represent any set of configurations. It explains why we consider that an FM is composed of a feature diagram plus a propositional formula (see Definition 2). The set of configurations represented by an FM can be logically translated to a propositional formula ϕ defined over a set of Boolean variables, where each variable corresponds to a feature [3]. Formally, $\phi = \phi^{FD} \wedge \psi$ where ϕ^{FD} is the propositional formula of the feature diagram. For example, the propositional formula of fm_1 (see Figure 2c) is $\phi_{fm_1} = \phi_{fm_1}^{FD_1} \wedge \psi_1$ where $\psi_1 = CT \vee Anonymized$.

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

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

2.3 Other Semantics

It is well-known that models that are syntactically very similar may actually have very different semantics (in the sense of meanings), and vice versa, models that describe the same concept may have very different syntactic representations [22]. This observation naturally applies to FMs. We now further investigate the different *meanings* of FMs.

Feature hierarchies. Let us consider the examples of Figure 2. The three FMs, all supposed to describe the characteristics of a medical image, have different feature hierarchies. fm_0 is the ideal model and the only one to correctly organize features in the hierarchy (i.e., DICOM and Nifti are indeed specific medical imaging Formats, CT and MRI are indeed specific medical imaging Modality Acquisitions). From a conceptual perspective, the two other FMs fm_1 and fm_2 are not correct. fm_1 is clearly erroneous: features DICOM and Nifti actually correspond to a medical

$s_0 = \{$

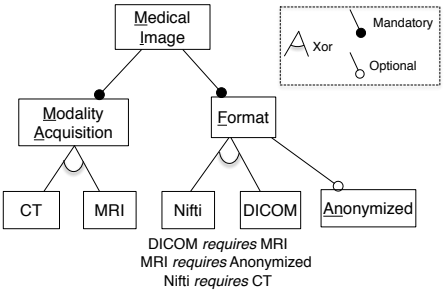
$\{MI, MA, F, CT, Nifti\},$

$\{MI, MA, F, CT, Nifti, AN\},$

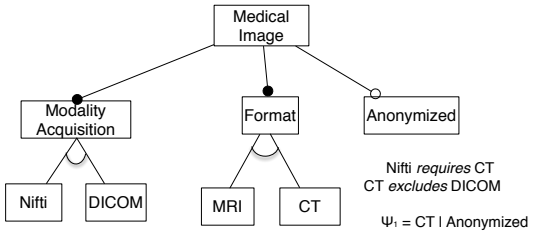
$\{MI, MA, F, DICOM, MRI, AN\}$

$\}$

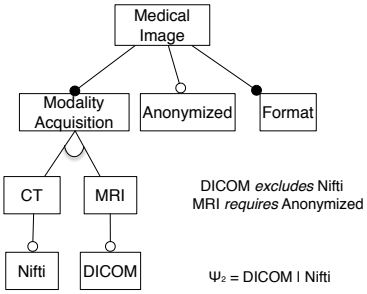
(a) set of configurations



(b) fm_0



(c) fm_1



(d) fm_2

Figure 2: For a given set of configurations, three possible yet different FMs ($s_0 = [\![fm_0]\!] = [\![fm_1]\!] = [\![fm_2]\!]$)

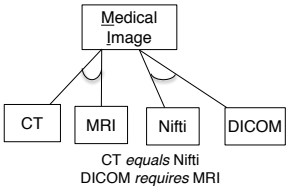
imaging format and not to a modality acquisition. Similarly, features MRI and CT are not medical imaging formats as modelled in fm_1: these are modality acquisitions. In fm_2, the intention of the modeller is to state that a CT (resp. MRI) modality acquisition always come with Nifti (resp. DICOM). The hierarchy contributes here to the configuration semantics (parent-child relationships encodes logical implications between the features). Traditionally, the hierarchy in an FM is also used to describe a notion of refinement between two concepts. As a result, from a conceptual perspective and w.r.t. the medical imaging domain, the relationship between CT (resp. MRI) and Nifti (resp. DICOM) is highly questionable in fm_2.

Feature groups. Let us now consider the example of Figure 3. Three FMs represent the same set of configurations s_1, corresponding to the projection of s_0 (see Figure 2a) onto $\{MI, DICOM, Nifti, CT, MRI\}$:

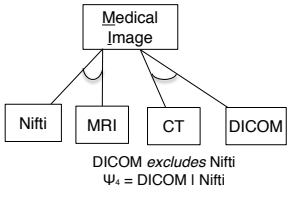
$$s_1 = \{\{MI, CT, Nifti\}, \{MI, DICOM, MRI\}\}$$

The hierarchy is the same in the three FMs. But we can observe that features Nifti and MRI are grouped together in fm_4 – it is not the case in fm_3 and fm_5. From a conceptual perspective, the way features are grouped in fm_4 is questionable: Nifti is a medical image format whereas MRI is a modality acquisition.

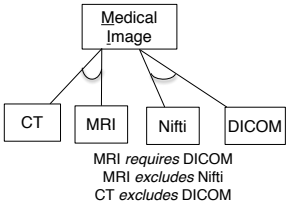
Constraints. Let us consider fm_3 and fm_5 of Figure 3. The two FMs have the same configuration semantics, hierarchy and feature groups. There are still differences since the constraints are not the same: one *equals* and one *requires* constraint in fm_3 versus one *requires* and two *excludes* constraints in fm_5. We can first notice that, in fm_5, the three constraints are logically redundant, i.e., one can remove one of the three constraints without altering the configuration semantics. Second, fm_3 keeps some original constraints expressed in fm_0 (see Figure 2b) whereas fm_5 contains an *excludes* constraint originally expressed in fm_1 (see Figure 2c, page). It is hard to determine whether a solution is preferable to another. An FM user might have preferences, considering that a given constraint is easier to understand or that his/her original specification must be restituted.



(a) fm_3



(b) fm_4



(c) fm_5

Figure 3: Feature groups and constraints matter: for a given set of configurations, three possible yet different FMs ($s_1 = [\![fm_3]\!] = [\![fm_4]\!] = [\![fm_5]\!]$)

3. PARAMATERIZING THE SYNTHESIS

Using different examples, we have shown that some FMs are not adequate. It may induce severe *maintenance* (e.g., understandability, changeability) overheads for FM users. It may also have an impact on the *automation* in a for-

ward engineering phase: some implementation approaches exploit the hierarchy of FM (e.g., see [20]) ; the way features are hierarchically organised and grouped can be considered when generating a configuration user interface from an FM, etc. Therefore it is crucial that the different meanings of FMs previously identified (hierarchies, feature groups, constraints) are properly restituted by the synthesis procedure.

These meanings have impacts on the design of the synthesis procedure. We want that a set of configurations leads to an *unique* FM. Furthermore, the procedure should be *deterministic* in order to avoid surprises and be reproducible, leading to the exact same resulting FM. Without additional *knowledge*, an automated procedure will take arbitrary decisions, possibly not conformant with the conceptual relationships across a domain or the intention of an FM user.

3.1 Synthesis Algorithm

Therefore we develop a *parameterised* synthesis procedure (see Figure 4). The principle is that from a propositional formula ϕ (logical encoding of a set of configurations) the different syntactical properties of a feature diagram are incrementally computed until obtaining a comprehensive FM. When needs be, the algorithm exploits a specification (called KSS, see Definition 4) describing the intended properties of the resulting FM. This specification is typically provided by an FM user but can also be inferred from another source of information (see Section 5). Importantly, a user does not necessarily have to describe all properties the resulting FM should exhibit (i.e., some of the properties can be automatically inferred by the synthesis procedure without additional knowledge). The different steps of the algorithm (see Figure 4) as well as the related satisfiability techniques are all based on prior works [3, 6, 7] that we extensively rely on. We briefly recall them in the remainder of this section. We also show how users can specify and breath knowledge into the synthesis procedure.

Preprocessing steps. The algorithm starts by removing variables in ϕ that correspond to dead features, i.e., features that are not present in any configuration. In line 2, the binary implication graph (noted BIG) of the formula ϕ over $\mathcal{F}$ (the set of non dead features) is computed. BIG is a directed graph constituting of vertices being features and edges denoting an implication between two features. In line 3, we use BIG to identify *atomic sets* (i.e., set of features that are mutually implied and always appear together in configurations) denoted AS. These are identified as the strongly connected components in BIG [3, 6]. In line 4, we compute the binary exclusion graph, noted BEG of the formula ϕ over $\mathcal{F}$. It corresponds to an undirected graph constituting of vertices being features and edges denoting a mutual exclusion between two features [6].

Definition 4 (Knowledge synthesis specification (KSS)) *A KSS is denoted* $K = \langle K_{FH}, K_r, K_{MAND}, K_{MUTEX}, K_{XOR}, K_{OR}, K_{EQ}, K_{RE}, K_{EX}, K_\psi \rangle$ *and is defined over a set of features* $\mathcal{F}$*:* $K_{FH} \subseteq \mathcal{F} \times \mathcal{F}$ *specifies (part of) a hierarchy ; other information corresponds to properties of FMs:* $K_r \in \mathcal{F}$ *is the root feature ;* $K_{MAND}, K_{MUTEX}, K_{XOR}$ *and* K_{OR} *correspond to intended variabilty information (mandatory features and feature groups) ;* K_{EQ}, K_{RE}, K_{EX} *and* K_ψ *specify sets of constraints to be included.*

Setting feature hierarchy and mandatory features. In general, BIG is not a rooted tree and cannot be used to define the hierarchy of the feature diagram. A preliminary step is to determine the root feature. It can be automatically inferred iff the number of *core* features (i.e., features that are present in all configurations) is equal to 1. Otherwise, it should be explicitly specified. A second step is to determine parent-child relationships between features. AS is a source of ambiguity since, for example, both members of the atomic set can be parent of the others. Moreover, a feature can be a child feature of many other features (following the large number of implication edges in BIG). For all these cases, the procedure has to rely on some knowledge about the hierarchy (e.g., K_{FH}, see Definition 4).

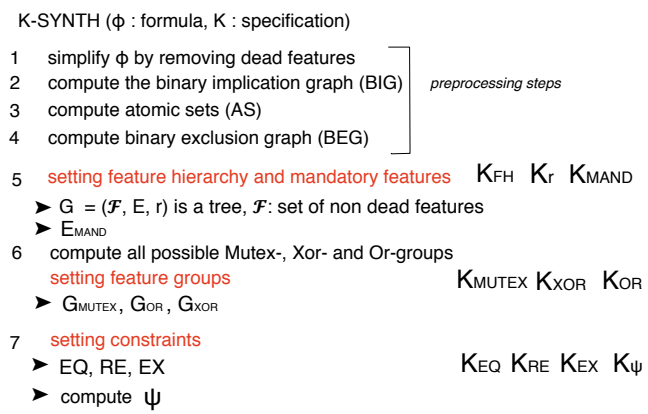


Figure 4: Sketch of the synthesis algorithm. Red part corresponds to decisions that need to be taken. In the left part, knowledge that may be used.

Setting feature groups. We reuse the prime implicants method proposed in [3] to identify Or-groups. We exploit BEG to compute Xor- and Mutex-groups [6]. An important issue is that a feature may be candidate to several feature groups. It is not allowed by FMs. An example is given in Figure 3: the feature MRI can be part of a Xor-group involving CT and is also candidate to a Xor-group involving Nifti. Therefore some feature groups should be dismissed to ensure well-formedness of FMs. For this purpose, the algorithm can exploit K_{MUTEX}, K_{XOR} or K_{OR}.

Setting constraints. The set of constraints K_{EQ}, K_{RE}, K_{EX}, and K_ψ specified by the FM user is first considered. Then we add the set of *equals* and *requires* constraints not already expressed by the feature diagram (e.g., parent-child relations) and deduced by removing some edges of BIG. Similarly, *excludes* constraints of BEG that were not chosen to be represented as a Mutex- or Xor-group are added. When adding constraints, we control that the constraint is not already induced by the FM for the purpose of decreasing the amount of *redundant* constraints (as in Figure 3c). Different strategies can be applied when synthesising constraints. In particular, an FM user can specify the order in which kinds of constraints are added (the default behavior is: first equals constraints, second implies constraints, third excludes constraints). For minimal representations, we compute the transitive reduction of BIG. At the end, the feature diagram may still be an over approximation of ϕ. Using propositional logics techniques, we can compute the complement ψ.

3.2 Breathing Knowledge

The general principle is that an FM user imposes some

constraints (e.g., on the hierarchy) and lets the synthesis algorithm constraining further the FM until obtaining the exact set of configurations. The procedure guarantees, by construction, that the exact set of configurations is represented.

Example 1. For synthesising fm_0 (see Figure 2b) it is sufficient to only specify the intended hierarchy $K_{FH} = E_0$ and $K_{RE} = RE_0$ (the root K_r is deduced from K_{FH}). Using this information, the algorithm is able to comprehensively synthesise the FM, i.e., all mandatory/optional features, Xor-groups and to determine that ψ_0 is not necessary.

It should be noted that a KSS (see Definition 4) can be a *partial* user specification of an FM (e.g., not all edges of the hierarchy or feature groups have to be specified). Nevertheless, it may happen that the specification is *inconsistent* or *incomplete*.

KSS consistency.

The constraints imposed by a KSS must not preclude the synthesis of an FM. Nevertheless it may happen, for example, when the feature hierarchy is not a spanning tree of the binary implication graph.

Example 2. Let us consider the set of configurations s_1 (see page). There exists no FM representing s_1 and in which DICOM is a child feature of CT. Intuitively, the constraint $DICOM \Rightarrow CT$ is inconsistent regarding s_1

In the general case, we say that a KSS is consistent w.r.t. a set of configurations s iff its constraints do not violate s. This truly holds for **Example 1.** but not for **Example 2.** The consistency of a KSS can be determined by checking the satisfiability of $\phi_{cmp} = \neg(\phi_{KSS} \Rightarrow \phi_s)$ where ϕ_{KSS} is the propositional formula of a KSS (the translation of a KSS to a formula is very similar to the encoding of FM to a formula) and ϕ_s the propositional formula of s. We rely on the algorithm exposed in [4] to analyse ϕ_{cmp} with satisfiability solvers. In case a KSS is not consistent, the differencing techniques exposed in [23] can be used to debug the specification.

KSS completeness.

A KSS is complete if the execution of the synthesis algorithm leads to an unique FM. For instance, the specification exposed in Example 1 is complete.

Example 3. Let us consider the set of configurations s_1. The specification only comprising $K_{FH} = E_3$ (the hierarchy of fm_3 in Figure 3a) is not complete. As previously discussed, different feature groups can indeed be synthesised.

When the KSS is not complete, two strategies can be considered. Firstly, KSS can be fixed during an *interactive* session where a series of questions are asked to users as soon as the algorithm has to take a decision. This knowledge can be serialised into a KSS and reused later on. Secondly, *arbitrary* decisions can be made by the synthesis procedure (e.g., for resolving conflicting feature groups). We implement and authorise both strategies in our tool support (see next section).

4. TOOL SUPPORT

FM users need a practical solution for using the synthesis technique previously described (e.g., a concrete syntax conforms to the KSS abstract syntax of Definition 4). We rely on FAMILIAR a language that already includes facilities for composing/decomposing FMs [24], editing FMs, reasoning about FMs (e.g., validity, comparison of FMs [4, 5]) and computing the differences [23]. FMs and other types (configuration, set, etc.) are manipulated using variables. Two reasoning back-ends – satisfiability solvers using SAT4J and binary decision diagrams (BDDs) using JavaBDD – are internally used when required (statisfiability techniques used are discussed in Section 5.5).

As compared to our previous effort, we extend the language and integrate the synthesis technique developed in the paper (a comprehensive tutorial is available in [25]). In particular, we now authorise to import formula – in the same way we authorise to import FMs in various formats (e.g., TVL [26] or FeatureIDE [20]) . For example, in line 1, we import a formula in DIMACS format (a standard format for boolean formulas in conjunctive normal forms) and store it in the variable $fla1$:

```
1  fla1 = FM ("fooFla1.dimacs")
2  n1 = counting fla1
3  fm1 = FM ("foo1.tvl")
4  r1 = root fm1
```

Then, numerous reasoning operations can be performed, e.g., in line 2 we count the number of valid solutions of the formula. Actually $fla1$ can be manipulated just as an FM, except for some operations that perform over the syntactical information of an FM. For example, in line 4, the accessor root returns the root feature of the FM $fm1$. This kind of operation cannot be applied to $fla1$ because the formula has not been synthesised into a complete FM.

It is where the synthesis procedure comes to the rescue. We provide an operator, called **ksynthesis**, that performs over formulae and that computes an FM. Two examples are given below:

```
5  fm2 = ksynthesis fla1 with hierarchy=
6  A : B G ; B : C E O; groups= xorgroup
7  (B: C E) constraints= E implies G
8
9  // another synthesis
10 fm1bis = ksynthesis fm1 with hierarchy=
11 A : B C G O ; G : E ;
```

The operator takes as first parameter either a formula (see line 5–7, $fla1$) or an existing FM (see line 10–11, $fm1$). In case the parameter is an FM, the corresponding formula of the FM is considered by the synthesis operator.

The other parameters correspond to a KSS specification. For example, in line 5–7, the hierarchy (A is the root, B and G are its children, C and E are children of G), the way features should be grouped (C and E form a Xor-group with B as parent feature) and the constraints expected to be in the resulting FM are specified. In line 10–11, another synthesis is called. This time, only the hierarchy is specified.

The **ksynthesis** operator can be used in interactive mode (see line 12–14 below).

```
12 fm2bis = ksynthesis --interactive fla1
13 with hierarchy= A : B G ; B : C E O;
14 constraints= E implies G
```

Let say the KSS is not *complete* (see Section 3) since the user forgets to specify the way features are grouped (it was not the case in line 5–7, see above). In this case, the procedure can ask to the user, e.g., which feature groups have to be restitued in the resulting FM.

5. APPLICATIONS

We now explain how the generic synthesis procedure can be applied in different application scenarios, showing the *applicability* of our contributions. We systematically compare our proposal to existing works. We show that we improve the *quality* of the resulting models and reduce the *user effort* needed when reverse engineering or refactoring FMs.

5.1 Reverse Engineering FMs

Numerous approaches propose to reverse engineer FMs. Specifically, we consider the approaches exposed in Haslinger et al. [14] and in our previous works [16]. The authors propose to extract FMs from "feature sets", typically documented in a table [14, 16]. It is an instance of the FM synthesis problem, where the feature sets correspond to the set of configurations and where column labels are feature names (see Figure 5a). A major difficulty is that the table does not contain structural or conceptual information, thus *many* FMs can be constructed.

Let us consider a basic strategy that uses a flattened hierarchy (a tree with a depth of 1): we assume that feature VOD (V for short) is the root and all other features are located below. Using our synthesis procedure, we can detect conflicting feature groups. For instance, the feature M is candidate to several feature groups: either as part of a Mutex group with Ae, an Xor group with T, an Or-group with Ae and C or a Mutex group with C. Similarly, the features S, M and C can be part of numerous feature groups. These conflicts can be detected by the procedure and then corrected accordingly. Two examples are given in Figure 5c and Figure 5d, resulting from two different feature groups fixed by the user. We can observe that the two FMs are not maintainable. The conceptual relationships between features are highly questionnable: TV (T for short) is grouped together with $Smart$ (S for short) ; $Aerial$ (Ae for short), $Cable$ (C for short) and $Mobile$ (M for short) are forming an Or-group. Moreover the presence of constraints does not ease the readability. Intuitively, the constraint C *requires* T is due to a lack of structure between features.

Fortunately, an expert is likely to *know* the intended hierarchy of the FM (see Figure 5b). Therefore our procedure can be directly applied. We encode the feature sets as a propositional formula and then impose a feature hierarchy using a KSS. When applying on the example, we observed that the specification is *complete*. We synthesised a unique and comprehensive FM without any additional knowledge and user effort. The resulting FM exactly corresponds to the one depicted in Figure 5b.

Comparison with Existing Approaches. The algorithm and the heuristics proposed in [14] fail to reconstruct the complete hierarchy of Figure 5b. The main reason is that there is no control on the way the algorithm synthesises the FM. Furthermore our solution allows one to build an FM with minimal effort. In [16], we proposed a tool-supported approach for extracting FMs from product descriptions (e.g., documented in tabular data). We raise the following limitations: *i)* the hierarchy of the resulting FM can be fully parameterized, avoiding hardly readable FMs (e.g., the ones of Figure 5c and Figure 5d) *ii)* interactive support for conflicting feature groups and *iii)* control on the way constraints should be restituted. In [6], She et al. propose a method to reverse engineer large scale FMs in the operating system domains (i.e., Linux, eCos, FreeBSD). They notably propose a heuristic to determine the feature hierarchy of the FM. A KSS can be used to impose the inferred hierarchy, to resolve conflicting feature groups and to restitute the specific constraints originally specified by the developer. Furthermore, the procedure can be re-invoked by reusing the KSS, for example, for another version of the Linux FM.

5.2 Refactoring FMs

We now tackle the problem of *refactoring* an FM, i.e., producing a revised version of an FM that has the same configuration semantics. Formally, let fm_{rf_1} be an FM. The refactoring process is a set of edits to fm_{rf_1} that produces a new FM, say fm_{rf_2}, such that $[\![fm_{rf_1}]\!] = [\![fm_{rf_2}]\!]$.

Refactoring an FM is needed in the following situations [27, 9, 4, 28]: *i)* Variability-intensive systems and their FMs do evolve and have to be maintained ; *ii)* FMs, coming from (more than) two different sources or stakeholders may have to be aligned and refactored accordingly ; *iii)* The FM resulting from an automated procedure (let say, a synthesis procedure) may have to be restructured if not satisfactory.

Due to its combinatorial nature, refactoring an FM is known to be impractical to perform manually [29, 4]. Even small edits to an FM, like moving a feature from one branch to another, can unintentionally change the set of valid feature combinations. We propose to use our synthesis procedure to refactor FMs. The principle is that we encode the set of configurations of fm_{rf_1} and we then specify the different properties fm_{rf_2} should have. In particular, we can impose a hierarchy or change the way features are grouped.

The advantages are as follows. Firstly, an FM user does not have to find all editing steps needed like "move feature DICOM below feature Format, then swap the position of features MRI and CT, ..., and finally remove the constraint MRI requires DICOM". She can simply specify the intended meanings, while the synthesis procedure automatically populates the FM with accurate variability information. Secondly previous meanings (feature groups, constraints) can be reused in the KSS so that the user can focus on inadequate parts (e.g., the way constraints are restituted). Thridly some edits are not allowed by construction and can be detected early by the synthesis procedure. For example, in Figure 3a, specifying that the feature MRI should be located below CT corresponds to an inconsistent KSS (see page).

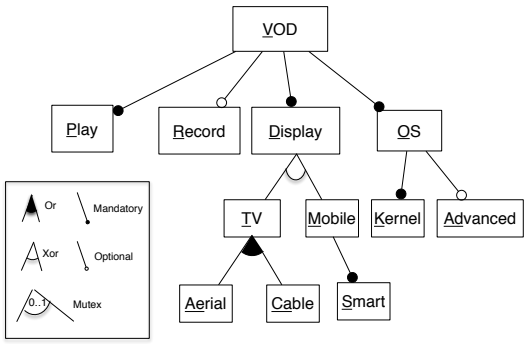
Comparison with Existing Approaches. A catalog of refactoring rules is proposed in [27]. The main disadvantage of using predefined operations is that all edits that convert one model into another one must be known. Furthermore the FM user can unintentionally change the set of valid feature combinations. As argued, a KSS allows an FM user to specify intended properties while the synthesis procedure does not alter the configuration semantics. Therefore our solution for refactoring FMs is more general, it guarantees the configuration semantics and it reduces the user effort. The algorithm described in [4] classifies the evolution of an FM via modifications (edits). The algorithm can be used to precisely understand the impact of a change, for example, if the set of configurations remains the same. When refactoring FMs, the problem is different: we want to modify FMs while ensuring the non alteration of configuration semantics.

5.3 Re-engineering FMs of SPLOT

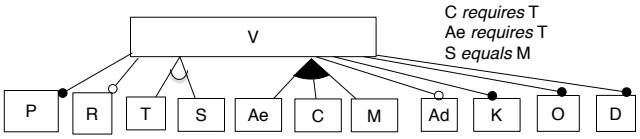
We conducted another experiment based on 201 FMs (see [25] for more details) of the SPLOT repository (`http://www.`

P	V	P	R	D	O	T	M	S	K	Ad	Ae	C
P1	✓	✓	✓	✓	✓	✓			✓	✓	✓	
P2	✓	✓	✓	✓	✓	✓			✓		✓	
P3	✓	✓		✓	✓	✓			✓	✓	✓	
P4	✓	✓		✓	✓	✓			✓		✓	
P5	✓	✓	✓	✓	✓	✓			✓	✓	✓	✓
P6	✓	✓	✓	✓	✓	✓			✓	✓		✓
P7	✓	✓	✓	✓	✓	✓			✓		✓	✓
P8	✓	✓	✓	✓	✓	✓			✓			✓
P9	✓	✓		✓	✓	✓			✓	✓	✓	✓
P10	✓	✓		✓	✓	✓			✓	✓		✓
P11	✓	✓		✓	✓	✓			✓		✓	✓
P12	✓	✓		✓	✓	✓			✓			✓
P13	✓	✓	✓	✓	✓		✓	✓	✓	✓		
P14	✓	✓	✓	✓	✓		✓	✓	✓			
P15	✓	✓		✓	✓		✓	✓	✓	✓		
P16	✓	✓		✓	✓		✓	✓	✓			

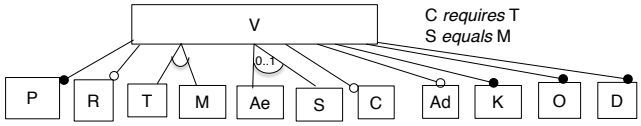
(a) Input tabular data (feature sets), extracted from [14]



(b) Expected [14] and actual synthesised FM



(c) Possible FM (1)



(d) Possible FM (2)

Figure 5: Features sets and corresponding FMs

`splot-research.org/`). It consists in synthesising FMs that represent the same set of configurations of original FMs included in our sample. The goal of the experiment was *i)* to determine what user effort is needed and *ii)* to characterize the syntactical properties of resulting FMs compared to original FMs. We used our support as follows. For each input FM of SPLOT, we compute its propositional formula. We only specify the intended hierarchy K_{FH} as being the original hierarchy (other components of the KSS are left empty). We obtained the following results:

- 147 synthesised FMs (69 %) were exactly the same as input FMs;
- 40 synthesised FMs (19%) were *corrections* of input FMs. Indeed, we observed that some FMs of SPLOT either *i)* exhibit false optional features, false Or-groups, or redundant constraints *ii)* or forget to group features as Xor or Or-groups. Our procedure automatically synthesises the correct syntactical constructs, without additional knowledge;
- 24 synthesised FMs (12%) were different. The first reason is that another set of cross-tree constraints was synthesised. The second one is that we observed feature group conflicts in six cases. In both cases, additional knowledge is required to get the same FM.

The results show that the main user effort is related to the feature hierarchy while other properties do not have to be specified in the vast majority of cases (88% of the FMs) since the procedure automatically synthesises them.

5.4 Revisiting FM Management Operations

FM management operations such as merge, diff, and slicing have proved to be useful [24, 23]. The FM synthesis is at the core of such operations, since they are defined on sets of configurations derived from the input FMs [24, 7, 13]. For example, the challenge of implementing the merge operator (in union mode) is as follows: "given a set of FMs $fm_1, fm_2, \ldots, fm_n$, how to synthesise an FM fm_r that represents the union of input sets of configurations, i.e., such that $[\![fm_r]\!] = [\![fm_1]\!] \cup [\![fm_2]\!] \cup \ldots \cup [\![fm_n]\!]$?"

The merge[2] can be implemented by *generating* a KSS and reusing the generic synthesis procedure ($[\![fm_r]\!]$ is encoded as a propositional formula ϕ_r, as explained in [24]). The KSS comprises the specification of a feature hierarchy (a minimum spanning tree of the binary implication graph of ϕ_r that maximises the parent-child relationships of input FM hierarchies [24]). Feature groups and constraints are computed on-the-fly, using some heuristics to retain the original constructs of input FMs.

Using our new proposals, we not only have an elegant and more generic solution for implementing the different heuristics. We can also control the completeness of the specification and detect which information should be refined. In case it does not correspond to her intention, an FM user can *refactor* the resulting merged FM (as previously exposed). For example, if we consider the merge in union mode of fm_0, fm_1 and fm_2 (see Figure 2), the automated procedure can produce many different hierarchies (there are different minimum spanning trees) – possibly a hierarchy corresponding to none of the three FM hierarchies. Our experience revealed that this situation does happen in practice. For example, when extracting an FM from product descriptions [16] and applying the merge operator, many feature groups' conflicts occur while many sets of cross-tree constraints could be restitued in the resulting FM. The merge operator has to take rather arbitrary decisions. As a result, we encountered diffculties to find effective heuristics in the general case. The tooling support now allows the user to be part of the extraction process either by refactoring the resulting merged FM or by initially parameterizing the merging.

5.5 Discussions

Interactive Support. Janota et al. propose an inter-

[2]The implementation of the slicing operator [24] follows a similar principle.

active editing environment that allows the user to decide interactively among parent and group candidates [29]. An FM user constructs an FM *from scratch* from a propositional formula until obtaining a complete FM. This approach does not cover all the scenarios presented in the paper like the refactoring (or merging/slicing, see below) of *existing* FMs. Nevertheless the interactive editor can be certainly adapted and integrated into our support. Another direction for improvement is to integrate more sophisticated strategies to assist users in completing an incomplete KSS. Currently our solution is to present conflicts in a randomized way but specific heuristics (e.g., as proposed in [6]) may decrease the user effort. In particular, the results of Section 5.3 show that we should investigate further how users can be assisted in specifying or completing a (partial) feature hierarchy.

Feature Diagram vs FM. As recalled in Section 2, the synthesised feature diagram may not be sufficient to represent a given set of configurations. Other constraints than *equals*, *requires*, or *excludes* constraints – corresponding to ψ in Definition 2 – are sometimes needed. We encountered this situation in practice (when re-engineering FMs of SPLOT and in [16]) and others reported similar observations in the operating system domain [6]. Techniques to efficiently and meaningfully synthesise ψ are left as future work.

Performance. We rely on the same satisfiability techniques exposed in [3, 6, 7]. Recently, the computation of Or-groups has been made possible using SAT solvers [7]. It has not been yet integrated into our tool support though. Moreover the performance evaluation of the synthesis procedure in *specific* reverse engineering and maintenance settings is left as future work.

6. CONCLUSION

We addressed the problem of synthesising an FM given a set of configurations, typically encoded as a Boolean formula. We characterised the different properties a synthesised FM could have. We contributed to a generic and sound FM synthesis procedure capable of restituting the intended meanings of FMs. The procedure is supported by a dedicated language and users can apply the following principle: *"Give me a formula and some knowledge, I will synthesise an accurate, meaningful and unique FM".*

We demonstrated the applicability of the principle for reverse engineering, refactoring, merging or slicing FMs. When we revisited and reimplemented existing approaches with our tool-supported procedure, we observed better results in terms of user effort and quality of the output FM. Our practical support for synthesising FMs paves the way for investigating further the combination of multiple information sources (from ontologies to source code) with some expert knowledge, especially when reverse engineering and maintaining variability-intensive systems.

Acknowledgments. We thank the anonymous reviewers for their helpful suggestions and comments. This work was supported by ITEA2 project MERGE.

7. REFERENCES

[1] K. Pohl, G. Böckle, and F. J. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer-Verlag, 2005.

[2] M. Svahnberg, J. van Gurp, and J. Bosch, "A taxonomy of variability realization techniques: Research articles," *Softw. Pract. Exper.*, vol. 35, no. 8, pp. 705–754, 2005.

[3] K. Czarnecki and A. Wasowski, "Feature diagrams and logics: There and back again," in *SPLC'07*. IEEE, 2007, pp. 23–34.

[4] T. Thüm, D. Batory, and C. Kästner, "Reasoning about edits to feature models," in *ICSE'09*. ACM, 2009, pp. 254–264.

[5] D. Benavides, S. Segura, and A. Ruiz-Cortes, "Automated analysis of feature models 20 years later: a literature review," *Information Systems*, vol. 35, no. 6, 2010.

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

[7] N. Andersen, K. Czarnecki, S. She, and A. Wasowski, "Efficient synthesis of feature models," in *SPLC'12*, 2012, pp. 106–115.

[8] A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay, "Symbolic model checking of software product lines," in *ICSE'11*. ACM, 2011, pp. 321–330.

[9] 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*, vol. 85, no. 10, pp. 2261 – 2274, 2012.

[10] A. Gotlieb, A. Hervieu, and B. Baudry, "Minimum pairwise coverage using constraint programming techniques," in *ICST'12*, 2012, pp. 773–774.

[11] M. F. Johansen, Ø. Haugen, and F. Fleurey, "An algorithm for generating t-wise covering arrays from large feature models," in *SPLC'12*, 2012, pp. 46–55.

[12] 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 *RE'07*, 2007, pp. 243–253.

[13] S. She, K. Czarnecki, and A. Wasowski, "Usage scenarios for feature model synthesis," in *Vary'12 workshop*, 2012, pp. 13–19.

[14] E. N. Haslinger, R. E. Lopez-Herrejon, and A. Egyed, "Reverse engineering feature models from programs' feature sets," in *WCRE'11*. IEEE CS, 2011, pp. 308–312.

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

[16] M. Acher, A. Cleve, G. Perrouin, P. Heymans, C. Vanbeneden, P. Collet, and P. Lahire, "On extracting feature models from product descriptions," in *VaMoS'12*. ACM, 2012, pp. 45–54.

[17] M. Acher, A. Cleve, P. Collet, P. Merle, L. Duchien, and P. Lahire, "Reverse engineering architectural feature models," in *ECSA'11*, ser. LNCS, vol. 6903, 2011, pp. 220–235.

[18] U. Ryssel, J. Ploennigs, and K. Kabitzsch, "Extraction of feature models from formal contexts," in *FOSD'11*, 2011, pp. 1–8.

[19] A. Rabkin and R. Katz, "Static extraction of program configuration options," in *ICSE'11*. ACM, 2011, pp. 131–140.

[20] S. Apel and C. Kästner, "An overview of feature-oriented software development," *Journal of Object Technology (JOT)*, vol. 8, no. 5, pp. 49–84, July/August 2009.

[21] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps, "Generic semantics of feature diagrams," *Comput. Netw.*, vol. 51, no. 2, pp. 456–479, 2007.

[22] D. Harel and B. Rumpe, "Meaningful modeling: What's the semantics of "semantics"?" *Computer*, vol. 37, no. 10, pp. 64–72, 2004.

[23] M. Acher, P. Heymans, P. Collet, C. Quinton, P. Lahire, and P. Merle, "Feature model differences," in *CAiSE'12*, ser. LNCS. Springer, 2012, pp. 629–645.

[24] M. Acher, P. Collet, P. Lahire, and R. France, "Separation of concerns in feature modeling: Support and applications," in *AOSD'12*. ACM, 2012, pp. 1–12.

[25] Companion web page, "https://nyx.unice.fr/projects/familiar/wiki/KSynthesis."

[26] A. Classen, Q. Boucher, and P. Heymans, "A text-based approach to feature modelling: Syntax and semantics of TVL," *Sci. Comput. Program.*, vol. 76, no. 12, pp. 1130–1143, 2011.

[27] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, and C. Lucena, "Refactoring product lines," in *GPCE'06*. ACM, 2006, pp. 201–210.

[28] H. Hartmann, T. Trew, and A. Matsinger, "Supplier independent feature modelling," in *SPLC'09*. IEEE, 2009, pp. 191–200.

[29] M. Janota, V. Kuzina, and A. Wasowski, "Model construction with external constraints: An interactive journey from semantics to syntax," in *MODELS'08*, ser. LNCS, vol. 5301, 2008, pp. 431–445.