FAMILIAR: A Domain-Specific Language for Large Scale Management of Feature Models

Mathieu Acher, Philippe Collet, Philippe Lahire, Robert France

FAMILIAR: A Domain-Specific Language for Large Scale Management of Feature Models

Mathieu Acher[a], Philippe Collet[b], Philippe Lahire[b], Robert B. France[c]

[a] *University of Namur, PReCISE Research Centre, Belgium*
University of Rennes 1, IRISA, INRIA, France
[b] *University of Nice Sophia Antipolis, I3S laboratory – CNRS UMR 6070, France*
[c] *Colorado State University, Department of Computer Science, USA*

Abstract

The feature model formalism has become the de facto standard for managing variability in software product lines (SPLs). In practice, developing an SPL can involve modeling a large number of features representing different viewpoints, sub-systems or concerns of the software system. This activity is generally tedious and error-prone. In this article, we present FAMILIAR a Domain-Specific Language (DSL) that is dedicated to the large scale management of feature models and that complements existing tool support. The language provides a powerful support for separating concerns in feature modeling, through the provision of composition and decomposition operators, reasoning facilities and scripting capabilities with modularization mechanisms. We illustrate how an SPL consisting of medical imaging services can be practically managed using reusable FAMILIAR scripts that implement reasoning mechanisms. We also report on various usages and applications of FAMILIAR and its operators, to demonstrate their applicability to different domains and use for different purposes.

Keywords: Domain-Specific Language, Feature Model, Software Product Lines, Variability, Model Management

1. Introduction

In many domains, software-intensive systems have to be efficiently extended, changed, customized or configured for use in a particular context (e.g., to respond to the specific expectations of a customer) [1, 2]. The challenge for software practitioners is to develop and use the appropriate models, languages, and tool-supported techniques to produce and maintain multiple similar software products (variants), exploiting what they have in common and managing what varies among them.

To address this challenge, the paradigm of *Software Product Line* (SPL) engineering has recently emerged [2, 3]. Its goal is to produce a family of related program variants, called an SPL, for a domain. When properly applied, SPLs can be used to realize substantial economies of scale, reduce defect rates and shorten time to market. These benefits are achieved by factoring out common artifacts (e.g., requirements, components and test cases) to facilitate their reuse. But a lack of flexibility in the reusable artifacts or a scope that is too large may have severe consequences on the engineering process. For instance, the capabilities of the software platform should not be overestimated – otherwise unrealizable software products can be proposed to customers. Therefore the modeling and management of *variability* is a critical task within SPL engineering [4].

The variability of an SPL is usually described in terms of features, where a feature is a domain abstraction relevant to stakeholders, typically an increment in program functionality [5, 6]. Every product of an SPL then corresponds to a unique combination of features, called *configuration*. *Feature Models* (FMs) [7, 8, 9, 10, 11, 12] compactly represent SPL commonalities and variabilities in terms of optional, alternative and mandatory features as well as constraints over the features. An FM is basically an AND-OR graph with propositional constraints that characterizes the set of legal configurations supported by an SPL. FMs with formal semantics, reasoning techniques and tooling [6, 10, 11, 13] have become the *de facto* standard for managing variability. They will also be part of the Common Variability Language (CVL), a future OMG standard for variability modeling [14].

When applied to realistic SPLs, FMs can be *large* and *complex*, with possibly thousands of features related by numerous complex constraints [13, 15, 16]. As an extreme case, the variability model of Linux exhibits 6000+ features [17]. Furthermore, FMs can be *multiple*. Numerous authors report that maintaining a single large FM for an entire system is neither feasible nor desirable [2, 8, 18, 19, 20, 21, 22]. Numerous FMs are describing variability at various levels of abstraction, ranging from hardware descriptions [8], organizational structures [15], business or implementation details [18].

Our experience with developing SPLs in the medical imaging and video surveillance domains (details are given in Section 7) and numerous examples in the literature show that there is an increasing need to support the management of FMs on a large scale. As there can be multiple, FMs composition support is needed to group and evolve a set of similar FMs, and to manage a set of inter-related FMs. For instance, organizations can manage variability in product parts provided by external suppliers or vendors [23, 24, 21], by forming compositional SPLs [2, 25, 26]. As FMs can be large and complex, complementary decomposition support is needed to reason about local properties or to support multi-perspectives of a large FM. For instance, it is the case when the feature-based configuration process is split into different steps, each tailored for a specific stakeholder [27, 28, 29]. Furthermore, as the number of configurations in an FM is exponential to the number of features, support should be automated as much as possible. As a result, a scalable management support is needed to handle complex, large, and multiple FMs.

In previous work [30, 31, 32, 33, 34], we developed the foundations for applying the principle of *Separation of Concerns* (SoC) to feature modeling. In [31, 32], we described a set of *composition* operators for FMs (insert, merge, aggregate) that preserved semantic properties expressed in terms of configuration sets of the composed FMs. In [33], we defined a *decomposition* operator that produces a projection of an FM (a slice) using a slicing criterion supported by an algorithm that can scale for very large FMs [30]. We also defined *differencing* techniques to understand, manipulate, visualize and reason about differences. Our previous research addressed the management of FMs mainly from a theoretical perspective. Yet, a practical solution is still missing for importing, exporting, composing, decomposing, manipulating, editing and reasoning about FMs. It is especially important for FM users since numerous FMs and numerous management operations have to be combined in practice.

This article is an extended version of the SAC'2011 paper in the Programming Languages track [35], in which we described how a set of FM management operators were integrated into a *Domain-Specific Language* (DSL) called FAMILIAR (for FeAture Model scrIpt Language for manIpulation and Automatic Reasoning [36]). In this article, we show how FAMILIAR constructs and its integrated composition, decomposition and differencing operators (to name a few) can be used to support large-scale management of FMs and thus realize complex variability management tasks. Our previous works [30, 31, 32, 33, 34, 35] did not comprehensively describe how these operators are integrated into the FAMILIAR language and how this language brings benefits to SPL practitioners. In this article, we make the following contributions:

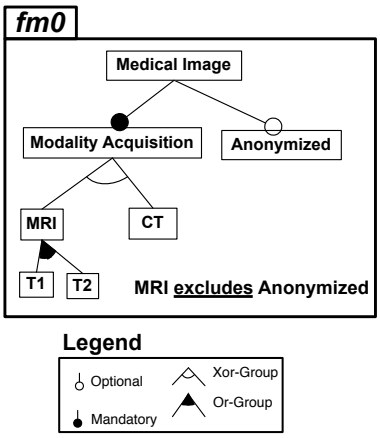
- The integration of all FM operators previously defined (e.g., **slice**, **merge**, **aggregate**) in a dedicated language, with additional DSL constructs to ensure an appropriate usage ;
- A real use of FAMILIAR for managing a repository of medical imaging services is illustrated ;
- Details about the language implementation (e.g., interpreter, reasoners) and development environment of the DSL are given ;
- An evaluation of the language is performed in different application domains (medical imaging, video surveillance) and for various purposes (e.g., reverse engineering FMs from legacy software artifacts, management of multi SPLs and supply chains, validation of SPLs).

The remainder of the article is organized as follows. After giving some background on FMs in Section 2, we motivate the needs to manage FMs on a large scale and discuss why a DSL appears to be an appropriate solution in Section 3. We give an overview of DSL constructs, syntactic facilities as well as operators provided in FAMILIAR so that FM users can import, export, edit, configure, compose, decompose, configure and reason about FMs in Section 4. We illustrate how a repository of medical imaging services can be practically managed using reusable scripts and automated reasoning techniques provided in FAMILIAR in Section 5. We describe the development environment of FAMILIAR and some implementation details (e.g., use of solvers) in Section 6. We report on various applications of FAMILIAR in different domains (medical imaging, video surveillance) and for different purposes (scientific workflow

design, variability modeling from requirements to runtime, reverse engineering) in Section 7. We also show that some of the FAMILIAR capabilities turn out to be mandatory at this scale since without the language some analysis and reasoning operations would not be possible in the different applications. We finally discuss future work in Section 8.

2. Background: Feature Models

Feature Models (FMs) were first introduced in the FODA method [8], which also provided a graphical representation through Feature Diagrams. An FM defines both a *hierarchy*, which structures features into levels of increasing detail, and some *variability* aspects expressed through several mechanisms. When decomposing a feature into subfeatures, the subfeatures may be *optional* or *mandatory* or may form *Xor* or *Or*-groups. In addition, any *propositional* constraints (e.g., implies or excludes) can be specified to express more complex dependencies between features. We consider that an FM is composed of a feature diagram coupled with a set of constraints expressed in propositional logic. Figure 1a shows an example of an FM. The feature diagram is depicted using a FODA-like graphical notation used throughout the article.



(a) FODA-like representation

$[\![fm0]\!] = \{$

$\{MI, MA, MRI, T1\},$

$\{MI, MA, MRI, T2\},$

$\{MI, MA, MRI, T1, T2\},$

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

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

$\}$

(b) set of configurations

$\phi_{fm_0} = MI$ // root
$\wedge \quad MA \Leftrightarrow MI$ // mandatory
$\wedge \quad AN \Rightarrow MI$ // optional
// Xor-group
$\wedge \quad MRI \Rightarrow MA \wedge CT \Rightarrow MA$
$\wedge \quad MA \Rightarrow MRI \vee CT$
$\wedge \quad \neg MRI \vee \neg CT$
// Or-group
$\wedge \quad T1 \Rightarrow MRI \wedge T2 \Rightarrow MRI$
$\wedge \quad MRI \Rightarrow T1 \vee T2$
// constraints
$\wedge \quad MRI \Rightarrow \neg AN$

(c) propositional formula

Figure 1: FM, set of configurations and propositional logic encoding (MI: Medical Image, MA: Modality Acquisition, AN: Anonymized)

The *hierarchy* of an FM is represented by a rooted tree $G = (\mathcal{F}, E, r)$ where $\mathcal{F}$ is a finite set of features and $E \subseteq \mathcal{F} \times \mathcal{F}$ is a finite set of edges (edges represent top-down hierarchical decomposition of features, i.e., parent-child relations between them) ; $r \in \mathcal{F}$ being the root feature.

An FM defines a set of valid feature *configurations*. A valid configuration is obtained by selecting features so that *i)* if a feature is selected, its parent is also selected; *ii)* if a parent is selected, all the mandatory subfeatures, exactly one subfeature in each of its Xor-groups, and at least one of its Or groups are selected; *iii)* propositional constraints hold. For example, in Figure 1a, MRI and CT cannot be selected at the same time, while T1 cannot be selected without MRI due to the parent-child relation between them. In Figure 1b, the set of valid configurations characterized by the FM of Figure 1a is enumerated.

Definition 1 (Configuration Semantics). *A configuration of an FM fm_0 is defined as a set of selected features. $[\![fm_0]\!]$ denotes the set of valid configurations of the FM fm_0 and is thus a set of sets of features.*

FMs have been semantically related to propositional logic [9, 10, 11]. The set of configurations represented by an FM can be described by a propositional formula ϕ defined over a set of Boolean variables, where each variable corresponds to a feature (see Figure 1c for the propositional formula corresponding to the FM of Figure 1a). The translation of FMs into logic representations allows to use reasoning techniques for numerous automated FM analyses, for example, enumerating configurations, counting configurations, interactive guidance during configuration [13, 37].

3. On Managing Multiple Feature Models

Our experience in the development of a real-world Video Surveillance Systems SPL [38] and a Medical Imaging Workflow SPL [39] suggests that support for separating concerns and synthesizing large FMs from smaller FMs can significantly improve management of complex SPLs. Our experience led us to develop composition and decomposition operators that we implemented [32, 33, 31]. Providing support for FM composition is not enough. To support effective development and management of large complex FM, SPL developers need scalable FM development environments that allow them to better control how large FMs are created, analyzed and evolved. This can be done by giving developers the means to define complex operations on FMs by combining basic FM operators that, for example, compose FMs, add new features, remove features, and support reasoning about FM properties.

3.1. Principles for Managing Large FMs

Scalable approaches to managing FMs should support, at least, *i)* separation of concerns and synthesis of large FMs from smaller FMs using decomposition and composition operators, and *ii)* rigorous reasoning about FM properties as they are created and evolved.

Support for separation of concerns may also help developers identify and act upon opportunities to reuse feature structures across different SPLs. For example, the reuse of software components between different SPLs in the consumer electronics domain is commonplace [23]. There are many different dimensions (concerns) that can form the basis for separating concerns. For example, system variability is driven by several different dimensions (e.g., product types and geographic regions). In addition, organizations are increasingly faced with the challenge of managing variability in product parts provided by external suppliers. The same observation can be made in the semiconductor industry where a set of components from several suppliers has to be integrated into a product [24]. In addition, separating concerns in FMs can help developers manage decisions made by different stakeholders (e.g., external suppliers of product parts) [40]. In general, maintaining a single FM for the entire system may not be feasible and structuring the modeling space for software product lines can become an issue [19]. A number of techniques do provide some support for separating concerns. For example, FORM [8] allows the connection of various layers of feature refinements. Pohl et al. distinguish external variability (visible to customers) from internal variability (hidden from customers) [2]. We are not aware of any approach that addresses all the issues highlighted above.

Support for separating and composing FMs must be coupled with support for reasoning about FMs before, during, and after composition. The FM notation must have an adequate formal semantics to support reasoning [10]. FM semantics are typically expressed in terms of configuration sets, that is, the meaning of an FM is typically defined as the set of configurations that satisfy the constraints expressed in the FM.

Automated analysis of FMs focuses on properties of an FM, for example, checking that an FM contains at least one product, and determining the number of valid configurations characterized by an FM [9, 13]. We are not aware of any approach that allows a developer to control when and how reasoning tools are applied in an FM development environment that supports separation and composition of concerns.

3.2. Rationale for an FM Management DSL

There are at least three possible solutions to meet the requirements above. One approach is to reuse existing FM development tools and graphical editors. The other two involve using a language, either general-purpose or domain-specific.

3.2.1. Existing (Graphical) Tools vs a New Language

Several graphical FM editors are currently available, and some do provide support for managing some aspects of FM development, for example, pure::variants [41], Gears [42], FeatureIDE [43], SPLOT [44], FaMa [45] or S^2T^2 [46]. Gears (Big Lever Software Inc) and pure::variants (Pure Systems GmbH) are commercial tools with good support for binding FMs to other models and for code generation. FeatureIDE is a comprehensive environment that interconnects with different FM management tools and has a Java API to manipulate FMs. Integration of reasoning tools is thus facilitated, for example, a tool for FM edits [47] has been integrated. Nevertheless, current tools do not fully support the composition of FMs or decomposition of an FM into several separated FMs. A conceivable solution would be to integrate our FM operators (insert, aggregate and merge) as additional functionalities inside a mainstream graphical editor. Numerous examples and our case studies indicate that manipulating several FMs

requires support for defining and replaying sequences of operations, observing properties as the FMs are manipulated, and organizing all these actions as reusable operations. These observations led us to consider developing a textual, executable language, that can be used in much the same way as scripting languages. Such a scripting language should provide *i)* basic sequencing of FM operations, *ii)* access to FM internals, *iii)* reasoning operations and *iv)* composition and decomposition mechanisms. A textual script performs a sequence of operations on FMs. Such operations are *reproducible* and *reusable*.

Obtaining the same properties in a graphical editor requires additional effort, for example, the implementation of an undo/redo system and serialization of the sequence of operations is not straightforward. The scripting aspect of the FAMILIAR language allows developers to perform operations on Feature Models by gluing together decomposition and composition operators in a straightforward manner.

It should be noted that our use of a textual scripting language does not preclude graphical counterparts built on top of the textual language.

3.2.2. A General-Purpose Language vs a Domain-Specific Language

As editors like FeatureIDE and frameworks such as FAMA or SPLOT provide an API, another conceivable solution would be to build an API extension in a mainstream programming language in order to provide support for using composition operators and other FM management operations. While this may be a feasible solution, it would require developers to be knowledgeable about the host language (i.e., Java), many require them to perform repetitive and error-prone actions (e.g., importing an FM or using reasoning operations). If the API was created with simplicity and readability as major objectives, and it is based on a limited set of concepts related to the domain of FMs, then one can argue that the API is an *internal*[1] DSL, written on top of a host language.

An internal or external DSL should allow an FM user to more quickly build the code they need to manipulate FMs. The facilities provided to the FM users must allow the description of complex operations dedicated to FMs, in both a compact and readable way, while being understandable by an expert who may not necessarily be a software engineer. An external DSL seems particularly adequate in our work as it would provide only the necessary expressive power for anticipated FM manipulations. In addition, such an external DSL should be used more easily by FM users as the learning curve is expected to be more favorable.

The decision to develop a new DSL (e.g., *when?*, *why?* and *how?*) is a difficult one as it involves both advantages/disadvantages or risks/opportunities [48, 49, 50, 51]. Essential to this decision is the notion of *domain* that determines the scope of a DSL. Automated analysis of FMs is an active area of research and is gaining importance in the SPL community. The operators we have developed and thus the language we want to create, are obviously in that domain. Benavides et al. conduct a literature review and claim that [13]: *"From the analysis of current solutions, we conclude that the analysis of FMs is maturing with an increasing number of contributions, operations, tools and empirical works."*

Hence the development of an FM manipulation DSL that integrates automated analysis concepts seems particularly adequate. Furthermore, DSLs are considered as "enablers of reuse" [50] and there is a clear opportunity to *reuse operations* already defined in the domain.

We further restrict the DSL to manipulation of *propositional FMs*. Our restriction to propositional FMs means that our DSL does not fully support extended FMs with feature attributes or constraints expressed in logics that go beyond propositional logic (e.g., [13, 52, 53]). We and others consider that more research effort is needed in the domain of extended FMs. For example, the semantics and implementation of composition and decomposition operators for extended FMs are still to be developed. Moreover Benavides et al. [13] state that one of the research challenge in feature modeling is *"to include feature attribute relationships for analyses on FMs and propose new operations of analysis leveraging extended FMs"*. Besides, FMs are usually mapped to other artefacts of an SPL (e.g., see [54, 55, 56, 57, 58, 14]). We do not consider the relationship between FMs and other artifacts, i.e., the domain of the DSL is restricted to FMs.

[1]The external/internal dichotomy is generally used to characterize DSLs. An *external* DSL is a completely separate language and has its own custom syntax. An *internal* DSL is more or less a set of APIs written on top of a host language (e.g., Java). Internal DSL is limited to the syntax and structure of its host language. Both internal and external DSLs have strengths and weaknesses (learning curve, cost of building, programmer familiarity, communication with domain experts, mixing in the host language, strong expressiveness boundary, etc.) [48]

3.2.3. Existing Languages and Associated Support

The predominant idea is that we propose a *textual* language dedicated to the *domain* of FMs. TVL [59] (for Textual Variability Language) has similar characteristics but its focus is on *specifying* FMs and not on manipulating FMs. In particular, the support for reasoning about FMs is not integrated into the language and is provided in the form of a Java library. VELVET is a language for multi-dimensional feature modeling [60]. It provides mechanisms (inheritance, aggregation, etc.) to compose[2] separated FMs on demand and aims at improving the reuse of FMs. Yet support for automated decomposition or automated reasoning (e.g., detection of anomalies in an FM) is not part of the language and is devoted to an external API. We consider that FMs expressed in TVL and VELVET can be used as inputs of an DSL for managing FMs.

Besides FeatureIDE, FaMa, SPLOT, and TVL frameworks do not pursue the objective of managing multiple FMs. Reasoning operations are dedicated to be applied on an unique FM. More importantly and to the best of our knowledge, there is no comprehensive support for composing and decomposing FMs. Moreover, framework users have typically to deal with details about FM imports or reasoning solvers. As a result, manipulations are not encapsulated by operations that hide low-level programming language level details (i.e., additional effort beyond simply manipulating FM concepts is needed) and developers are required to master the framework as well as advanced skills in Java language.

4. The FAMILIAR Language

The next subsections will detail and illustrate the main constructs of the language as well as important FM management operators.

4.1. FAMILIAR DSL constructs

We first describe the constructs of the DSL. To structure the explanations and show the specific emphasize of the constructs, we choose to explain the DSL in terms of concepts used to study general-purpose languages [61, 62]: values and types, storage and variables, bindings and encapsulation, abstractions, sequencers.

4.1.1. Values and Types

FAMILIAR is a typed language that supports both complex and primitive types. The various types supported by the language have been chosen according to the domain concepts identified in the previous section. Complex types are domain-specific (*Feature Model*, *Configuration*, *Feature*, *Constraint*, etc.) or generic *Set* which represents container values. Primitive types include *String* (e.g., feature names are strings), *Boolean*, *Enum*, *Integer* and *Real*.

A set of operations, called *operators*, are defined for a given type. To prevent FAMILIAR scripts from performing illegal operations on data, type checking is performed at run-time. Operators evaluate expression and yield one value. For example, the operator **counting** performs on a *Feature Model* and yields an *Integer* value (see Listing 1, line 9). More examples of FM management operators are given in the remainder of the section. Basic arithmetic, logical, set and string operators are also provided and perform on *Integer*, *Real*, *Boolean*, *Set* and *String*.

4.1.2. Storage and Variables

We provide user-defined *variables* to store values. Variables representing complex types record a reference to the data whereas other variables record the data value itself. (The notions of *reference* and *value* are similar to the ones used in the Java programming language). Therefore variables may be inspected and updated. In Listing 1, lines 1-4 define two variables of type *Feature Model*: *mi1* and *mi2*. More details about the operator **FM** and the internal concrete syntax for specifying FMs are given in Section 4.2.

For variables with complex types we may need to compare either the reference or the content of the recorded data so that we propose two operators. Lines 5 and 6 illustrate the use of content equality (**eq**) and reference equality (**==**) on complex types. Lines 7 and 8 show assignments to variables of type *String* (*str*) and *Set* ($fmSet$ is a set of FMs).

[2]The comparison of the compositional mechanisms offered by the VELVET language and the composition operators developed in [32, 30] is out of the scope of this article.

```
1  mi1 = FM ( MedicalImage: Modality Format Anatomy [Anonymized];
2     Modality: (PET | CT); Format: (DICOM|Nifti); Anatomy: Brain;)
3  mi2 = FM ( MedicalImage: Modality Format Anatomy [Anonymized];
4     Modality: (PET | CT); Format: (DICOM|Nifti); Anatomy: Brain;)
5  b1 = mi1 eq mi2 // b1 is true
6  b2 = mi1 == mi2 // b2 is false
7  str = "PET" // str records the value "PET"
8  fmSet = { mi1 mi2 }  // fmSet records a reference to a set
9  n = counting mi1 // n is an integer
10
11 mi3 = FM ( MedicalImage: Modality Format Anatomy Anonymized;
12    Modality: (MRI | CT); Format: Analyze; Anatomy: Kidney;)
13 f1 = parent MRI // f1 refers to feature named 'Modality' in mi3
14 f2 = root mi3 // f2 refers to feature named 'MedicalImage' in mi3
15 s1 = name f2 // s1 is a string "MedicalImage"
16 fs = children f1 // set of features named 'MRI' and 'CT' in mi3
17 nfs = size fs // 2
18
19 oldFeature = parent Analyze // 'Format' feature of mi3
20 oldFeatureBis = parent mi3.Analyze // 'Format' feature of mi3
21 conflictingFeature = parent DICOM // feature present in mi1 and mi2
```

Listing 1: A first glimpse: variables and basic operations

Note that for the variable *str*, the use of content or reference equality would return the same result which corresponds to content equality.

As stated above, types come with a set of operators. In particular, types have accessors for observing the content of a variable. Lines 13-17 of Listing 1 illustrate the use of accessors. In line 13, variable *f1* records a reference to feature Modality (the parent of feature MRI). We consider that features are uniquely identified by their names in an FM. In line 14, variable *f2* contains a reference to the feature MedicalImage (the root of the FM *mi3*). The remaining lines (15-17) illustrate operations returning *i)* the name of a feature, *ii)* the set of direct subfeatures of a given feature and *iii)* the number of elements of a set.

We provide a conditional construct (i.e., a classical **if then else**) and a loop control structure (i.e., **foreach**). Lines 2-5 in Listing 2 illustrate how to iterate over a set of variables (e.g., representing FMs, features, and configurations). A script writer can use a wildcard "*" to access to a set of elements (e.g., FMs, features). It may be placed just after "." or anywhere within a variable or feature name. For example, line 1 gathers in *varset* all the features of *mi1*.

```
1 varset = mi1.* // it can be written: varset = features mi1
2 foreach (f in varset) do
3   newName = strConcat "new_" f
4   renameFeature f as newName
5 end
```

Listing 2: Foreach and wildcard: each feature name of FM mi1 becomes prefixed with "new_"

4.1.3. Bindings and Encapsulation

Let us consider again the FAMILIAR script of Listing 1. In line 19, the identifier Analyze is used. It corresponds to the feature Analyze of the FM *mi3*. As shown in line 20, it could be expressed in an equivalent but more explicit manner. Indeed, an identifier may refer to a variable identifier (e.g., oldFeatureBis in line 20 of Listing 1) or to a feature in an FM (e.g., MRI in line 13 of Listing 1).

It may happen though that two features with the same identifier are present in two different FMs (see line 21). In this case, the execution of the script is stopped and the error is reported to the FAMILIAR user.

To allow disambiguation of variables having the same name, FAMILIAR relies on namespaces. By default, a namespace is attached to each variable of type *Feature Model* so that it is possible to identify a feature by specifying the name of the variable of type FM followed by "."

Lines 1-2 of Listing 3 illustrate the use of namespace. Although features Modality and MedicalImage are present in two FMs, *mi1* and *mi2*, they can be identified thanks to the namespace mechanism. Note that, in 3 of Listing 3, MRI appears only in *mi3* and is non-ambiguous so that there is no need to explicitly use the namespace.

```
1 children mi1.Modality // explicit notation needed
2 mi2.MedicalImage // MedicalImage exists also in mi1 and mi3
3 parent MRI // non ambiguous: equivalent to mi3.MRI
```

Listing 3: Namespaces and encapsulation

Moreover FAMILIAR provides modularization mechanisms that allow for the creation and use of multiple *scripts* in a single SPL project, and that support the definition of reusable scripts. Namespaces are also used to logically group related variables of a script, making the development more modular.

Listing 4 illustrates how FAMILIAR supports the reuse of existing scripts. Line 1 shows how to run a script contained in the file *fooScript1* from the current script. The namespace *script_declaration* is an abstract container providing context for all the variables of the script *fooScript1*. Then, in line 2 (resp. 4), we access to the set of all variables of *script_declaration* (resp. all variables starting by *mi* in *script_declaration*) using the wildcard previously explained. By default, a script makes visible to other scripts all its variables. Using **export** with several variable names means that only those variables remain visible. Using **hide** instead means that all variables mentioned are not visible.

```
1 run "fooScript1" into script_declaration
2 varset = script_declaration.*
3 export varset
4 hide script_declaration.mi*
```

Listing 4: Scripts, namespaces, and wildcard

4.1.4. Abstraction

FAMILIAR provides a *fixed* set of abstractions. It is not possible to create a new type. We consider that predefined types are sufficient for managing FMs. Besides, a script embodies a sequence of operations and can be *parameterized* using an ordered list of **parameter**s (see lines 2-3 in Listing 5). A parameter records a variable and, optionally, the type expected. Parameterized scripts are typically used to develop reusable analysis procedures for FMs and configurations. Listing 6 illustrates how a parameterized script can be called.

```
1 // fooParameterizedScript.fml: a parameterized script
2 parameter fmA : FeatureModel
3 parameter fmB : FeatureModel // type specification is optional
4
5 c1 = cores fmA
6 c2 = cores fmB
7 //...
```

Listing 5: A parameterized script

4.1.5. Sequencers

A sequencer is a construct that varies the normal flow of control. We provide only two sequencers, **exit** and **assert**. Both stop the execution of FAMILIAR. Specifically, the operator **assert** stops whenever the evaluation of the *Boolean* expression yields **false**.

```
newMI = FM (MedicalImage: Anatomy [Header]; Anatomy: (Brain|n256);)
newMI2 = copy newMI
setMandatory newMI2.Header // editing facilities: Header is now a...
//... mandatory feature in newMI2
run "fooParameterizedScript" { newMI newMI2 }
```

Listing 6: Calling a parameterized script

4.2. Operators in a Nutshell

In the remainder of the section, we describe some important FM management operators.

4.2.1. Importing and Exporting FMs

We provide multiple notations for importing and exporting FMs. It includes the support[3] of SPLOT, FeatureIDE, a subset of TVL, a subset of FAMA, and S^2T^2. A FAMILIAR user can *i)* load FMs in these notations (using **FM** constructor) ; *ii)* serialize the FMs in these notations (using **serialize**).

```
1 fm0 = FM (MedicalImage : ModalityAcquisition [Anonymized] ;
2                ModalityAcquisition : (MRI|CT); // Xor-group
3                             MRI : (T1|T2)+ ;  // Or-group
4                 MRI excludes Anonymized ; // constraint )
5 sfm0 = configs fm0 // set of configurations
6 fm0tvl = FM ("fm0.tvl") // TVL notation
7 fm0m = FM ("fm0.m") // FeatureIDE notation
8 serialize fm0 into SPLOT // export in SPLOT notation
```

Listing 7: Importing and exporting FMs

FAMILIAR also provides a concise notation, largely inspired from FeatureIDE [43] and the feature-model-synthesis project. Let us explain the notation and consider Listing 7. In line 1, the variable *fm0* actually corresponds to the FM depicted in Figure 1a, page 3. MedicalImage is the root feature. ModalityAcquisition and Anonymized are child-features of MedicalImage: ModalityAcquisition is mandatory while Anonymized is optional. MRI and CT form a Xor-group and are child-features of ModalityAcquisition. T1 and T2 form an Or-group and are child-features of MRI. In line 4, MRI **excludes** Anonymized corresponds to a propositional constraint over the set of features of *fm0* (i.e., the constraint states that features MRI and Anonymized are mutually exclusive).

Then, the variable *fm0* can be used, for example in line 5, to obtain its set of configurations (the same set enumerated in Figure 1b, page 3). FMs can also be imported in other notations using the same **FM** constructor (in line 6 we import a TVL model while in line 7 we import an FM in FeatureIDE notation). Line 8 gives an example of a serialization in SPLOT notation.

4.2.2. Handling FM configurations

The language also allows FAMILIAR users to create FM configurations, and then **select**, **deselect**, or **unselect** a feature. To **select** a feature means that the configuration includes the feature. To **deselect** means that it will not be part of the configuration. To **unselect** means that no decision has been made: the feature is neither selected nor deselected. Each of these configuration manipulation operations returns a boolean value, i.e., true if the feature selected/deselected/unselected does exist and if the choices are conformant to the constraints of the FM.

An exemplified usage of these operators is given in Listing 8. In line 1, the operator **configuration** creates and initializes a configuration of the FM *mi1* as follows: features that appear in every configuration are automatically "selected", features that appear in any configuration are automatically "deselected" while others are considered as "unselected". Lines 2-4 provide examples of the configuration manipulations. In addition, the accessors **selectedF**, **deselectedF**, **unselectedF** return respectively the set of selected, deselected and unselected features of a configuration.

[3] Details about interoperability between notations and implementation of import/export mechanisms are given in Section 6.

```
conf1 = configuration mi1 // create a configuration of mi1
b1 = select Anonymized in conf1 // feature Anonymized is selected
b2 = deselect Anonymized in conf1 // override the previous selection
b3 = unselect Anonymized in conf1 // neither selected nor deselected
```

Listing 8: Configuration facilities

4.2.3. Reasoning about FMs and Configurations

FAMILIAR provides several operators to support reasoning about FMs and configurations. Listing 9 provides a few examples of the FM manipulation and reasoning operators. The **isValid** operator checks whether a configuration is consistent according to its FM (see line 1). Indeed, a selection or deselection of feature may lead to an invalid configuration (e.g., when two mutually exclusive features are selected). The **isValid** operator can also perform on an FM (see line 3) and determines its *satisfiability* (i.e., an FM is unsatisfiable if it represents no configurations). The operator **isComplete** checks whether a configuration is *complete*, that is, whether all features have been selected or deselected.

```
b1 = isValid conf1
b2 = isComplete conf1
b3 = isValid mi1
cmp = compare mi1 mi2 // refactoring
```

Listing 9: Some operators for reasoning about configurations and FMs

In addition, the **compare** operation is proposed to determine whether an FM is a refactoring, a generalization, a specialization or an arbitrary edit of another FM. This operation is based on the algorithm and terminology used in [47] (see Definition 2). Line 4 of Listing 9 illustrates comparison capabilities based on sets of configurations of FMs. *cmp* is an *Enum* type whose possible values can be **REFACTORING, SPECIALIZATION, GENERALIZATION** and **ARBITRARY**. In the example above, *cmp* has the value **REFACTORING** since *mi1* and *mi2* represent the same set of configurations.

Other operators have been defined in FAMILIAR for reasoning about *differences* of two FMs. The interested reader is referred to [34, 63] for further details.

Definition 2 (Kind of edits and Comparison). *Let f and g be two FMs. f is a specialization of g if $[\![f]\!] \subset [\![g]\!]$. f is a generalization of g if $[\![g]\!] \subset [\![f]\!]$. f is a refactoring of g if $[\![g]\!] = [\![f]\!]$. f is an arbitrary edit of g if f is neither a specialization, a generalization nor a refactoring of g. A comparison computes the relationship between two FMs.*

4.2.4. Decomposition

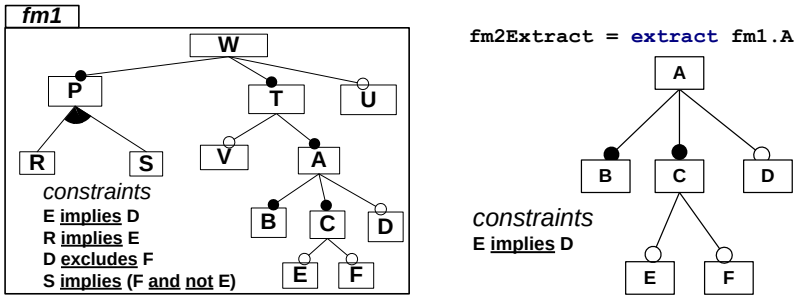
Managing a large number of features, governed by many and often complex rules, is obviously a problem per se for an SPL practitioner. Dividing FMs into localized and separated parts seem particularly well-suited solution to the problem, because SPL practitioners can focus their attention on one part at a time.

A first basic mechanism is to "copy" a sub-tree of an FM, including cross-tree constraints involving features of the subtree. It is the role of the **extract** operator. For example, let us consider $fm1$ (see Figure 2a). The extraction of the sub-tree rooted at feature A gives the FM depicted in Figure 2b.

This operator is purely syntactical and has two important limits. First, the operator ignores cross-tree constraints that involve features not present in the sub-tree. However these constraints may have an impact on the set of configurations. For example, considering the example of Figure 2a, the basic extraction fails to infer the transitivity of implications (e.g., D implies E) or that features E and F are mutually exclusive. Second, using the **extract** operator, features must belong to the same sub-tree whereas a more generic technique would be to consider any set of features of an FM (i.e., wherever they are located in the FM).

To raise the two limitations of the **extract** operator[4], we developed another decomposition mechanism, called **slice** in FAMILIAR.

[4]Despite its limitations, the operator is still part of the language since we consider that such a syntactical manipulation may be useful in some circumstances.



(a) $fm1$, input FM

(b) $fm2Extract$, basic extraction

Figure 2: FM to be decomposed and naive decomposition (extract)

The overall idea behind FM slicing is similar to program slicing [64]. Program slicing techniques proceed in two steps: the subset of elements of interest (e.g., a set of variables of interest and a program location), called the slicing *criterion*, is first identified ; then, a *slice* (e.g., a subset of the source code) is computed. In the context of FMs, we define the slicing criterion as a set of features considered to be pertinent by an SPL practitioner while the slice is a new FM. It should be noted that the slicing criterion is no longer restricted to a set of features located in a specific sub-tree.

Another important difference with the **extract** operator is that the **slice** operator takes the *configuration semantics* of the FM into account. The operator produces a partial view of an FM that characterizes a *projected* set of configurations based on a criterion. The properties guaranteed by the slice are recalled in Definition 3 and illustrated hereafter. The interested reader is referred to [30, 33] for a formal definition. Implementation details of the operator are discussed in Section 6.

Definition 3 (Slice). *We define slice as an operation on FM, denoted* $\Pi_{\mathcal{F}_{slice}}(FM)$*, where* $\mathcal{F}_{slice} = \{ft_1, ft_2, ..., ft_n\} \subseteq \mathcal{F}$ *is a set of features. The result of the slice operation is a new FM,* FM_{slice}*, such that:*

$$[\![FM_{slice}]\!] = \{\ x\ \cap\ \mathcal{F}_{slice} \mid x \in [\![FM]\!]\}$$

$[\![FM_{slice}]\!]$ *is called the projected set of configurations.*

Furthermore, the feature hierarchy of FM_{slice} *is defined as follows: its features are connected to their closest ancestor if their parent is not part of the slicing criterion.*

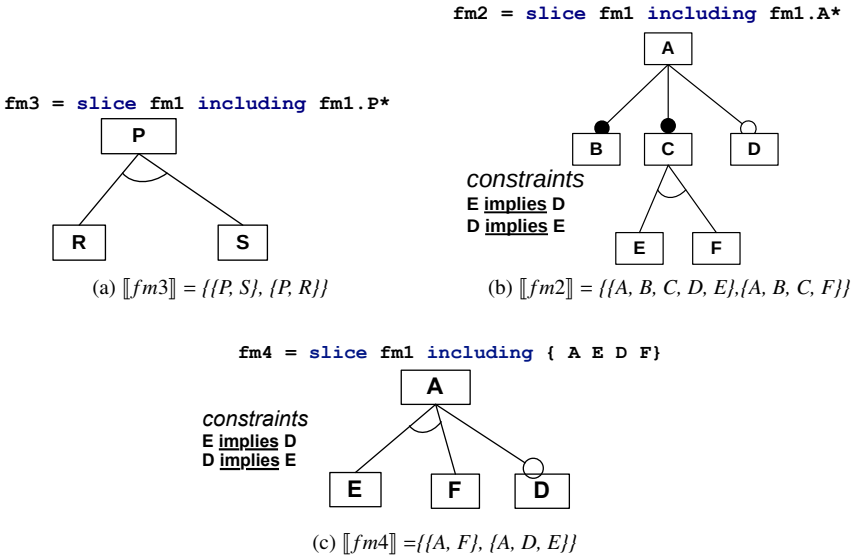
In order to be practically used in FAMILIAR, we developed a specific syntax for the **slice** operator:

```
fmS = slice anFM including | excluding setOfFeatures
```

Listing 10: Syntax of the slice operator

anFM is the input FM to be sliced and *fmS* is the resulting FM (note that *anFM* remains unchanged). The set of features that constitutes the slicing criterion can be specified either by inclusion (keyword: **including**) or exclusion (keyword: **excluding**). Language facilities are offered by FAMILIAR to specify the slicing criterion. A specific notation allows an SPL practitioner to select a set of features. Basic operators to perform the union, intersection or difference of feature sets are also provided. Some syntax examples of the **slice** operator are given in Figure 3.

In Figure 3a, *fm1.P** corresponds to a set of features including feature P *plus* the descendant features of feature P in $fm1$. We can notice that the features R and S form an Xor-group in $fm3$ whereas these features form an Or-group in the original FM $fm1$. The presence of an Or-group in $fm1$ can be seen as an *anomaly*, since the variability information is not correctly modeled [65, 13]. These anomalies can be the consequence of a human design error or an an automated operation (e.g., when two FMs are composed, see below for an example). Generally, these anomalies



(a) $[\![fm3]\!]$ = *{{P, S}, {P, R}}*

(b) $[\![fm2]\!]$ = *{{A, B, C, D, E},{A, B, C, F}}*

(c) $[\![fm4]\!]$ =*{{A, F}, {A, D, E}}*

Figure 3: Examples of slice operation applied on the FM of Figure 2a.

are regarded as a negative property of an FM since it can decrease its maintainability or understandability. Therefore the slice can be used to *correct* some anomalies.

In Figure 3b, *fm1.A** corresponds to a set of features including feature A *plus* the descendant features of feature A in $fm1$. We can notice that features E and F now form an Xor-group, thus showing again the corrective capabilities of the slice operator. In Figure 3c, the features of the slicing criterion do not necessary belong to the same sub-tree: *{A, E, D, F}* corresponds to a set of four features A, E, D, F of *fm1*.

4.2.5. Composition

Two forms of composition, aggregate and merge, are described hereafter.

Merging FMs. The **merge** operator is dedicated to the composition of FMs that exhibit similar features (i.e., features with the same name). In this case, the merge operator can be used to merge the overlapping parts of the input FMs and then to obtain a new integrated FM. The merge uses name-based matching: two features match if and only if they have the same name.

The syntax and the semantics of the **merge** operator relies on the information provided in Table 1 where the properties of the merged feature model are summarized with respect to the sets of configurations of input FMs and the mode (intersection, union, diff). The hierarchy of a merged FM restores as much as possible the parent-child relationships of the input FMs. Algorithms for automatically merging FMs are given and compared in [31, 32]. We give some implementation details in Section 6.

In FAMILIAR, the merge operators act on a set of FMs and produce FMs with semantics properties according to the mode specified by the programmer. For example, the merge in intersection mode computes a new FM that represents the set of common configurations of the input FMs[5].

Listing 11 is part of a script that uses the merge operator in intersection mode. In line 5, the merge operator in intersection mode is applied on *mi4* and *mi5* and produces a new FM that can be manipulated through the variable *mi_inter*. In line 8, we check that *mi_inter* is equal to *mi_inter_expected*. The binary operator **eq** is specific to variable complex types. In particular, two variables of FM type are equal if *i)* they represent the same set of configurations, i.e., the **compare** operator applied to the two variables returns **REFACTORING** and *ii)* they have the same hierarchy.

[5]The merge in intersection and diff mode can be arguably seen as decomposition operators since the resulting FM is typically smaller than the input FMs. Nevertheless, we prefer to use the term "composition" to reflect the idea of combining different models together.

Mode	Semantics properties	Mathematical notation	FAMILIAR notation
Intersection	$[\![FM_1]\!] \cap [\![FM_2]\!] \cap \ldots \cap [\![FM_n]\!] = [\![FM_r]\!]$	$FM_1 \oplus_\cap FM_2 \oplus_\cap \ldots \oplus_\cap FM_n = FM_r$	fmr = **merge intersection** { fm1 fm2 ... fmn}
Union	$[\![FM_1]\!] \cup [\![FM_2]\!] \cup \ldots \cup [\![FM_n]\!] = [\![FM_r]\!]$	$FM_1 \oplus_{\cup_s} FM_2 \oplus_{\cup_s} \ldots \oplus_{\cup_s} FM_n = FM_r$	fmr = **merge union** { fm1 fm2 ... fmn}
Diff	$\{x \in [\![FM_1]\!] \mid x \notin [\![FM_2]\!]\} = [\![FM_r]\!]$	$FM_1 \setminus FM_2 = FM_r$	fmr = **merge diff** { fm1 fm2 }

Table 1: Merge: semantics properties and notation

```
1  mi4 = FM ( MedicalImage: Modality Format Anatomy [Anonymized];
2  Modality: (v10.1|v10) ; Format: (NiftiII|Analyze) ; Anatomy: Brain;)
3  mi5 = FM ( MedicalImage: Modality Format Anatomy [Header]; Modality:
4  (v10.1|v10|v9) ; Format: NiftiII ; Anatomy: (Kidney|Brain);)
5  mi_inter = merge intersection { mi4 mi5 }
6  mi_inter_expected = FM ( MedicalImage: Modality Format Anatomy ;
7   Modality: (v10.1|v10) ; Format: NiftiII ; Anatomy: Brain ;)
8  assert (mi_inter eq mi_inter_expected)
9  mi_union = merge union { mi4 mi5 }
10 n_union = counting mi_union // number of valid configurations
11 n_expected = counting mi4 + counting mi5 - counting mi_inter
12 assert (n_union eq n_expected)
```

Listing 11: Merge operator in union and intersection mode

In line 9, the merge operator in union mode is applied on *mi4* and *mi5* and produces a new FM that can be manipulated through the variable *mi_union*. In lines 10-12, we check the following property:

$$|[\![mi4]\!] \cup [\![mi5]\!]| = |[\![mi4]\!]| + |[\![mi5]\!]| - |[\![mi4 \cap mi5]\!]| = |[\![mi\_union]\!]|$$

using **counting** operations (i.e., the value of n is equal to $|[\![mi\_union]\!]|$).

Aggregating FMs. A second form of composition is the **aggregate** operator. Contrary to the merge operator, it assumes that separated FMs do not have features with the same name. It also supports cross-tree constraints, written in propositional logics, over the set of features so that separated FMs can be inter-related. The input FMs are aggregated under a synthetic root *synthetic* so that the root features of input FMs are child-mandatory features of *synthetic*. In addition, the propositional constraints are added in the resulting FM. For example, the aggregate operator can be used to compose three FMs $fm5$, $fm6$ and $fm7$ together with constraints (see Figure 4).

When FMs are related through constraints, some features may become *dead* (see Definition 4) or *core* features (see Definition 5). For example, the feature E is a dead feature in $fm8$ (see Figure 4), resulting from the aggregation of $fm5$, $fm6$ and $fm7$. FAMILIAR provides two operators to compute the set of core and dead features of an FM (see lines 7–8 of Listing 12).

Definition 4 (Dead features). *A feature f of FM is dead if it cannot be part of any of the valid configurations of FM. The set of dead features of FM is noted $deads(FM) = \{f \in \mathcal{F} \mid \forall c \in [\![FM]\!], f \notin c\}$*

Definition 5 (Core features). *A feature f of FM is a core feature if it is part of all valid configurations of FM. The set of core features of FM is noted $cores(FM) = \{f \in \mathcal{F} \mid \forall c \in [\![FM]\!], f \in c\}$*

The presence of dead features is usually considered as an anomaly [13] Such "anomalies" can be reported to an FM user. But it would be even better to automatically remove the anomalies. For this purpose, we use the **slice** operator (see line 11 of Listing 12) and its corrective capabilities [30] to *i)* remove dead features (here: E is removed due to the constraint $C \Rightarrow \neg E$) *ii)* to set the variability information (here: the feature C, originally optional, becomes mandatory due to the constraints). Figure 4 recaps the situation at the end of the script execution.

The simple example shows that the composition of FMs (i.e., $fm5$, $fm6$, $fm7$) can lead to the presence of anomalies and that FAMILIAR operators can be used to automatically detect and remove them.

```
fm5 = FM (A : B [C] [D] ; D : (E|F); C excludes E; ) // E, F: Xor
fm6 = FM (I : J [K] L ; )        // K is optional
fm7 = FM (M : (N|O|P)+ ; ) // M, N, O, P form an Or-group
cst = constraints (J implies C ; )
fm8 = aggregate fm* withMapping cst
// recall: fm* is equivalent to { fm5 fm6 fm7 }
dfm8 = deads fm8
cfm8 = cores fm8

// simplify fm8 (e.g., by removing dead features)
fm9 = slice fm8 including fm8.*

op = operator fm9.F
assert (op eq mand) // E is mandatory and no longer in an Xor-group

// renaming of the root feature
rfm9 = root fm9
renameFeature rfm9 as "MySPL"
```

Listing 12: Composition of FMs, anomalies detection and correction

5. An Illustrative Application: Management of Multiple SPLs

In some SPL environments, support for manipulating *multiple SPLs* (i.e., a set of SPLs) may be needed [66, 25, 15, 24, 19, 21]. Some of these SPLs may be developed and maintained by external vendors or suppliers, and some of them may *compete* to deliver similar products. For example, in [35], we developed a case study in which a family of laptops is constituted of customizable components (chipsets, processors, graphic cards, etc.) provided by competing vendors. In this section we illustrate how FAMILIAR can be used to support the management of multiple SPLs in the medical imaging domain (see also the case study ① in Section 7).

Motivating Scenario. In the medical imaging domain, several highly customized software services are provided by different suppliers (e.g., researchers or scientific teams). The goal for a medical imaging expert is to assemble these services in a processing chain, called *workflow*. The tasks of identifying, tailoring and composing these services with their variability become tedious and error-prone. There is thus a strong need to manage the variability so that developers can more easily choose, configure and compose those services. To tackle this problem, our approach is to consider services as SPLs while the entire workflow is then seen as a multiple SPL in which the different service SPLs are composed. An example scenario is presented in Figure 5. The scenario involves three steps.

In the first step the medical imaging expert produces an FM with no assumptions about the services provided by external suppliers – see ①. A family of workflow is designed for the different processes of the workflow: for example, Intensity Correction, Segmentation, Registration. It corresponds to an aggregation of several FMs, each describing the variability of a process.

The next step is to choose which suppliers' services (if any) fulfill the features specified for each process of the workflow. In the scenario, for instance, the need for an external supplier for a Registration medical imaging service is identified. This requirement results in the generation of an FM that represents the offerings of the three suppliers for the Registration (see ②). The FM of each supplier's services is built from its services' descriptions. Once the requirements of the medical imaging expert is mapped with the suppliers' offers (see ③), reasoning operations can be performed. Among others, the medical imaging expert can use FMs to *i)* determine whether the set of suppliers is able to provide the entire set of services and to cover all combinations of features or to *ii)* identify missing services, and to *iii)* eliminate the suppliers that do not provide the required services.

Building FMs' Repositories. In the script presented in Listing 13, $Supplier_1$ proposes eight registration services, each one being distinguished from the others by features. (Note that we can use the same name convention for all suppliers since namespaces are used to disambiguate names.) The set of registration services can then be organized

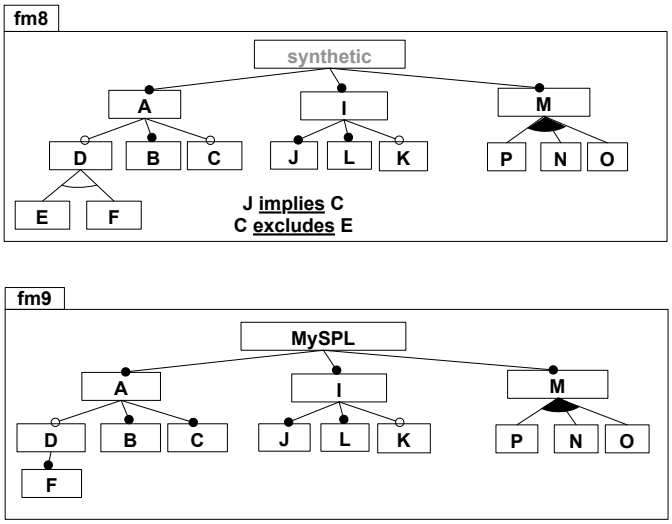


Figure 4: The two FMs $fm8$ and $fm9$ at the end of FAMILIAR script execution (see Listing 12)

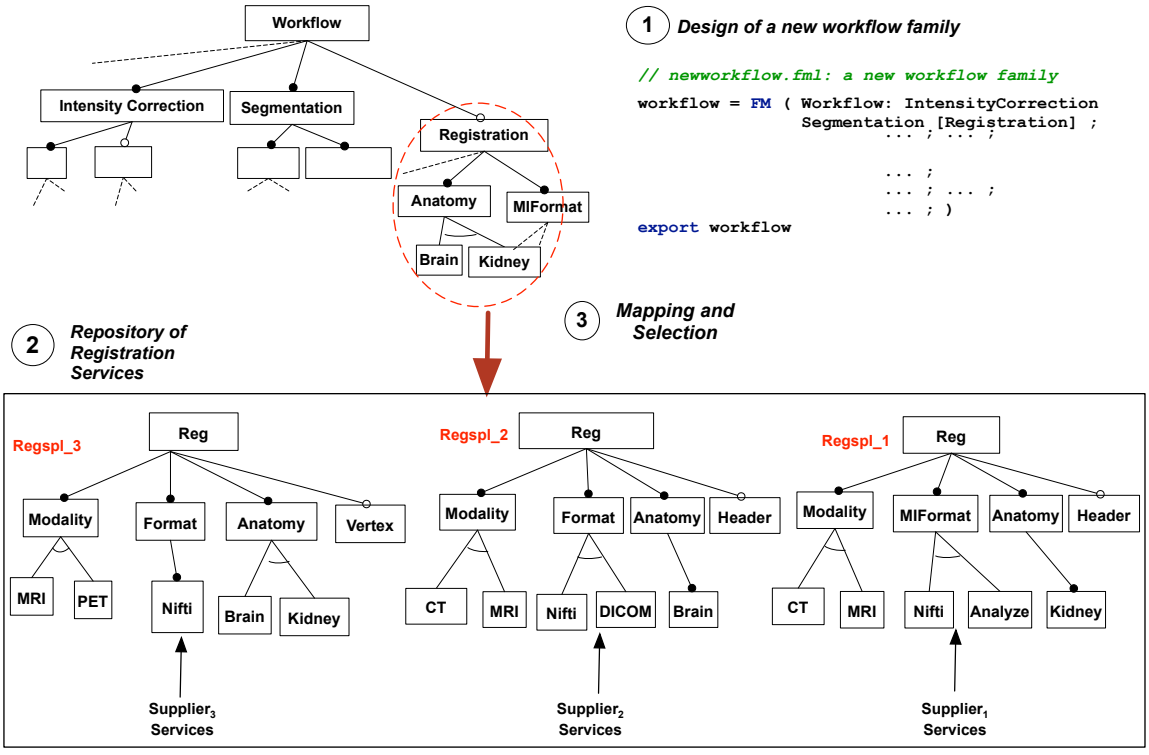


Figure 5: Managing Multiple SPLs

(e.g., grouped together) by the medical imaging expert within an FM.

We consider that services exhibit no variability: Each registration service description is represented as an FM in which all features are mandatory (see Listing 13).

```
1 // RegSupplier_1.fml: registr. services' specification of Supplier_1
2 Regservice1 = FM ( Reg: Modality Format Anatomy ; Modality: MRI ; ..)
3 Regservice2 = FM ( Reg: Modality Format Anatomy ; Modality: CT ; ..)
4 //...
5 Regservice8 = FM ( Reg: Modality Format Anatomy Anonymized ;
6       Modality: CT ; ..)
```

Listing 13: Descriptions of registration services

Building an FM from the set of existing services can be done by applying the merge operator in union mode on the set of corresponding FMs (e.g., line 3 in Listing 14). An FM is produced for each supplier (e.g., *Regspl_*1 FM for $Supplier_1$).

```
1 // repositoryReg.fml: repository of registration services
2 run "RegSupplier_1" into Regsupp1
3 Regspl_1 = merge union Regsupp1.*
4 run "RegSupplier_2" into Regsupp2
5 Regspl_2 = merge union Regsupp2.*
6 run "RegSupplier_3" into Regsupp3
7 Regspl_3 = merge union Regsupp3.*
8 renameFeature Regspl_1.MIFormat as "Format" // aligning terms
9 export Regspl_* // export the three suppliers' FMs/SPLs
```

Listing 14: Building a repository of FMs with the merge operator

A repository of registration services is represented by a set of different FMs (one per supplier). Similarly, other repositories can be built for other kinds of services (intensity correction, segmentation, etc.) and imported in a script (see lines 4-6 in Listing 15).

```
1 // workflowScenario.fml: implementation of suppliers' scenario
2 run "newworkflow" // new family of workflow firstly designed
3 // load repositories
4 run "repositoryReg" into Reg
5 run "repositorySegm" into Segm
6 run "repositoryIntensity" into intensity
```

Listing 15: Loading the workflow specification (see Figure 5) and the repositories of FM services (part 1 of "workflowScenario.fml")

Mapping Repositories. The FAMILIAR script presented in Listing 16 describes mappings from FMs in repositories to the medical imaging expert's FM. For each part of the family of workflow (specified by the script *newworkflow.fml* of Figure 5), it is necessary to determine which FMs and suppliers are suitable to provide the given service. Indeed, the family of workflow specifies different alternatives for the choice of a registration service. In order to reason about the registration service of FM $workflow$, the variability information of the registration service (sub-tree rooted at feature Registration) is first extracted. We use the **slice** operator to extract the FM, *originalReg* (line 9 in Listing 16).

Each valid configuration of *originalReg* should correspond to at least one service provided by a supplier and present in the registration service repository. It may happen that such a property is not respected and two cases have to be considered. First, the intersection between the set of services of *originalReg* and the set of suppliers' service may be empty (lines 16-20). Verifying this property can be done by first performing the merge operations on *originalReg*,

Regspl_1, *Regspl_2* and *Regspl_3* (line 16). This gives a new FM *mi_merged* and its satisfiability can then be controlled, i.e., checking whether or not there is at least one valid configuration (lines 17-20). Another possibility is that the family of workflow offers to medical imaging experts some services that cannot be *entirely* provided by suppliers (lines 22-27). In this case, *originalReg* is a *generalization* or an *an arbitrary edit* of the union of set of suppliers' services (line 22). Performing a **merge diff** operation (see Table 1) assists users in understanding which set of services is missing (line 23-26).

```
7  // we map the FM Reg (workflow) with the FM registration repository
8  allServicesReg = merge union Reg.* // Regspl_1, Regspl_2, Regspl_3
9  originalReg = slice workflow.Registration* // semantical "extraction"
10
11 // alignment: renaming terms to be coherent with the repository
12 renameFeature originalReg.MIFormat as "Format"
13 renameFeature originalReg.Registration as "Reg" //...
14
15 /***** checking the availability of services *****/
16 mi_merged = merge intersection { originalReg allServicesReg }
17 if (not (isValid mi_merged)) then
18         print "No service can be provided"
19         exit // stop the program
20 end
21 cmp_mi = (compare originalReg allServicesReg)
22 if (cmp_mi eq GENERALIZATION || cmp_mi eq ARBITRARY) then
23   mi_losed = merge diff { originalReg allServicesReg } // missing
24   s_losed = configs mi_losed // set of configurations of mi_losed
25   n_losed = counting mi_losed // number of configurations
26   println n_losed " service(s) cannot be provided: " s_losed
27 end
28 assert (cmp_mi eq REFACTORING || cmp_mi eq SPECIALIZATION)
```

Listing 16: Checking availability of registration services (part 2 of "workflowScenario.fml")

In other cases, *all* services of *originalReg* can be provided by suppliers. For example, the property holds for the FMs depicted in Figure 6: the set of configurations of *originalReg* is included or equal to the union of set of suppliers' services (we assert this property in line 28).

Selecting Suppliers. At this point, the medical imaging expert needs to determine which suppliers can provide a subset of the services of *originalReg*. We now consider the lines 29-40 of Listing 17 that implement a solution.

First, some suppliers cannot provide at least one service corresponding to any configuration of *originalReg* (lines 31-33) and so should not be considered. Figure 6 illustrates the situation: $Supplier_3$ is no longer available since the intersection between $[\![originalReg]\!]$ and $[\![Regspl\_3]\!]$ is the empty set. Some suppliers offer services that correspond to a valid configuration of *originalReg* but also offer out-of scope services. To remove these services, a merge in intersection mode is systematically performed to restrict attention to the set of relevant supplier services (line 31). For example, the feature Multi is no longer included in the set of services of $Supplier_1$ and $Supplier_2$ (see $Regspl'\_1$ and $Regspl'\_2$ in Figure 6) while $Supplier_2$ is now able to deliver only one service.

At the end of the loop, *Reg_suppliers_in* corresponds to a set of FMs, where each FM represents a family of services offered by a given supplier. Then some choices can performed in *originalReg* (e.g., in order to fix a given medical imaging format) and similar sequences of merge operations are executed to update the suppliers offer.

Refactoring and Reusable Scripts. The FAMILIAR script depicted in Listing 15, 16 and 17 can fully realize the scenario of Figure 5. Nevertheless, the script has some limitations.

First, when users select or deselect features, some suppliers may become unable to provide services corresponding to the new requirements. We want to perform validity checks *at each step* of the configuration process. An approach where sequences of code are copied from above and pasted in again is not desirable.

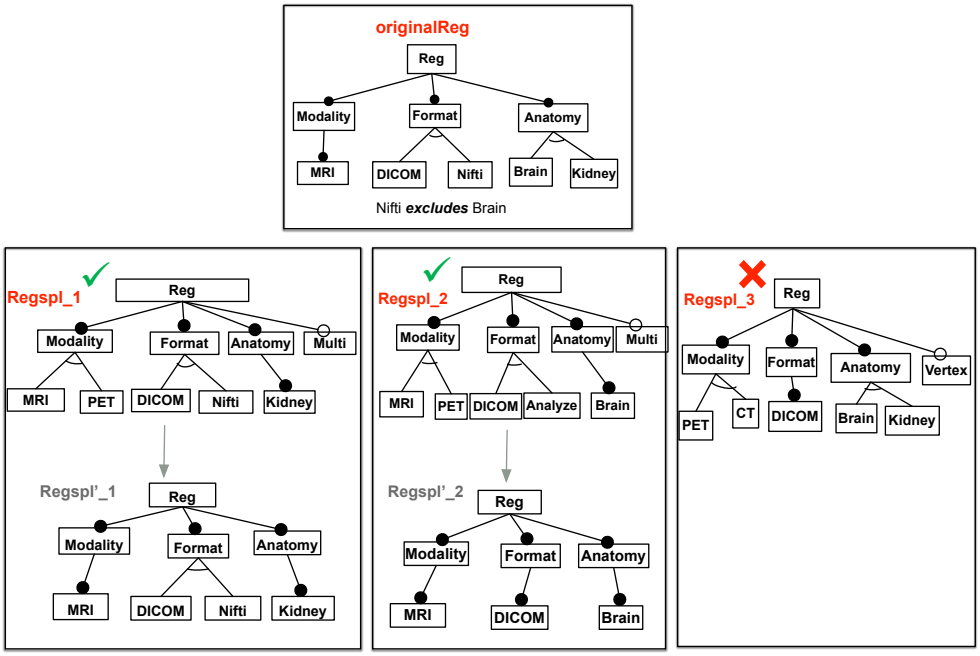


Figure 6: Available suppliers and services

```
Reg_suppliers_in = setEmpty // create an empty set
foreach (supplReg in Reg.*) do
        // checking each supplier providing Regs
        fmReg_inter = merge intersection { originalReg supplReg }
        bReginter = isValid fmReg_inter
        if (not bReginter) then
           println "The supplier is unable ...:\t" supplReg
        else
           setAdd Reg_suppliers_in fmReg_inter // add relevant FM
        end
end
assert ((size Reg_suppliers_in) >= 1) // available supplier >= 1
```

Listing 17: Determining suppliers that provide registration services (part 3 of "workflowScenario.fml")

Second, reasoning operations are planned to be performed on *each part* of the workflow (as done with the registration service). The current script does not allow a programmer to reuse the repetitive tasks performed on FMs, leading again to duplicate code.

Modularization mechanisms provided by FAMILIAR can be applied to raise the limitations mentioned above. In particular, the checking operations (i.e., *i)* for checking availability of services and *ii)* for computing available suppliers) can be performed in two parameterized scripts. When performing choices, users can have feedback from available suppliers (even if the set of features is not fully selected or deselected) by calling the two parameterized scripts. Listing 18 illustrates the principle for registration services. For each service of the workflow (e.g., segmentation and intensity correction services), similar checking operations and scripts can be reused.

6. Environment and Implementation

We provide an Eclipse-based development environment (see Figure 7) for FAMILIAR that is composed of:

```
// workflowScenario2.fml
// originalReg, allServicesReg and Reg are loaded as
// previously illustrated with workflowScenario.fml
run "checkAvailability" { originalReg allServicesReg } into
mi_availability
if (mi_availability.available) then
        run "availableSuppliers" { Reg.* originalReg } into
mi_suppliers
end
```

Listing 18: Refactoring of "workflowScenario.fml": modularizing the code

- an Eclipse text editor (including syntax highlighting, formatting, code-completion, etc.) ;
- an interpreter that executes the various FAMILIAR scripts ;
- an interactive toplevel, connected with graphical editors.

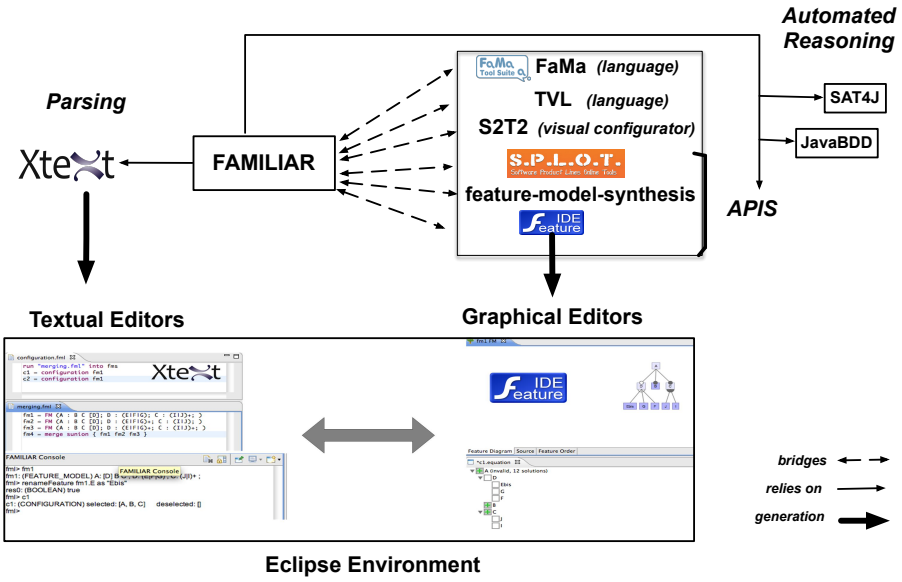


Figure 7: FAMILIAR: ecosystem and environment

Ecosystem. FAMILIAR is developed in Java language using Xtext [67], a framework for the development of external DSLs. We reuse Xtext facilities to parse FAMILIAR scripts and develop the Eclipse text editor. FAMILIAR is connected with several other languages and frameworks (see Figure 7). Therefore several notations can be used for specifying and serializing FMs (as mentioned in Section 4.2).

Some notations (TVL, FaMa) support feature attributes and non-Boolean constructs. As FAMILIAR is restricted to the management of propositional FMs (see Section 3.2), we only consider their Boolean forms. Other notations (SPLOT, FeatureIDE, S^2T^2) are expressively complete with respect to propositional logics. Therefore their semantic domain is the same and there are only syntactical differences. The formats supported by FeatureIDE and S^2T^2 only allow one feature group per feature whereas FAMILIAR, as SPLOT, supports FMs with several features groups per feature. When exporting an FM to FeatureIDE and S^2T^2 the kinds of transformations described in [68] are internally applied. They eliminate feature groups that belong to a same parent feature by introducing new intermediate features.

The connection with other languages and frameworks has several benefits. First, the support of different notations encourages *interoperability* between feature modeling tools. As an FM or a configuration can be exported (using the **serialized** operation), outputs generated by FAMILIAR can be processed by other third party tools. Specifically, the connection with the FeatureIDE framework [43] allows us to reuse the graphical editors (for editing and configuring FMs). All graphical edits are synchronized with the content of program variables and all interactive commands are synchronized with the graphical editors. We also developed a bridge with the configurator S^2T^2 [69] that provides interactive visualization mechanisms [70].

Perhaps more importantly, the support of different formats allows one to easily reuse state-of-the-art reasoning operations already implemented in existing frameworks. For implementing the **compare** operator, we reuse an efficient technique to reason about edits described in [47] and implemented in FeatureIDE [43]. The project feature-model-synthesis implements an algorithm to synthesize an FM from a propositional formula. We reuse and adapt some techniques of the algorithm to implement the **merge** and the **slice** operators [33]. We also reuse some heuristics developed in SPLOT [44] to compile large FMs into BDDs. More details are given below.

Implementation of Operators. Numerous operators of FAMILIAR manipulate or reason about the set of configurations of FMs. These operations are difficult computational problems (most of them are NP-complete) and require efficient algorithms. We rely on state-of-the-art techniques that consist in encoding an FM as a propositional formula (see Section 2). *Constraint Satisfaction Problem* (CSP), *SAT* (for SATisfiability) or *Binary Decision Diagrams* (BDDs) solvers can then be applied to implement FM operators. In FAMILIAR, two reasoning back-ends are internally used and perform over propositional formulae: SAT solvers (using SAT4J) and BDDs (using JavaBDD) - see Figure 7. Both have strengths and weaknesses when reasoning about FMs (see below for more details).

We extensively reuse the efficient techniques developed in SPLOT and FeatureIDE framework to implement FAMILIAR operators such as **isValid**, **cores**, **compare**, etc. The implementation of the **merge** and **slice** operators also relies on satisfiability techniques. A key property of the two operators is that they compute a new propositional formula representing the expected set of configurations (i.e., the projected set of configuration in the case of the slice, and the union, intersection and difference of sets of configurations in the case of the merge). This new formula is then used by other reasoning operators. For example, in lines 2 and 3 of Listing 19, the formula of the merged FM fmR is used to count the number of its configurations or compute its core features.

```
1 fmR = merge intersection { fmA fmB fmC } // formula is computed
2 nR = counting fmR // propositional formula is used
3 cR = cores fmR // propositional formula is used
4 serialize fmR into SPLOT // synthesis of FM is needed
```

Listing 19: Propositional formula and lazy synthesis

In some cases (e.g., when an FM has to be visualized or serialized), though, we need to construct a complete FM, including its hierarchy, variability information and cross-tree constraints. For example, in line 4 of Listing 19, we need to serialize fmR. The key idea is to *synthesis* an FM from the propositional formula. Details of the synthesis procedure (e.g., how we select a hierarchy and restitute variability information) are out of the scope of this article and are described in [30, 34].

FAMILIAR implements a *lazy strategy*: for all **merge** and **slice** operations, we synthesize the FM from the propositional formula only when needs be (e.g., it is the case in line 4 of Listing 19). The lazy strategy is useful since the synthesis procedure extensively relies on satisfiability techniques and is thus costly in time and space (see [11, 30, 17, 71, 34] for more details).

Reasoning back-end and Performance Analysis. A notable benefit for FAMILIAR users is that they do not have to deal with implementation details. They can directly use operators, thus focusing on domain concepts (FMs, configuration) and avoid accidental complexity. In practice, the order of complexity of FMs publicly available in SPLOT's repository [44] can be handled without any difficulty with the operators provided in FAMILIAR.

Yet, we chose to provide more control to FAMILIAR users. We indeed observed that the choice of a reasoning back-end can influence the performance of an operator, especially for larger FMs with thousands of features (i.e.,

randomly generated or the Linux FM). As an extreme case, it can even preclude the use of an operator (e.g., due to an excessive memory usage). FAMILIAR makes possible to *choose* a SAT or BDD-based implementation (except for the **merge** operator, see below). Using some annotations (see Listing 20 for an example), FAMILIAR users can fix the choice of a particular reasoning back-end for a given operator; otherwise, reasoning back-ends are chosen by default.

```
counting @backend=SAT fm1 // SAT solvers are used
counting fm1 // BDDs are used (default)
```

Listing 20: Setting the reasoning back-end for particular operations

We now discuss empirical results, reported in the literature [47, 72, 13, 30, 73, 45], that can help to efficiently choose such reasoning back-ends for operators of FAMILIAR.

Benavides et al. analyzed the performance of CSP, SAT, and BDD solvers [13, 74] in checking satisfiability of an FM (corresponding to **isValid** operator) and counting the number of valid configurations of an FM (corresponding to **counting** operator). They reported that *i)* BDDs are faster than SAT solvers for the **isValid** operator, but with a ten times higher memory usage ; *ii)* SAT solvers do not scale well when **counting** an FM whereas BDDs are much faster for the same operation. Pohl et al. [72] present a performance comparison regarding nine contemporary high-performance solvers, three for each base problem structure (BDD, CSP, and SAT). Some operations on 90 FMs from the SPLOT repository [44] are considered and correspond to the operators **isValid**, **configs** and **counting** of FAMILIAR. The results confirm the findings of Benavides et al. by showing that BDD solvers outperform other solvers on real large FMs. As stated, the results of their study *"facilitate the choice of the appropriate solver depending on the size of the models to be analyzed and the intended analysis operation"*.

The two previous studies only focus on some reasoning operations. Many others are still to be considered and empirically evaluated. Thüm et al. empirically showed that their SAT-based algorithm (reused in FAMILIAR) can efficiently **compare** FMs with thousands of features [47]. To the best of our knowledge, there is no empirical study of a BDD-based implementation of the same operator. Mendonca et al. report that FMs with a number of features up to 2000 cannot be compiled to BDD, even with the use of heuristics [73]. Therefore a SAT-based implementation can handle larger FMs than a BDD-based implementation. As future work, we plan to empirically compare the response time of the two reasoning back-ends for this operator.

The memory limit of a BDD-based implementation has also consequences on other operators. For example, it is not possible to use the **slice** operator with BDDs for FMs with 2000+ features. In [30], we developed heuristics for SAT solvers and showed that the **slice** operator can handle FMs with 10000 features. Finally, the performance of the **merge** operator is reported in [30]. Currently, FAMILIAR only proposes a BDD-based implementation of the **merge** operator. To the best of our knowledge, there is no existing work proposing the use of SAT solvers for this operator. Hence we cannot compare the two kinds of implementation.

FAMILIAR reuses state-of-the-art reasoning techniques. It allows FM users to efficiently operate over complex FMs with thousands of features. Another benefit of reuse is that the empirical results mentioned above also apply to FAMILIAR operators. We exploit these results to choose default reasoning back-ends (e.g., the choice of BDD for **counting** and **isValid**, the choice of SAT solvers for **slice** and **compare**). Nevertheless, some choices are still arbitrary for some operators of FAMILIAR. We recognize that an empirical comparison of BDD and SAT-based implementations for each FAMILIAR operator is needed. It would perhaps improve our default choices. For this purpose, existing benchmarks found in the literature have to be extended. We leave it as future work.

7. Evaluation

We now report on our experience with FAMILIAR in five different case studies. We then discuss key properties of the FAMILIAR DSL.

7.1. Case Studies

Table 2 summarizes the different case studies in which FAMILIAR has been involved. We believe the case studies are representative of large-scale management problems: FMs are large (hundreds of features), complex (numerous

relationships between the FMs) and multiple. For each case study, we briefly describe the application context (domain and goal), the stakeholders involved, how FAMILIAR has been used, the order of complexity in terms of feature modeling and the benefits observed.

Case study (domain)	Stakeholders	Complexity
① Composing Multiple Variability Artifacts (medical imaging, grid computing)	Repository maintainer, Medical imaging/Grid experts, Workflow designer	100+ inter-related FMs, 10+ features per FMs
② Modeling Variability From Requirements to Runtime (video surveillance systems)	Video surveillance expert, Software engineer	2 FMs, 77 features and 10^8 configurations in VSAR, 51 features and 10^6 configurations in PFC, 39 cross-tree constraints
③ Management of Product Line and Software Variability (any domain)	Products manager, Software engineer	2 FMs, 25 features in fm_{PL} (162 configurations), 11 features in $fm_{software}$ (13 configurations), 13 cross-tree constraints
④ Reverse Engineering FMs (component and plugin based systems)	Software architect	2 FMs with 92 features in total, from $\approx 10^6$ to $\approx 10^{11}$ configurations, 158 cross-tree constraints,
⑤ From Product Descriptions to Feature Models	Product manager, Domain analyst	100+ FMs, 10+ features per FMs

Table 2: Case studies and FAMILIAR

① **Composing Multiple Variability Artifacts to Assemble Coherent Workflows.** In the medical imaging domain, we proposed a comprehensive modeling process and tooling support (including FAMILIAR and a set of domain specific languages) for combining multiple variability artifacts with the purpose of assembling coherent processing chains, called workflows. Separated FMs are used to describe the variability of the different artifacts. At each step of the workflow design, automated reasoning techniques assist medical imaging experts in selecting services from among sets of competing services organized in a repository while guaranteeing that the composition of services does not violate important constraints. Repositories of FMs are built and organized as reusable FAMILIAR scripts. As shown in Section 5, FMs that document variability of services are merged together; querying a repository of FMs is realized using FAMILIAR operators (**merge**, **compare**, **slice**, etc.).

Variability Requirements Specification. FAMILIAR is used to specify variability requirements (e.g., medical image formats, algorithm method, deployment information, etc.) within services of the workflow. More precisely, FAMILIAR is *embedded* into the DSL *Wfamily* which enables stakeholders to:

- *import* FMs from external files while performing some high-level operations (extraction, renaming/removal of features, etc.). For example, the user can load an existing FM from a repository, then extract the sub-parts that are of interest and finally specialize the different FMs ;
- *weave* FMs to specific places of the workflow ;
- *constrain* FMs within and across services by specifying propositional constraints. Each FM that have been woven has an unique identifier and can be related to another through cross-tree constraints.

Fully-fledged Workflow Configuration. FAMILIAR code is *generated* from the workflow analysis and a *Wfamily* specification, for example, to reason about data compatibility between services. The FAMILIAR code is then interpreted to check the consistency of the whole workflow, to report errors to users as well as to automatically propagate choices. Users can incrementally configure, using graphical facilities provided by FeatureIDE editors, the various FMs of the workflow. Finally, in order to derive a final workflow product, competing services can be chosen from among sets of services in the repository using FAMILIAR reusable scripts. Our first experimental results showed that the overall approach offers an adequate user assistance and degree of automation for managing the large number of features and

FMs, thereby decreasing the effort and time needed [39].

② **Modeling Variability From Requirements to Runtime.** In the development of video surveillance systems, we observed multiple variability factors, both for i) specifying an application, including the environments and contexts where an application is deployed and run (e.g., lighting conditions, information on the objects to recognize), the quality of service required, etc. ii) describing the software platform, including the number of components, their variations due to choices among possible algorithms, the different ways to assemble them, the number of tunable parameters, etc. We applied the principles of Separation of Concerns and represented the variability of the application requirements and the variability of the software platform as two separated FMs [38]. 77 features and 10^8 configurations were present in the application requirements FM, noted VSAR (for Video Surveillance Application Requirements), while 51 features and 10^6 configurations were present in the software platform FM, noted PFC (for PlatForm Configuration). The relationships between the two FMs were described as 39 rules (propositional constraints) relating features across models.

Reachability Property Checking. Before the execution of a system, FMs are used to verify important properties. Among others, one want to guarantee the *reachability* property, i.e., that for *all* valid specifications and contexts, there exists at least one valid software configuration. In terms of FMs, the reachability property can be formally expressed as follows:

$$\forall c \in [\![\text{VSAR}]\!], c \in [\![\Pi_{\mathcal{F}_{\text{VSAR}}} (VS_{full})]\!] \quad (1)$$

where VS_{full} is the aggregation of VSAR, PFC together with cross model constraints, while $\mathcal{F}_{\text{VSAR}}$ denotes the set of features of VSAR. Intuitively, if the projection of the VSAR features to $[\![VS_{full}]\!]$ is equivalent to the original $[\![\text{VSAR}]\!]$, the constraints map_{SoftPL} has no effect on the VSAR part of VS_{full} and thus the reachability property holds. Otherwise some specifications (i.e., configurations of VSAR) cannot be reached. The property of Equation 1 can then be implemented with the combined use of **slice**, **aggregate** and **compare** operators. More precisely, Equation 1 implies to check if VSAR is a *refactoring* of $\Pi_{\mathcal{F}_{\text{VSAR}}} (VS_{full})$. The checking can be realized with a few lines of FAMILIAR (see Listing 21).

```
// reachability property
VSFull = aggregate { VSARfm PFCfm } withMapping prules
VSARfm2 = slice VSFull including VSFull.VSApplicationRequirement*
if (compare VSARfm2 VSARfm eq SPECIALIZATION) then
      println "For some specifications, no software configuration..."
      // differencing techniques
      // ...
else
       println "Reachability property holds."
end
```

Listing 21: Checking reachability: for all valid configurations of VSARfm, there exists at least one valid configuration in PFCfm.

A brute force strategy which consists in enumerating all possible specifications and then checking the existence of a software configuration would be clearly inappropriate, especially in our case where we have more than 10^8 valid specifications and more than 10^6 software configurations. The combined use of **slice** and **compare** are a much more scalable technique. As exposed in Section 6, **slice** and **compare** operate over a propositional formula and can scale for very large FMs. Without these capabilities, this kind of reasoning would not be made possible for the order of complexity encountered in this case study.

Step-wise Specialization and Choices Propagation. In line with specific requirements and deployment scenarios, the video surveillance expert step-wise *specialized* VSAR by removing some features, by modifying some feature groups, etc. After the specialization of VSAR, automated techniques were used to update the software platform FM. To do so, the specialized VSAR FM and the software platform FM together with rules were aggregated. Finally, we used the **slice** operator on the aggregated FM, only including the set of features related to the software platform.

We observed that from a specification of a context, the possible configurations in the software platform can be highly reduced. We applied the techniques on six different scenarios: the average number of features to consider in the software platform FM was less than 10^4 (instead of 10^6 configurations). In the six scenarios, we *reused i)* the FMs and the constraints and *ii)* automated procedures to control the specialization process and reduce the variability in the software part.

③ **Management of Product Line Variability and Software Variability.** In SPL engineering, two *concerns* of variability are usually distinguished [2, 18]: software variability, related to technical details and hidden from customers (also called internal variability), as opposed to product line (PL) variability, proposing a set of products that are visible to them (also called external variability). In [18], Metzger et al. proposed a formal and concise approach for *separating* PL variability and software variability and enabling automatic analysis. The two concerns are modeled as two FMs and inter-related by constraints. The authors mention several properties that should be checked when reasoning about the two kinds of variability.

The operators provided in FAMILIAR can be combined to support separation of concerns in this context and to reason about the two kinds of variability. Let fm_{PL} be the FM documenting the PL variability, $fm_{software}$ be the FM documenting the software variability and map_{SoftPL} be the constraints relating the two FMs.

Realized-by Property Checking. 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 for building. In terms of feature modeling, we want to ensure that for each valid selection/deselection of features of fm_{PL} performed by a customer, there exists at least one corresponding software product described by $fm_{software}$. The *realized-by* property is similar to the *reachability* property described above: **aggregate**, **slice**, **compare** and **merge** operators in **diff** and **intersection** modes can be used to automatically check this property. The FAMILIAR scripts developed in the context of video surveillance systems have been reused.

Usefulness Property Checking. Another important property is to determine whether a product is *useful* (i.e., if it is a possible realization of a PL member). As argued in [18], the list of non-useful products is a symptom of unused flexibility of the software platform. The usefulness property can be seen as the "symmetric" of the realized-by property: FAMILIAR parameterized scripts are called and reused.

Evolving the FMs. In some circumstances, the realized-by or/and usefulness property may not hold. It can be on purpose, for example, justified by future marketing extensions. It can also be an unexpected property, (i.e., an error). FAMILIAR scripts are used to assist stakeholders in *detecting* the error and *identifying* the missing configurations in fm_{PL} or/and $fm_{software}$.

For this purpose, the *differencing* operators, formally presented in [34] and integrated (see [63] for more details) into FAMILIAR, are used. Based on differencing information, stakeholders can *correct* fm_{PL}, $fm_{software}$ or the mapping between the two FMs (map_{SoftPL}) and reiterate the process until the expected properties hold. The corrective instructions consist in reusing FAMILIAR facilities to edit FMs while checking operations can be performed afterwards. In this case, the FAMILIAR language offers a comprehensive solution to manage PL and software variability, including their specifications, their checking and their evolutions.

On Scalability. We revisited the approach defended in [18]. We applied the techniques using the larger example described in the reference. We successfully retrieved the same results, but our approach is more efficient since we do not enumerate configurations/products. Hence, for larger FMs, the use of our proposed techniques are mandatory since without them the checking of some important properties (e.g., realizability) would not be possible.

④ **Reverse Engineering Architectural Feature Models.** FAMILIAR was evaluated when applied to FraSCAti [75], a large and highly configurable component and plugin-based system. The goal was to reverse engineer a variability model of the FraSCAti architecture, i.e., obtain a software architecture FM.

The idea was that the software architecture FM, noted $FM_{Arch_{150}}$, originally produced by an extraction procedure, represents only an over approximation in terms of sets of valid configurations. Hence several sources of information, namely software architecture ($FM_{Arch_{150}}$), plugin dependencies (FM_{Plug}) and the correspondences between software

elements and plugins, were *combined* using the **aggregate** operator. Intuitively, the presence of constraints in the aggregated FM reduces the legal combinations of $FM_{Arch_{150}}$'s features. This is the role of the **slice** operator to compute the projection corresponding to the software architecture part of the aggregated FM. Figure 8 illustrates the process using a simplified example. The FAMILIAR script presented in Listing 22 implements a solution.

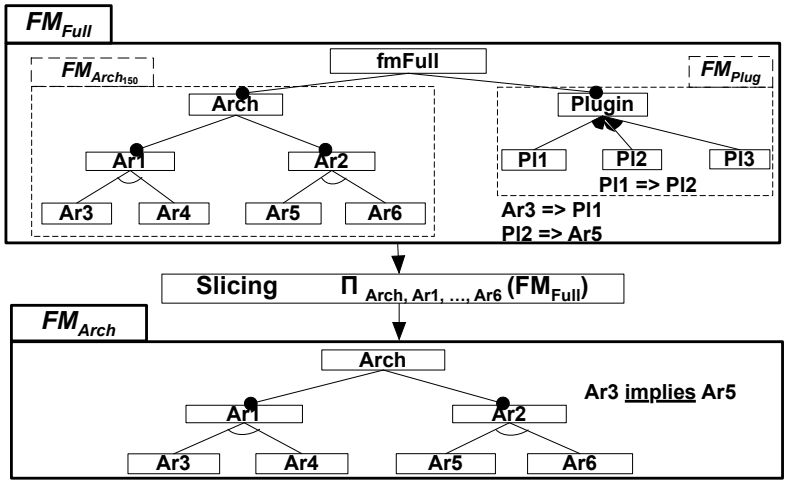


Figure 8: Enforcing an FM using **aggregate** and **slice**

```
1 fmArch150 = FM ( Arch : Ar1 Ar2; Ar1: (Ar3|Ar4); Ar2 : (Ar5|Ar6); )
2 fmPlugin = FM ( Plugin : (Pl1|Pl2|Pl3)+ ; Pl1 -> Pl2 ; )
3 fmFull = aggregate { fmArch150 fmPlugin } withMapping
4                    constraints (Ar3 -> Pl1 ; Pl2 -> Ar5; )
5 fmArch = slice fmFull including fmArch150.* // enforced FM
```

Listing 22: Combining FMs to restrict the configuration space of fmArch150

The aggregated FM resulting from the combination of different variability sources and the bidirectional mapping contains 92 features and 158 cross-tree constraints. The slicing technique significantly reduced the over approximation of the original architectural FM (from $\approx 10^{11}$ to $\approx 10^{6}$) so that we obtained a more accurate variability representation of the architecture.

Step-wise Refinement. The architectural FM resulting from the automatic extraction was compared with another architectural FM, this time manually designed by the software architect (SA) of FraSCAti. Unfortunately, the direct comparison yielded to unexploitable results, mainly due to the difference of *granularity* (i.e., some features in one FM are not present in the other). Basic manual edits of FMs were unpractical and the **slice** operator was extensively applied as we needed to safely remove features involved in several constraints or that were in the middle of the hierarchy. Once the two FMs have been reconciled, they were compared and differences were identified (e.g., using the **merge** operator in diff mode or the **compare** operator). This encouraged the SA to correct his initial model and the architectural FM produced by the automatic procedure [76]. FAMILIAR is now used to manage the development and releases of FraSCAti (see `http://frascati.ow2.org/doc/1.4/ch12.html`).

⑤ **From Product Descriptions to Feature Models.** The task of manually building FMs may prove very arduous for practitionners, especially when they have to collect and model them from such inputs as unstructured product descriptions, tabular data or product matrices [77, 78, 79, 80, 81]. In [82] we proposed to semi-automatically extract FMs from several tabular data files documenting a set of products. First, practitionners can specify directives (variability interpretation, hierarchical representation, etc.) to build as much FMs as there are products. Second, the FMs of

the products are merged, producing a new FM that compactly represents valid combinations of features supported by the set of products. It is similar to the building of FM services' repository (see Section 5). To realize the extraction process, FAMILIAR instructions are generated:

- **insert**, **aggregate** and editing operators are used to build FMs corresponding to products ;
- **merge** in **union** mode is applied.

We applied the extraction procedure on public data, involving hundreds of features per FMs and dozens of FMs [82]. FAMILIAR plays a key role and fully automates the synthesis of an FM from the composition of smaller ones.

7.2. Discussions

We can observe that FAMILIAR has been used in different application domains (medical imaging, video surveillance), for different purposes (scientific workflow design, variability modeling from requirements to runtime, reverse engineering FMs) and by different kinds of stakeholders (e.g., domain experts, software engineer). In the different case studies, the main benefits are an adequate and scalable support for separation of concerns and automated reasoning – two important properties identified in Section 3.

Separation of Concerns (SoC). In the five case studies, support for SoC is clearly needed and can be justified by two phenomena. Firstly, the FMs may be originally separated: It is the case when one describes the variability of services that are *by nature* modular entities or when independent suppliers describe the variability of their different products (see case study ①). It is also the case when two FMs need to be integrated or reconciled (see case study ④). Finally, in case study ⑤, several FMs are grouped together. As a result, the composition operators **merge** and **aggregate** were extensively used in all case studies.

Secondly, it can be the intention of an SPL practitioner to modularize the variability description of the system according to different criteria or concerns. It is the case when separating the requirements variability from the software variability (see case study ②), when PL variability is distinguished from software variability (see case study ③): Stakeholders can focus on their know-how while co-evolution and maintenance of the two FMs are facilitated. In these cases, the decomposition operator (**slice**) and differencing techniques (e.g., **compare**, **merge** in **diff** mode) were required.

Furthermore, the modular mechanisms of FAMILIAR allow an FM user to *reuse* FMs and reasoning procedures. In the case study ②, VSAR and PFC have been reused in the six deployment scenarios as well as parameterized scripts to control the realizability property. In the case study ①, the catalog of FMs has been reused to configure different workflows. In the case study ③, the PL and software variability can be *incrementally* managed: edits to the two FMs are applied while reasoning operations can be *repeated*.

Automated Reasoning. For most of the management activities used in the case studies, a manual intervention is both error-prone, labor-intensive and time-consuming. In practice, there are hundreds of features whose legal combinations are governed by many and often complex rules. This complexity has already been observed on a scale of one FM in the literature, but it becomes even more complex with multiple FMs, like in the case studies. It is thus of crucial importance to be able to *automate* the decision making process as much as possible. Another important aspect is the ability of techniques to *reason* about FMs. As previously shown, FAMILIAR does provide a set of automated reasoning techniques (operators) that can scale for large FMs and a large number of constraints. In particular, the FMs involved in the case studies can be efficiently handled.

In all case studies, large-scale management is needed. FM users have to combine numerous operators, perform sequences of composition and decomposition on several FMs, while reasoning about intermediate results. The FAMILIAR language brings such capabilities to the FM users. Without these new capabilities, some analysis and reasoning operations would not be made possible, for example, in the case study ② where an enumerative technique is not adequate.

8. Conclusion

As variability modeling is a key activity in a growing number of software engineering contexts, techniques and support to manage the complexity of larger variability models (i.e., feature models) are needed. In this article, we

introduced FAMILIAR, a textual and executable domain-specific language that provides a practical support for managing feature models. We gave an overview of syntactic facilities, modular mechanisms, as well as operators provided in FAMILIAR so that feature model users can import, export, edit, configure, compose, decompose, configure and reason about feature models. We illustrated how a repository of medical imaging services can be practically managed using both reusable scripts and automated reasoning techniques provided by FAMILIAR. We also reported on various applications of FAMILIAR in different domains (medical imaging, video surveillance) and for different purposes (scientific workflow design, variability modeling from requirements to runtime, reverse engineering), demonstrating its applicability. The FAMILIAR constructs enable feature model users to realize complex variability management tasks as needed in the different case studies. Besides some of the FAMILIAR capabilities turn out to be mandatory at a large scale since without the language some analysis and reasoning operations would not be made possible in the different applications.

As future work, we plan to evaluate the FAMILIAR language from a user perspective. The different qualities of a domain-specific language, such as learnability, expressiveness, and usability, are likely to be considered. We hope to conduct the experiments in various settings, i.e., in industrial and academic contexts, in different application domains, and with various kinds of participants having different degrees of expertise. We will also investigate how FAMILIAR can be combined with other tools to manage both feature models and associated artefacts (e.g., source code, design models).

Acknowledgments

Dr. Acher's work was partially funded by the FP7 Marie-Curie COFUND program. The authors thank Foudil Bendjabeur and Aleksandar Jakšić for their work with FAMILIAR.

References

[1] M. Svahnberg, J. van Gurp, J. Bosch, A taxonomy of variability realization techniques: Research articles, Software Practice and Experience 35 (8) (2005) 705–754.
[2] K. Pohl, G. Böckle, F. J. van der Linden, Software Product Line Engineering: Foundations, Principles and Techniques, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.
[3] P. Clements, L. M. Northrop, Software Product Lines : Practices and Patterns, Addison-Wesley Professional, 2001.
[4] L. Chen, M. A. Babar, A systematic review of evaluation of variability management approaches in software product lines, Information & Software Technology 53 (4) (2011) 344–362.
[5] A. Classen, P. Heymans, P.-Y. Schobbens, What's in a feature: a requirements engineering perspective, in: 11th International Conference on Fundamental approaches to Software Engineering (FASE'08), Vol. 4961 of LNCS, 2008, pp. 16–30.
[6] S. Apel, C. Kästner, An overview of feature-oriented software development, Journal of Object Technology (JOT) 8 (5) (2009) 49–84.
[7] K. Kang, S. Cohen, J. Hess, W. Novak, S. Peterson, Feature-Oriented Domain Analysis (FODA), Tech. Rep. CMU/SEI-90-TR-21, SEI (Nov. 1990).
[8] K. Kang, S. Kim, J. Lee, K. Kim, E. Shin, M. Huh, FORM: a feature-oriented reuse method with domain-specific reference architectures, Annals of Software Engineering 5 (1) (1998) 143–168.
[9] D. S. Batory, Feature models, grammars, and propositional formulas, in: 9th International Software Product Line Conference (SPLC'05), Vol. 3714 of LNCS, 2005, pp. 7–20.
[10] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, Y. Bontemps, Generic semantics of feature diagrams, Computer Networks 51 (2) (2007) 456–479.
[11] K. Czarnecki, A. Wąsowski, Feature diagrams and logics: There and back again, in: 11th International Software Product Line Conference (SPLC'07), IEEE, 2007, pp. 23–34.
[12] T. Thüm, C. Kästner, S. Erdweg, N. Siegmund, Abstract features in feature modeling, in: 15th International Software Product Line Conference (SPLC'11), IEEE, 2011, pp. 191–200.
[13] D. Benavides, S. Segura, A. Ruiz-Cortés, Automated analysis of feature models 20 years later: A literature review, Information Systems 35 (2010) 615–636.
[14] Common Variability Language (CVL) standard, `http://www.omgwiki.org/variability/doku.php`.
[15] M.-O. Reiser, M. Weber, Multi-level feature trees: A pragmatic approach to managing highly complex product families, Requir. Eng. 12 (2) (2007) 57–75.
[16] M. Mendonca, D. Cowan, Decision-making coordination and efficient reasoning techniques for feature-based configuration, Science of Computer Programming 75 (5) (2010) 311 – 332.
[17] S. She, R. Lotufo, T. Berger, A. Wasowski, K. Czarnecki, Reverse engineering feature models, in: 33rd International Conference on Software engineering (ICSE'11), ACM, 2011, pp. 461–470.
[18] A. Metzger, K. Pohl, P. Heymans, P.-Y. Schobbens, G. Saval, Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis, in: 15th IEEE International Conference on Requirements Engineering (RE '07), 2007, pp. 243–253.

[19] D. Dhungana, P. Grünbacher, R. Rabiser, T. Neumayer, Structuring the modeling space and supporting evolution in software product line engineering, Journal of Systems and Software 83 (7) (2010) 1108–1122.
[20] L. A. Zaid, F. Kleinermann, O. D. Troyer, Feature assembly: A new feature modeling technique, in: J. Parsons, M. Saeki, P. Shoval, C. C. Woo, Y. Wand (Eds.), ER, Vol. 6412 of Lecture Notes in Computer Science, Springer, 2010, pp. 233–246.
[21] D. Dhungana, D. Seichter, G. Botterweck, R. Rabiser, P. Gruenbacher, D. Benavides, J. A. Galindo, Configuration of multi product lines by bridging heterogeneous variability modeling approaches, in: 15th International Software Product Line Conference (SPLC'11), IEEE, 2011, pp. 120–129.
[22] A. Hubaux, T. T. Tun, P. Heymans, Separation of concerns in feature diagram languages: A systematic survey, ACM Computing Surveys (to appear).
[23] H. Hartmann, T. Trew, Using feature diagrams with context variability to model multiple product lines for software supply chains, in: 12th International Software Product Line Conference (SPLC'08), IEEE, 2008, pp. 12–21.
[24] H. Hartmann, T. Trew, A. Matsinger, Supplier independent feature modelling, in: 13th International Software Product Line Conference (SPLC'09), IEEE, 2009, pp. 191–200.
[25] T. van der Storm, Variability and component composition, in: International Conference on Software Reuse (ICSR'04), Vol. 3107 of LNCS, Springer, 2004, pp. 157–166.
[26] J. Bosch, Toward compositional software product lines, IEEE Software 27 (2010) 29–34.
[27] T. T. Tun, Q. Boucher, A. Classen, A. Hubaux, P. Heymans, Relating requirements and feature configurations: A systematic approach, in: 13th International Software Product Line Conference (SPLC'09), IEEE, 2009, pp. 201–210.
[28] K. Czarnecki, Overview of generative software development, in: International Workshop on Unconventional Programming Paradigms (UPP'04), Vol. 3566 of Lecture Notes in Computer Science, Springer, 2005, pp. 326–341.
[29] A. Hubaux, P. Heymans, P.-Y. Schobbens, D. Deridder, E. K. Abbasi, Supporting multiple perspectives in feature-based configuration, Software and Systems Modeling (2011) 1–23.
[30] M. Acher, P. Collet, P. Lahire, R. France, Separation of concerns in feature modeling: Support and applications, in: 11th International Conference on Aspect-oriented Software Development (AOSD'12), ACM, 2012, pp. 1–12.
[31] M. Acher, P. Collet, P. Lahire, R. France, Composing feature models, in: 2nd International Conference on Software Language Engineering (SLE'09), LNCS, Springer, 2009, pp. 62–81.
[32] M. Acher, P. Collet, P. Lahire, R. France, Comparing approaches to implement feature model composition, in: 6th European Conference on Modelling Foundations and Applications (ECMFA'10), Vol. 6138 of LNCS, Springer, 2010, pp. 3–19.
[33] M. Acher, P. Collet, P. Lahire, R. France, Slicing feature models, in: 26th IEEE/ACM International Conference on Automated Software Engineering (ASE'11), IEEE, 2011, pp. 424–427.
[34] M. Acher, P. Heymans, P. Collet, C. Quinton, P. Lahire, P. Merle, Feature Model Differences, in: 24th International Conference on Advanced Information Systems Engineering (CAiSE'12), LNCS, Springer, 2012, p. , to appear.
[35] M. Acher, P. Collet, P. Lahire, R. France, A Domain-Specific Language for Managing Feature Models, in: 26th International Symposium on Applied Computing (SAC'11), Programming Languages Track, ACM, Taiwan, 2011, pp. 1333–1340.
[36] FAMILIAR: FeAture Model scrIpt Language for manIpulation and Automatic Reasonning, `http://nyx.unice.fr/projects/familiar/` (2010).
[37] M. Mendonca, A. Wąsowski, K. Czarnecki, SAT-based analysis of feature models is easy, in: 13th International Software Product Line Conference (SPLC'09), IEEE, 2009, pp. 231–241.
[38] M. Acher, P. Collet, P. Lahire, S. Moisan, J.-P. Rigault, Modeling Variability from Requirements to Runtime, in: 16th International Conference on Engineering of Complex Computer Systems (ICECCS'11), IEEE, 2011, pp. 77–86.
[39] M. Acher, P. Collet, A. Gaignard, P. Lahire, J. Montagnat, R. France, Composing Multiple Variability Artifacts to Assemble Coherent Workflows, Software Quality Journal Special issue on Quality Engineering for Software Product Lines 20 (3-4) (2012) 689–734.
[40] K. Czarnecki, S. Helsen, U. Eisenecker, Staged Configuration through Specialization and Multilevel Configuration of Feature Models, Software Process: Improvement and Practice 10 (2) (2005) 143–169.
[41] pure::variants, `http://www.pure-systems.com/pure_variants.49.0.html` (2006).
[42] C. W. Krueger, The biglever software gears unified software product line engineering framework, in: 12th International Software Product Line Conference (SPLC'08), IEEE Computer Society, 2008, p. 353.
[43] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, T. Leich, FeatureIDE: An extensible framework for feature-oriented software development, Science of Computer Programming (to appear).
[44] M. Mendonca, M. Branco, D. Cowan, S.P.L.O.T.: software product lines online tools, in: Proceeding of the 24th International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'09 companion papers), ACM, 2009, pp. 761–762.
[45] P. Trinidad, D. Benavides, A. Ruiz-Cortés, S. Segura, A. Jimenez, FAMA framework, in: 12th International Software Product Line Conference (SPLC'08), IEEE, 2008, pp. 359–.
[46] G. Botterweck, M. Janota, D. Schneeweiss, A design of a configurable feature model configurator, in: D. Benavides, A. Metzger, U. W. Eisenecker (Eds.), VaMoS, Vol. 29 of ICB Research Report, Universität Duisburg-Essen, 2009, pp. 165–168.
[47] T. Thüm, D. Batory, C. Kästner, Reasoning about edits to feature models, in: 31st International Conference on Software Engineering (ICSE'09), ACM, 2009, pp. 254–264.
[48] M. Fowler, Domain Specific Languages, Addison-Wesley Professional, 2010.
[49] A. van Deursen, P. Klint, Little languages: little maintenance, Journal of Software Maintenance 10 (1998) 75–92.
[50] M. Mernik, J. Heering, A. M. Sloane, When and how to develop domain-specific languages, ACM Computer Surveys 37 (4) (2005) 316–344.
[51] T. Kosar, M. Mernik, J. C. Carver, Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments, Empirical Software Engineering 17 (3) (2012) 276–304.
[52] K. Czarnecki, S. Helsen, U. Eisenecker, Formalizing Cardinality-based Feature Models and their Specialization, in: Software Process Improvement and Practice, 2005, pp. 7–29.
[53] R. Michel, A. Classen, A. Hubaux, Q. Boucher, A formal semantics for feature cardinalities in feature diagrams, in: Fifth International

Workshop on Variability Modelling of Software-intensive Systems (VaMoS'11), 2011, pp. 82–89.
[54] T. Ziadi, J.-M. Jézéquel, Product Line Engineering with the UML: Deriving Products, no. 978-3-540-33252-7 in Software Product Lines: Reasearch Issues in Engineering and Management, Springer Verlag, 2006, Ch. 15, pp. 557–586.
[55] M. Voelter, I. Groher, Product line implementation using aspect-oriented and model-driven software development, in: 11th International Software Product Line Conference (SPLC'07), 2007, pp. 233–242.
[56] K. Czarnecki, M. Antkiewicz, Mapping features to models: A template approach based on superimposed variants, in: GPCE'05, Vol. 3676 of LNCS, 2005, pp. 422–437.
[57] R. E. Lopez-Herrejon, A. Egyed, Detecting inconsistencies in multi-view models with variability, in: 6th European Conference on Modelling Foundations and Applications (ECMFA'10), 2010, pp. 217–232.
[58] F. Heidenreich, P. Sanchez, J. Santos, S. Zschaler, M. Alferez, J. Araujo, L. Fuentes, U. K. amd Ana Moreira, A. Rashid, Relating feature models to other models of a software product line: A comparative study of featuremapper and vml*, Transactions on Aspect-Oriented Software Development VII, Special Issue on A Common Case Study for Aspect-Oriented Modeling 6210 (2010) 69–114.
[59] A. Classen, Q. Boucher, P. Heymans, A text-based approach to feature modelling: Syntax and semantics of TVL, Science of Computer Programming, Special Issue on Software Evolution 76 (12) (2011) 1130–1143.
[60] M. Rosenmüller, N. Siegmund, T. Thüm, G. Saake, Multi-Dimensional Variability Modeling, in: Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS), ACM Press, 2011, pp. 11–20.
[61] Y. Sun, Z. Demirezen, M. Mernik, J. Gray, B. Bryant, Is my DSL a modeling or programming language?, in: Proceedings of 2nd International Workshop on Domain-Specific Program Development (DSPD), 2008, pp. 1–4.
[62] D. Watt, Programming Language Concepts and Paradigms, Prentice-Hall, 1990.
[63] Differencing techniques for feature models in familiar: `https://nyx.unice.fr/projects/familiar/wiki/DiffFMs`.
[64] M. Weiser, Program slicing, in: Proceedings of the 5th International Conference on Software engineering, ICSE '81, IEEE Press, Piscataway, NJ, USA, 1981, pp. 439–449.
[65] P. Trinidad, D. Benavides, A. Durán, A. Ruiz-Cortés, M. Toro, Automated error analysis for the agilization of feature modeling, Journal of Systems and Software 81 (6) (2008) 883–896.
[66] R. van Ommering, Building product populations with software components, in: 22nd International Conference on Software Engineering (ICSE'02), ACM, 2002, pp. 255–265.
[67] Xtext, `http://www.eclipse.org/Xtext/` (2009/2011).
[68] F. Cao, B. R. Bryant, C. C. Burt, Z. Huang, R. R. Raje, A. M. Olson, M. Auguston, Automating feature-oriented domain analysis, in: B. Al-Ani, H. R. Arabnia, Y. Mun (Eds.), Software Engineering Research and Practice, CSREA Press, 2003, pp. 944–949.
[69] S^2T^2 configurator, `http://download.lero.ie/spl/s2t2/`.
[70] A. Pleuss, G. Botterweck, D. Dhungana, Integrating automated product derivation and individual user interface design, in: VaMoS'10, ACM, 2010, pp. 69–76.
[71] N. Andersen, K. Czarnecki, S. She, A. Wasowski, Efficient synthesis of feature models, in: ACM (Ed.), Proceedings of 16th International Software Product Line Conference (SPLC'12), 2012, pp. 106–115.
[72] R. Pohl, K. Lauenroth, K. Pohl, A performance comparison of contemporary algorithmic approaches for automated analysis operations on feature models, in: P. Alexander, C. S. Pasareanu, J. G. Hosking (Eds.), ASE, IEEE, 2011, pp. 313–322.
[73] M. Mendonca, A. Wąsowski, K. Czarnecki, D. Cowan, Efficient compilation techniques for large scale feature models, in: 7th International Conference on Generative programming and component engineering (GPCE'08), ACM, 2008, pp. 13–22.
[74] D. Benavides, S. Segura, P. Trinidad, A. Ruiz–Cortés, A first step towards a framework for the automated analysis of feature models, in: Managing Variability for Software Product Lines: Working With Variability Mechanisms, 2006, pp. 1–5.
[75] L. Seinturier, P. Merle, R. Rouvoy, D. Romero, V. Schiavoni, J.-B. Stefani, A component-based middleware platform for reconfigurable service-oriented architectures, Software: Practice and Experience 42 (5) (2012) 559–583.
[76] M. Acher, A. Cleve, P. Collet, P. Merle, L. Duchien, P. Lahire, Reverse engineering architectural feature models, in: 5th European Conference on Software Architecture (ECSA'11), LNCS, Springer, 2011, pp. 220–235.
[77] A. Lora-Michiels, C. Salinesi, R. Mazo, A method based on association rules to construct product line models, in: International Workshop on Variability Modeling of Software Intensive Systems (VAMoS'10), Vol. 37 of ICB-Research Report, 2010, pp. 147–150.
[78] E. N. Haslinger, R. E. Lopez-Herrejon, A. Egyed, Reverse engineering feature models from programs' feature sets, in: Working Conference on Reverse Engineering (WCRE'11), IEEE CS, 2011, pp. 308–312.
[79] V. Alves, C. Schwanninger, L. Barbosa, A. Rashid, P. Sawyer, P. Rayson, C. Pohl, A. Rummler, An exploratory study of information retrieval techniques in domain analysis, in: 12th International Software Product Line Conference (SPLC'08), IEEE, 2008, pp. 67–76.
[80] U. Ryssel, J. Ploennigs, K. Kabitzsch, Extraction of feature models from formal contexts, in: I. Schaefer, I. John, K. Schmid (Eds.), SPLC Workshops, ACM, 2011, p. 4.
[81] H. Dumitru, M. Gibiec, N. Hariri, J. Cleland-Huang, B. Mobasher, C. Castro-Herrera, M. Mirakhorli, On-demand feature recommendations derived from mining public product descriptions, in: International Conference on Software Engineering (ICSE'11), ACM, 2011, pp. 181–190.
[82] M. Acher, A. Cleve, G. Perrouin, P. Heymans, C. Vanbeneden, P. Collet, P. Lahire, On extracting feature models from product descriptions, in: International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'12), ACM, 2012, pp. 45–54.