A Domain-Specific Language for Managing Feature Models

Mathieu Acher
acher@i3s.unice.fr

Philippe Collet
collet@i3s.unice.fr

Philippe Lahire
lahire@i3s.unice.fr

University of Nice Sophia Antipolis
I3S Laboratory (CNRS UMR 6070)

Robert B. France
france@cs.colostate.edu
Colorado State University
Computer Science
Department

ABSTRACT

Feature models are a popular formalism 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. To manage complexity, there is a need to separate, relate and compose several feature models while automating the reasoning on their compositions in order to enable rigorous SPL validation and configuration. In this paper, we propose a Domain-Specific Language (DSL) that is dedicated to the management of feature models and that complements existing tool support. Rationale for this language is discussed and its main constructs are presented through examples. We show how the DSL can be used to realize a non trivial scenario in which multiple SPLs are managed.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Constructs and Features

Keywords

Domain-Specific Language, Feature Models, Product Lines

1. INTRODUCTION

A software product line (SPL) is a family of software products that are built in a prescribed manner from a common set of core development assets [1]. In model-based SPL engineering, feature models (FMs) [2, 3, 4] are often used to compactly represent product commonalities and variabilities in terms of optional, alternative and mandatory features. FMs can be used to describe software features at various levels of abstraction and thus can be used throughout the software lifecycle to capture commonalities and variabilities in artifacts that are produced in different development phases [1,5,6]. SPLs are becoming increasingly complex. For example, SPLs for systems of systems and for service assemblies that span multiple organizations often require feature models consisting of thousands of features [7, 8, 9, 10, 11]. Building, manipulating, and evolving these large FMs using current technologies is challenging and error-prone [12, 13]. There is a need for more compositional approaches to managing large, complex FMs.

In order to improve scalability, researchers are developing approaches that allow developers to build complex FMs from smaller FMs and to rigorously reason about FM properties as FMs evolve. In previous work [14, 15], we developed FM *composition* operators (insertion, merge) that complement common atomic modifications of FMs. Their semantics is defined in terms of properties on configuration sets characterized by FMs. To improve scalability it is not enough to provide composition and reasoning mechanisms; there is also a need to provide SPL developers with the means to control when and how composition and analysis mechanisms are applied, and with the means to replay and reuse composition and analysis procedures.

In this paper, we present a language that is specifically designed to support FM manipulations. This domain-specific language (DSL), named FAMILIAR (for FeAture Model scrIpt Language for manIpulation and Automatic Reasoning [16]), can be used together with FM editors and reasoning tools to support large scale management of FMs. The proposed DSL notably enables developers to manipulate FMs and their configuration sets as variables. Basic access functions to FM elements are complemented with composition and reasoning operators. Dedicated conditional and loop constructs are provided, as well as modularization through executable and reusable scripts with scope handling. In the next section, background on FMs is given and rationale for the DSL is discussed. In Section 3, we then present the main capabilities of the language through a small example. Section 4 describes an application of the DSL on a larger scale problem that involves manipulating multiple SPLs with different scripts. Section 5 discusses the adequacy of the language as well as future work.

2. BACKGROUND AND MOTIVATION

2.1 Feature Models

Feature models (FMs) hierarchically structure application features into multiple levels of increasing detail. When decomposing a feature into subfeatures, the subfeatures may be optional or mandatory or may form *Alternative-*, *Or-*, or *And*-groups.

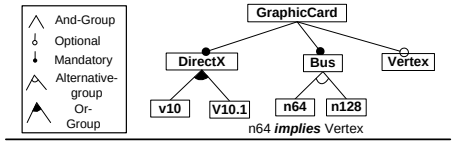

Figure 1: Modeling a graphic card family

An FM can be described graphically or textually (see Fig-

ure 1) and defines a set of valid feature *configurations*. A valid configuration is obtained by selecting features in a manner that respects the following rules: *i)* If a feature is selected, its parent must also be selected; *ii)* If a parent is selected, the following features must also be selected - all the mandatory subfeatures in its And-group, exactly one subfeature in each of its Alternative-groups, and at least one of its subfeatures in each of its Or-groups; *iii)* Constraints relating features in different subtrees must hold ; *iv)* Additional constraints represented as propositional formulas must hold. A configuration is defined as a set of selected features, for example, {GraphicCard, DirectX, v10, v10.1, Bus, n128} is a valid configuration of the FM shown in Figure 1. An SPL is a set of products where each product corresponds to a valid configuration of an FM.

2.2 On Managing Feature Models

Our experience in the development of a real-world Video Surveillance Systems SPL [17] and a Medical Imaging Workflow SPL [18] provides evidence that support for separating and composing concerns while developing FMs can significantly improve management of system complexity. The composition operators we proposed in previous work [14] have been implemented [15], but the large scale handling of multiple SPLs (e.g., SPLs for systems of systems) requires managing sequences of different manipulations, for example, evolving FM structures and their configurations, inserting new features, extracting sub-FMs while maintaining constraints, and reasoning about intermediate results during composition. There is thus a need for scalable FM development environments that allow developers to better control how large FMs are developed and evolved. Scalable approaches to managing FMs should support, at least, the two principles discussed in the following paragraphs.

Separation and composition of concerns in FMs. As mentioned earlier, FMs are becoming increasingly complex. A contributing factor is that FMs are being used not only to describe variability in software designs, but also variability in wider system contexts [9,19]. To manage the complexity of building FMs with a large number of features that are related in a variety of ways, developers need tools that help them (1) separate concerns so that they can be understood and analyzed independently of other concerns, and (2) build large FMs from smaller FMs. Support for separation of concerns also makes it possible to reuse feature structures across different product lines. For example, the reuse of software components between different product lines in the consumer electronics domain is commonplace [9]. System variability is driven by several different dimensions, for example, different product types and different 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 [10]. The need to support multiple SPLs (also called product populations) makes developing SPLs challenging [1]. Separating concerns in FMs can also help manage decisions made by different stakeholders [7]. 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 [11]. A number of techniques do provide some support for separating concerns. For example, FORM [2] allows the connection of various layers of feature refinements. Pohl et al. distinguish external variability (visible to customers) from internal variability (hidden from customers) [1]. We are not aware of any approach that addresses all the issues highlighted above.

Rigorous reasoning about FMs. 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 [4]. An FM is defined not only by its structure, but also by its configuration set. Manually analyzing complex FMs is an error-prone and tedious task, and thus there is a need for tools that automate significant aspects of the reasoning process [12]. There are some approaches to systematically analyze (relationships among) FMs. For example, in [13], a classification is proposed to characterize the relationship between two FMs in terms of sets of configuration and an algorithm is designed to compute the kind of relations between two FMs. Automated analysis of FMs focuses on properties of a feature model, for example, checking that an FM contains at least one product, and determining the number of valid configurations characterized by an FM [3,12]. 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.

2.3 Why a Domain-Specific Language?

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

Several graphical FM editors are currently available, and some do provide support for managing some aspects of FM development (pure::variants [20], FeatureIDE [21], SPLOT [22], etc.). For large scale management, *pure::variants* is a commercial tool with good support for binding to other models and for code generation. *FeatureIDE* is an open editor 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 reasoning about FM edits [13] has been integrated. Nevertheless, current tools do not fully support the composition of several separated feature models. A conceivable solution would be to integrate our FM composition operators (insert and merge) as additional functionalities inside a mainstream graphical editor. Our case studies [17,18] indicate that manipulating several FMs with composition operators requires support for replaying sequences of operations, observing properties along its manipulations, and organizing all these actions as reusable parts. We thus identify this as a requirement for a textual, executable language, close to common scripting languages, as it should define basic sequencing of FM operations while providing access to FM internals, reasoning operations and already identified composition operations. This does not avoid the possibility to also provide graphical counterparts built on top of the textual language, as in many other domains (see Section 5).

As editors like FeatureIDE and frameworks such as FAMA [23] 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 imply many repetitive and error-prone actions. An DSL should then allow a stakeholder to more quickly build the scripts they need to manipulate FMs. In addition, such a DSL should be used more easily by FM stakeholders as the learning curve is expected to be more favorable. The facilities provided to the stake-

holders 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. A DSL seems particularly adequate in this case, as it would provide only the necessary expressive power for anticipated FM manipulations.

3. LANGUAGE IN A NUTSHELL

The FAMILIAR DSL is an executable scripting language that supports manipulating and reasoning about FMs. The next subsections will detail and illustrate the main constructs of the language.

3.1 Types and Variables

FAMILIAR is a typed language that supports both complex and primitive types. 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 programming languages.) Complex types are *Feature Model*, *Configuration*, *Feature* or generic *Set* which represents container values. Primitive types include *String* (e.g., feature names are strings), *Boolean*, *Integer* and *Real*. An example script that illustrates typing is given below:

```
1 gc1 = FM ( GraphicCard: DirectX Speed MemoryBus [Multi];
2    DirectX: (v10.1 | v10); Speed: (n800|n1000); MemoryBus: n128;)
3 gc2 = FM ( GraphicCard: DirectX Speed MemoryBus [Multi];
4    DirectX: (v10.1 | v10); Speed: (n800|n1000); MemoryBus: n128;)
5 b1 = gc1 eq gc2 // b1 is true
6 b2 = gc1 == gc2 // b2 is false
7 str = "v10" // str records the value "v10"
8 fmSet = { gc1 gc2 }  // fmSet records a reference to a set
```

Lines **1-4** define two variables of type *Feature Model*: *gc1* and *gc2*. 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). Note that for the variable *str*, the use of content or reference equality would return the same result which corresponds to content equality. Types have accessors for observing the content of a variable. The example below illustrates the use of accessors :

```
1 gc3 = FM ( GraphicCard: DirectX Speed Bus Multi;
2             DirectX: (v11|v10.1) ; Speed: n1000 ; Bus: n256;)
3 f1 = parent v11 // f1 refers to feature named 'DirectX' in gc3
4 f2 = root gc3 // f2 refers to feature named 'GraphicCard' in gc3
5 s1 = name f2 // s1 is a string "GraphicCard"
6 fs = children f1 // set of features named 'v11' and 'v10.1' in gc3
7 nfs = size fs // 2
```

In line **3**, variable *f1* records a reference to feature DirectX (the parent of feature v11). (We consider that the name of a feature is unique within an FM [3, 4] and is used to identify a feature.) In line **4**, variable *f2* contains a reference to the feature GraphicCard (the root of the FM *gc*3). The remaining lines (**5-7**) illustrate operations returning *i)* the name of a feature, *ii)* the set of direct subfeatures of a given feature and *iii)* the cardinality of a set.

3.2 Operations

3.2.1 Modifying FMs

The language provides operations for renaming and removing features in FMs. Renaming is particularly important when composing several FMs where it cannot be guaranteed that the developers will use the same name for the same feature. The following illustrates how features can be renamed:

```
1 oldFeature = parent n256 // 'Bus' feature of gc3
2 newName = strConcat "Memory" (name oldFeature)
3 b1 = renameFeature oldFeature as newName // aligning terms
4 assert (b1 eq true) // assert (b1) is equivalent
```

In lines **1-3**, the feature Bus of FM *gc3* is renamed to MemoryBus by concatenating the string *Memory* with the old name *Bus*. The operation **assert** in line **4** stops the program with an appropriate error message if the renaming is not successful (i.e., *b1* is **false**). Similarly, the operation **removeFeature** takes a feature as an argument and removes the feature and its subfeatures from the FM it belongs to (see Section 3.4). It also returns **true** or **false** depending whether it is successful or not.

3.2.2 Handling FM configurations

The language also allows developers to create FM configurations, and then **select**, **deselect**, or **unselect** a feature. To **select** a feature means that the derived product will include the feature. On the contrary, **deselect** means that it will not be part of it. By default, a feature is unselected so that **unselect** should be used only after **select** or **deselect**; it means that nothing is yet decided: the feature may belong or not to the final product, i.e., some variability remains within the configuration. Each of these configuration manipulation operations returns a boolean value to notify success or failure. An example usage of these operations is given below:

```
1 conf1 = configuration gc1 // create a configuration of gc1
2 b1 = select Multi in conf1 // feature Multi of gc1 is selected
3 b2 = deselect Multi in conf1 // override the previous selection
4 b3 = unselect Multi in conf1 // neither selected nor deselected
```

In line **1**, the operation **configuration** creates and initializes a configuration of the FM *gc1*. Lines **2-4** provide examples of the configuration manipulations.

3.2.3 Reasoning about FMs

FAMILIAR provides several operations to support reasoning about FMs. The script fragment below provides examples of the FM manipulation and reasoning operations:

```
1 conf2 = copy conf1
2 nb = counting gc1 // number of valid configurations: 8
3 b1 = isValid conf1
4 b2 = (selectedF conf1) eq (selectedF conf2) // true
5 cmp = compare gc1 gc2 // refactoring
```

Line **2** computes the number of valid configurations of *gc1*. The **isValid** operation checks whether a configuration is valid (see line **3**) according to its FM, i.e., satisfies the semantics reminded in Section 2.1. The **isValid** operation can also perform on an FM and determines its satisfiability [12], i.e., whether or not there is at least one valid configuration. FAMILIAR also provides an operation that checks whether a configuration is complete, i.e., whether all features have been selected or deselected. In addition, the *Configuration* type provides three accessors that return the set of selected, deselected and unselected features: **selectedF**, **deselectedF** and **unselectedF**. Line **4** checks that the set of selected features in both *conf1* and *conf2* are equal (which is true simply because *conf2* is a copy of *conf1*). The **compare** operation is used 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 [13]: In the following let f and g be FMs, and let $[\![f]\!]$ (resp. $[\![g]\!]$) denote the set of configurations for g; 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 $[\![g]\!] = [\![f]\!]$; otherwise, f is an *arbitrary* edit of g. Line **5** illustrates comparison capabilities based on sets of configurations of FMs.

In addition, FAMILIAR provides *i)* a basic **if then else** conditional construct,*ii)* a **foreach** loop which can be used to iterate over a set of variables (e.g., representing FMs, features, and configurations) to perform a sequence of operations (see Section 4 for an illustration).

3.3 Composition

3.3.1 Inserting FM

The **insert** operator produces an FM by inserting an FM into another base or target FM. The operator takes three arguments: *i)* the FM to be inserted *ii)* the feature in the base/target FM where the insertion is to take place, and *iii)* the operator (e.g., Alternative) that determines the form of the insertion. The insertion fails and returns **false** if the resulting FM is not well-formed, i.e., each feature's name must be unique in an FM [3, 4]. The base FM is modified if the insertion succeeds. An example script fragment describing an insertion is given below:

```
1  base = FM (Keyboard: [ControlCD] Wireless Wiring
2             [International] ; Wiring: (USB|PS2); )
3  aspect1 = FM ( Lang : [US] European [Chinese]; )
4  insert aspect1 into International with mand  // 'base' is modified
5  removeVariable aspect1 // no longer need 'aspect1' variable
6  fInt = parent Lang // feature International is now in 'base' FM
7  assert ((name fInt) eq "International") // check it
8  // check feature Lang in 'base' FM has still three child features
9  assert ((size (children Lang)) eq 3)
```

In the example, the feature Lang is inserted below the feature International (line **4**): Lang is a child feature of International with the mandatory status, i.e., the selection of International implies the selection of Lang. The **assert** operations are performed to check that the insertion produced an FM with correct properties.

3.3.2 Merging FMs

When multiple perspectives or views on a SPL need to be managed or when SPLs are composed with SPLs (e.g., see [18] or Section 4), it is likely that FMs representing SPLs share several features. In this case, the **merge** operator can be used to merge the overlapping parts of the FMs and then to obtain an integrated FM of the system. The merge uses a name-based matching: two features match if and only if they have the same name. Several modes are defined for this operator according to the set of configurations one wants to preserve in the merged FM. The *intersection* mode is the most restrictive option: the merged FM, FM_r, expresses the common valid configurations of FM_1, FM_2, ..., FM_n i.e., a configuration that is valid in FM_1, FM_2, ..., FM_n is also valid in FM_r. The *union* mode is the more permissive option: the merged FM can express either valid configurations of FM_1, FM_2, ..., or FM_n, i.e., a valid configuration of the merged FM, FM_r, is valid *either* in FM_1, FM_2 ... *or* FM_n. A more restrictive property, called *strict union* mode, requires that the set of configurations of FM_r is exactly the union of sets of configurations of FM_1, FM_2, ..., and FM_n. In the *diff* mode, the merge operator takes two input FMs, FM_1 and FM_2, and computes the set-theoretic difference of FM_1 and FM_2. In Figure 2, the properties of the merged FM is summarized with respect to the sets of configurations of two input FMs and the mode.

The variability information associated with features in the merged FM is different according to the merge mode and the properties that one wants to preserve. In addition to the semantics properties defined above, the hierarchy of the merged FM should be as close as possible the hierarchy of input FMs, dead features should not be present in the merged FM, etc. (see [15] for more details).

In FAMILIAR, the merge operators act on (a set of) FMs or configurations and produce FMs with semantics properties according to the mode specified by the programmer. Below is part of a script that uses a merge operator:

```
1  gc4 = FM ( GraphicCard: DirectX Speed Bus [Multi];
2           DirectX: (v10.1|v10) ; Speed: (n800|n1000) ; Bus: n128;)
3  gc5 = FM ( GraphicCard: DirectX Speed Bus [Vertex];
4           DirectX: (v10.1|v10|v9) ; Speed: n800 ; Bus: (n64|n128);)
5  gc_inter = merge intersection gc4 gc5
6  gc_inter_expected = FM ( GraphicCard: DirectX Speed Bus ;
7           DirectX: (v10.1|v10) ; Speed: n800 ; Bus: n128 ;)
8  assert (gc_inter eq gc_inter_expected)
```

In line **5**, the merge operator in intersection mode is applied on *gc4* and *gc5* and produces a new feature model that can be manipulated through the variable *gc_inter*. In line **8**, we check that *gc_inter* is equal to *gc_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.

```
9  gc_sunion = merge sunion gc4 gc5
10 n_sunion = counting gc_sunion // number of valid configurations
11 n_expected = counting gc4 + counting gc5 - counting gc_inter
12 assert (n_sunion eq n_expected)
```

In line **9**, the merge operator in strict union mode is applied on *gc4* and *gc5* and produces a new feature model that can be manipulated through the variable *gc_sunion*. In lines **10-12**, we check the following property: $[\![gc4]\!] \cup [\![gc5]\!] = |[\![gc4]\!]| + |[\![gc5]\!]| - |[\![gc4 \cap gc5]\!]| = |[\![gc\_sunion]\!]|$ using **counting** operations, i.e., the value of n is equal to $[\![gc\_sunion]\!]$.

3.3.3 Relating FMs

Another form of composition can be applied using cross-tree constraints between features so that separated FMs are inter-related. The operator **aggregate** is used for producing a new FM in which a *synthetic* root relates a set of FMs and integrates a set of propositional constraints. All reasoning operations (e.g., **counting**, **isValid**) can be similarly performed on the new FM resulting from the aggregation.

Note that some operations of FAMILIAR, not presented in this paper (**configs**, **deads**, **autoSelect**, **map**, **cleanup**, etc.), are documented online in the reference manual [16].

3.4 Modularization mechanisms

Statements are organized in scripts. 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.

3.4.1 Namespace and Script Calling

Variable name conflicts may occur, for example, when it is necessary to run the same script several times (see discussion on parameterized scripts below) or when features having the same name are used by FMs referred to by different variables. FAMILIAR relies on namespaces to allow disambiguation of variables having the same name. By default, a namespace is attached to each variable of type FM so that it is possible to identify a feature by specifying the name of the variable of type FM followed by "."

```
1  children gc1.DirectX // explicit notation needed
2  gc2.GraphicCard // GraphicCard exists also in gc1 and gc3
3  parent v11 // non ambiguous: equivalent to gc3.v11
```

Lines **1-3** illustrate the use of namespace: features DirectX and GraphicCard are present in two FMs, *gc1* and *gc2*, and are identified thanks to the namespace. Note that v11 appears only in *gc3* and is non-ambiguous so that there is no need to explicitly use the namespace.

Namespaces are also used to logically group related variables of a script, making the development more modular. The example below is used to illustrate how FAMILIAR supports the reuse of existing scripts:

```
1  run "script1" into script_declaration
2  varset = script_declaration.*
3  export varset
4  hide script_declaration.gc*
```

Line **1** shows how to run a script contained in the file *script1* from the current script. The namespace *script_declaration* is an abstract container providing context for all the variables of the script *script1*. In addition, FAMILIAR allows a

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

Figure 2: Merge: semantics properties and notation

script programmer to use a wildcard "*" to access 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 **2** (resp. **4**) accesses the set of all variables of *script_declaration* (resp. all variables starting by *gc* in *script_declaration*). 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.

3.4.2 Parameterized Script

A script can be parameterized using an ordered list of **parameter**s (see lines **2-3** below). A parameter records a variable and, optionally, the type expected. Lines **5-11** implement the replacement of a subtree rooted at the feature *parentF* in the FM *target* by the FM *fmToInsert*.

```
1  // replaceFMbyFM.fml : a parametrized script that replaces a subtree
2  parameter target : FeatureModel
3  parameter fmToInsert : FeatureModel // type specification is optional
4
5  ft = root fmToInsert
6  f = name ft
7  parentF = parent target.f // save the parent of feature 'f'
8  operatorF = operator target.f // save the operator of the feature 'f'
9  assert (removeFeature target.f) // the feature must exist
10 insert fmToInsert into parentF with operatorF
11 hide ft f parentF operatorF // no more need temporary variables
```

The example below illustrates how the parameterized script can be called:

```
1  originalLaptop = FM (Laptop: Motherboard Processor [Wifi]
2                  [GraphicCard] ; GraphicCard: Bus [Multi] ; ... )
3  newGC = FM (GraphicCard: Bus [Vertex]; Bus: (n128|n256);)
4  run "replaceFMbyFM" { originalLaptop newGC }
5  assert (isExisting originalLaptop.Vertex)
```

4. AN APPLICATION TO THE MANAGEMENT OF MULTIPLE SPLS

In this section we illustrate how FAMILIAR can be used to support the management of multiple SPLs. Many examples of SPLs that consist of components that are themselves SPLs can be found in industry. For example, consider an online computer vendor that provides several families of laptops. In order to facilitate the reader understanding, the chosen example is not directly related to a SPL organizing software as the manipulated concepts are supposed to be already known.

Customers use the vendor's online facilities to build and order a laptop by selecting features that best match their requirements. Each laptop family can be configured according to a range of options (e.g. Wifi, Accessories like Keyboard or Mouse) and alternatives (e.g. the choice of a Processor or a GraphicCard). Not all the parts in a laptop configuration are manufactured by the vendor. External suppliers that specialize in the manufacturing of one or several categories of parts (chipsets, processors, graphic cards, etc.) may provide the vendor with needed parts. Each category of parts can be considered to be a product family. Furthermore, several suppliers may produce parts from the same product family.

4.1 First Management of the Multiple SPL

The laptop vendor can benefit from support that combines SPLs from its suppliers in order manage variability in its SPLs. Developing support for building and using multiple SPLs is one of the challenges that the SPL community is tackling [1, 9, 10, 8]. Below we propose a formal definition of multiple SPLs:

DEFINITION 1 (MULTIPLE SPL). *A multiple SPL M_{SPL} is an SPL that manages a set of SPLs $\{SPL_1, SPL_2, \ldots, SPL_n\}$. Its set of products is described by a feature model $FM_{M_{SPL}}$. Any product of a multiple SPL M_{SPL} is a product belonging to either SPL_1, SPL_2, ..., or SPL_n, i.e., any configuration of $FM_{M_{SPL}}$ should correspond to at least one valid configuration of FM_1, FM_2, ..., FM_n. Formally:* $\forall c \in [\![FM_{M_{SPL}}]\!], c \in [\![FM_1]\!] \vee c \in [\![FM_2]\!] \vee \ldots \vee c \in [\![FM_n]\!]$

In a multiple SPL, each constituent SPL describes a different family of products in the same market segment produced by *competing* suppliers.

The example scenario that will be used to illustrate how FAMILIAR can be used to manage multiple SPLs is presented in Figure 3. The scenario involves three steps. In the first step the vendor produces an FM with no assumptions about the parts provided by external suppliers – see ①. In the next step, this family of laptops is viewed as an aggregation of several competing multiple SPLs for different parts of the laptop: for example, Processor, GraphicCard, Keyboard. In the scenario, the need for an external supplier for a graphic card is identified. This requirement results in the generation of multiple SPLs that represents the offerings of the three suppliers for the GraphicCard (see ②). The SPL of each supplier's product family is built from its products. After the requirements of the vendor is mapped with the one of suppliers (see ③), application-specific processing should be performed. The vendor can use multiple SPLs to *i)* determine that the set of suppliers is able to provide the entire set of products and cover all combinations of features or *ii)* identify missing products, and *iii)* eliminate the suppliers that do not provide the required products.

4.1.1 Building SPLs' Repositories

In the script below, $Supplier_1$ proposes eight graphic cards, 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 graphic cards can then be organized by the vendor within an SPL. As, by definition, products exhibit no variability, each graphic card description is represented as an FM in which all features are mandatory.

```
1  // GCSupplier_1.fml: grahic card products specification of Supplier_1
2  GCproduct1 = FM ( GC: Speed Bus DirectX ; Speed: 800 ; ...)
3  GCproduct2 = FM ( GC: Speed Bus DirectX ; Speed: 1200 ; ...)
4  ...
5  GCproduct8 = FM ( GC: Speed Bus DirectX Multi ; Speed: 1200 ; ...)
```

Building an SPL from the set of existing products can be done by applying the merge operator in strict union mode (cf. Section 3.3.2) on the set of corresponding FMs (line **3** in the script below). An FM is then produced for each supplier (e.g., $GCspl\_1$ FM for $Supplier_1$).

```
1  // repositoryGC.fml: repository of graphic cards
2  run "GCSupplier_1" into GCsupp1
```

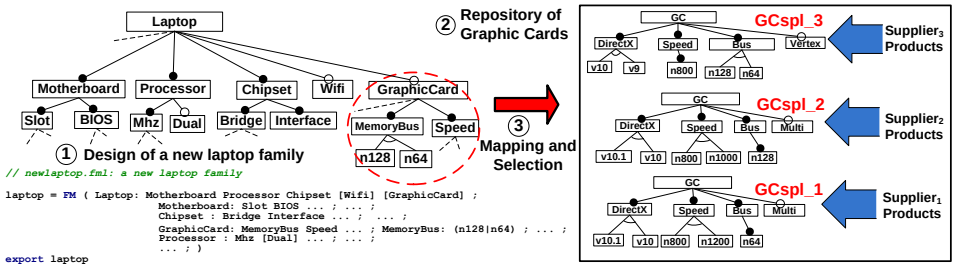

Figure 3: Managing Multiple SPLs

```
3   GCspl_1 = merge sunion GCsupp1.*
4   run "GCSupplier_2" into GCsupp2
5   GCspl_2 = merge sunion GCsupp2.*
6   run "GCSupplier_3" into GCsupp3
7   GCspl_3 = merge sunion GCsupp3.*
8   renameFeature GCspl_3.MemoryBus as "Bus" // aligning terms
9   export GCspl_* // export the three suppliers' FMs/SPLs
```

A repository of graphic card products is now represented by a set of different FMs (one per supplier). Similarly, other repositories can be built for other kinds of products (CPU, monitor, etc.) and imported in a script (see lines **3-5** below).

```
1   // laptopScenario.fml: implementation of suppliers' scenario
2   run "newlaptop" // new family of laptop firstly designed
3   // load repositories
4   run "repositoryGC" into GC
5   run "repositoryCPU" into CPU
6   run "repositoryMonitor" into monitor
```

4.1.2 Mapping Repositories to the SPL

The script below describes mappings from SPLs in repositories to the vendor's SPL. For each part of the family of laptops (handled by the script *newlaptop.fml* – see **run** command line **1** and Figure 3), it is necessary to determine which SPLs and suppliers are suitable to provide the given product. Indeed, the family of laptops offers to customers different alternatives for the choice of a graphic card. In order to reason about the graphic card of FM *laptop*, the variability information of the graphic card (sub-tree rooted at feature GraphicCard) is first extracted. *originalGC*, resulting from the **extract** (line **10**), is a copy of the sub-tree and can be manipulated as an FM. In particular, each valid configuration of *originalGC* should correspond to at least one product provided by a supplier and present in the graphic card repository, i.e., the following relation (C_1) should hold:

$$[\![originalGC]\!] \cap ([\![GCspl\_1]\!] \cup [\![GCspl\_2]\!] \cup [\![GCspl\_3]\!]) = [\![originalGC]\!]$$

It may happen that such a property is not respected and two cases have to be considered. First, the intersection between the set of products of *originalGC* and the set of suppliers' product may be empty (lines **9-16**). Verifying this property can be done by first performing the merge operations on *originalGC*, *GCspl_1*, *GCspl_2* and *GCspl_3* (line **16**). This gives a new FM *gc_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 laptops offers to customers some products that cannot be entirely provided by suppliers (lines **22-27**). In this case, *originalGC* is a *generalization* or an *an arbitrary edit* of the union of set of suppliers' products (line **22**). Performing a **merge diff** operation (see Figure 2) assists users in understanding which set of products is missing (line **23-26**).

```
7   // we map the laptop GC description with the GC repository
8   allProductsGC = merge sunion GC.* // GCspl_1, GCspl_2, GCspl_3
9   originalGC = extract laptop.GraphicCard
10  // alignment: renaming terms to be coherent with the repository
11  renameFeature originalGC.GraphicCard as "GC"
12  renameFeature originalGC.MemoryBus as "Bus" //...
13
14  /***** checking the availability of products *****/
15  gc_merged = merge intersection { originalGC allProductsGC }
16  if (not (isValid gc_merged)) then
17          print "No product can be provided"
18          exit // stop the program
19  end
20  cmp_gc = (compare originalGC allProductsGC)
21  if (cmp_gc eq GENERALIZATION || cmp_gc eq ARBITRARY) then
22    gc_losed = merge diff { originalGC allProductsGC } // missing products
23    s_losed = configs gc_losed // set of configurations of gc_losed
24    n_losed = counting gc_losed // number of configurations
25    println n_losed " product(s) cannot be provided: " s_losed
26  end
27  assert (cmp_gc eq REFACTORING || cmp_gc eq SPECIALIZATION)
```

At this step, *all* products of *originalGC* can be provided by suppliers. For example, the relation holds considering the FMs depicted in Figure 4. Indeed, the set of configurations of *originalGC* is included or equal to the union of set of suppliers' products since, according to set theory, the relation (C_1) is equivalent to $[\![originalGC]\!] \subseteq ([\![GCspl\_1]\!] \cup [\![GCspl\_2]\!] \cup [\![GCspl\_3]\!])$. We check this property in line **27**.

4.1.3 Selecting Suppliers

At this point, the vendor needs to determine which suppliers can provide a subset of the products of *originalGC* (lines **28-40**). Some suppliers cannot provide at least one product corresponding to any configuration of *originalGC* (lines **31-33**) and so should not be considered. Figure 4 illustrates the situation: $Supplier_3$ is no longer available since the intersection between $[\![originalGC]\!]$ and $[\![GCspl\_3]\!]$ is the empty set. Some suppliers offer products that correspond to a valid configuration of *originalGC* but also offer out-of scope products. To remove these products, a merge in intersection mode is systematically performed to restrict attention to the set of relevant supplier products (line **31**). For example, the feature Multi is no longer included in the set of products of $Supplier_1$ and $Supplier_2$ (see $GCspl'\_1$ and $GCspl'\_2$ in Figure 4) while $Supplier_2$ is now able to deliver only one product.

```
28  GC_suppliers_in = setEmpty // create an empty set
29  foreach (supplGC in GC.*) do
30          // checking each supplier providing GCs
31          fmGC_inter = merge intersection { originalGC supplGC }
32          bGCinter = isValid fmGC_inter
33          if (not bGCinter) then
34             println "The supplier is unable ...:\t" supplGC
35          else
36             setAdd GC_suppliers_in fmGC_inter // add relevant FM
37          end
38  end
39  assert ((size GC_suppliers_in) >= 1) // available supplier >= 1
```

At the end of the loop, *GC_suppliers_in* corresponds to a set of FMs, where each FM represents a valuable product line of a supplier. Then, *originalGC* can be configured: a similar sequence of merge operations on configurations can be executed until an unique product of a supplier is chosen.

```
41  gcProduct = configuration originalGC // configuration process
42  select n64 in gcProduct // ...
```

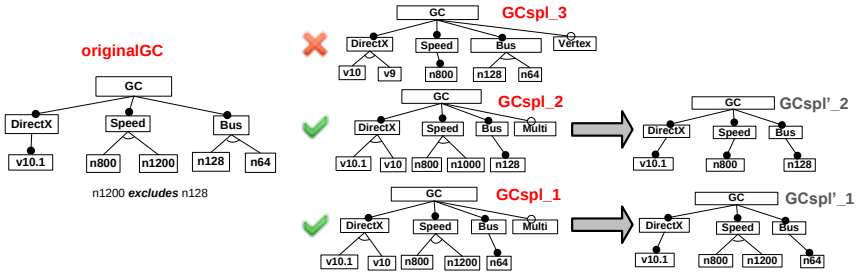

Figure 4: Available suppliers and products.

4.2 Towards Reusable Scripts

The FAMILIAR script *laptopScenario.fml* can fully realize the scenario of Figure 3. Nevertheless, this script has some limitations. First, when users select or deselect features, some suppliers may become unable to provide products 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. Second, reasoning operations are planned to be performed on each part of the laptop (as done with the graphic card). The current script does not allow a programmer to reuse the repetitive tasks performed on FMs, leading again to duplicate codes. We thus propose to apply modularization mechanisms provided by FAMILIAR to raise the limitations mentioned above.

4.2.1 Modularizing the code

The script *laptopScenario2.fml* (see below) is a refactoring of the code *laptopScenario.fml*. The code used in line **1-13** remains the same: repositories of products are built; the laptop FM describing the valid combinations of features is specified; some alignment operations are performed so that one can reason about the laptop FM and the FMs of the repositories. This time, the checking operations are performed in two parameterized scripts: *checkAvailability.fml* and *availableSuppliers.fml* (see below).

```
14 // laptopScenario2.fml: lines 1-13 remain the same
15 /***** checking and inform which available suppliers are *****/
16 run "checkAvailability" { originalGC allProductsGC } into
17 gc_availability
18 if (gc_availability.available) then
19         run "availableSuppliers" { GC.* originalGC } into
20 gc_suppliers
21 end
```

The script *checkAvailability.fml* compares the set of configurations of two FMs, *originalFM* and *allProducts*, which are two parameters of the script. It controls whether or not each valid configuration of *originalFM* is also valid in *allProducts*. It also provides some feedback to users, for example, the number of configurations valid in *originalFM* but invalid in *allProducts*. When the set of configurations of *originalFM* is fully covered by *allProducts*, the boolean variable *available* is set to **true** and exported. Then, this variable is accessible from the calling script, i.e., *laptopScenario2.fml* (see line **18**).

```
1  // checkAvailability.fml : is the set of products fully covered
2  parameter originalFM : FeatureModel
3  parameter allProducts : FeatureModel
4
5  fm_merged = merge intersection { originalFM allProducts }
6  available = true
7  if (not (isValid fm_merged)) then
8          available = false
9          println "No product can be provided: "
10         exit // stop the script execution
11 end
12 cmp = compare originalFM allProducts
13 if (cmp eq GENERALIZATION || cmp eq ARBITRARY) then
14    available = false
15    fm_losed = merge diff { originalFM allProducts } // missing products
16    println (counting fm_losed) " product(s) cannot be provided: " ...
17 end
18 export available
```

In line **17** of *laptopScenario2.fml*, each combination of features of a laptop component (e.g., graphic card) corresponds to at least one product from the repository. We want to determine which specific suppliers are able to offer such products. A parameterized script *availableSuppliers.fml* is used to perform the needed operations. The first parameter, named *suppliers*, is a set of FMs and represents the suppliers' offer: an FM of the set corresponds to the offer of one supplier. The second parameter, named *products*, is an FM representing the product specification. For each FM *suppl* that belongs to the set *suppliers*, we determine whether or not some valid configurations of *products* are valid in *suppl*. If this is not the case, this means the supplier is unable to provide any product and feedback is given to users. At the end, the script displays the available suppliers and the corresponding FMs are updated.

```
1  // availableSuppliers.fml: suppliers able to provide products
2  parameter suppliers : Set // set of feature models
3  parameter products : FeatureModel
4
5  suppliers_in = setEmpty
6  foreach (suppl in suppliers) do
7          // checking suppliers' offer
8          fminter = merge intersection { products suppl }
9          if (not (isValid fminter)) then
10           println "The supplier is unable to provide ...:\t" suppl
11         else
12           setAdd suppliers_in fminter // add relevant FM
13         end
14 end
15 nsuppliers = size suppliers_in
16 println nsuppliers " suppliers are available:"
17 foreach (supp in suppliers_in) do
18         println supp
19 end
```

4.2.2 Resulting benefits

The FAMILIAR script code has been modularized. We can raise the limitations described above. At each step of the configuration process, users can have feedback from available suppliers (even if the set of features is not fully selected or deselected). In addition, for each component of the laptop (CPU, monitors, etc.), similar checking operations can be performed by reusing parameterized scripts.

```
22 /**** laptopScenario2.fml: multi-step configuration ****/
23 gcProduct = configuration originalGC
24 select n64 in gcProduct
25 gcFM = asFM gcProduct // convert a configuration to a FM
26 run "availableSuppliers" GC.* gcFM
27 // configuration continues until a supplier's product is chose
```

5. DISCUSSION AND CONCLUSION

We have presented the main features of FAMILIAR, a DSL to manage FMs, and illustrated it with a non trivial and realistic scenario. We now discuss some salient properties of this language with respect to the requirements identified in Section 2.

Expressiveness and Modularity. Our case study has shown that managing SPLs on a large scale requires a developer to describe and perform complex tasks (aligning terms, checking validity, splitting and composing parts of FM). In a FAMILIAR script, high-level operators (**insert**, **merge**, etc.)

and accessors (**isValid**, **compare**, etc.) can be used to easily manipulate and observe FMs. Manipulable data types are also specific to feature modeling and FM elements (FM, Feature, String) and related information (e.g. Set for configurations) are first-class entities.

When prototyping large scripts, one can manually check at runtime program correctness with the **assert** command. FAMILIAR also supports modularization of scripts, which can be run in other scripts, and even parameterized if need be. In scripts, FMs and features, the uniform handling of namespaces together with information hiding also enable one to develop large and potentially more reusable scripts.

Integration and Complementarity. FAMILIAR [16] is developed in Java language using Xtext [24], a framework for the development of DSLs. We provide an Eclipse text editor and an interpreter that executes the various scripts. The interpreter can be used in an interactive mode. We provide multiple notations for specifying FMs (SPLOT [22], GUIDSL/FeatureIDE [3,21], a subset of TVL [25], etc.) and the notation used in the paper. The support of different formats allows one to easily reuse state-of-the-art operations already implemented in existing tools such as FeatureIDE [21]. The tool also allows users to import FMs or configurations from their own environments and encourages *interoperability*. Moreover, an FM or a configuration can be exported (using the **save** operation). Outputs generated by FAMILIAR can be processed by other tools, for example, in order to relate FMs to other artefacts (e.g., code, models) [5, 6].

A FAMILIAR textual script performs a sequence of operations on FMs. Such operations are *reproducible* and *reusable*. Obtaining the same properties in a graphical editor requires an additional effort, for example, the implementation of an undo/redo system and serialization of the sequence of operations. This is very close to what *GUI scripting* languages do with macro-commands. This could have been the only requirement of the FAMILIAR language, but using the language, we believe its textual form favors *readability* of the specified operations, and more *usability* and *productivity* when dealing with compositional operations on FMs. On the other hand, graphical visualization has proved to assist users: the complementarity of the textual and visual techniques needs to be more deeply investigated (see next paragraph).

Future Work. The next step of our work is to empirically assess FAMILIAR. First, we are currently developing several small case studies to experiment and validate the different language constructs on a small but varied scale. On a large scale two case studies are going to start at the time of writing. The first one involves hundreds of (legacy) services in the medical imaging domain that *i)* exhibit high variability, *ii)* are selected using different criteria and *iii)* are consistently assembled in a scientific workflow [18]. Various scripts are being developed to *i)* build catalogs (or repositories) of analysis services to facilitate the tasks of identifying and tailoring services ; *ii)* to select services among sets of competing services provided by different organizations (companies, research groups, scientists, etc.). The second case is in the Video Surveillance domain where several components are composed into a processing chain to be deployed in various contexts while being adaptable at runtime [17]. In the two case studies, the DSL will be used to implement the management of several multiple SPLs. Besides the integration of the language to the FeatureIDE graphical editor has been done to support experimentation [16]. This will help us identify complementarity ways of managing FMs and to develop methodological guidelines.

Acknowledgments Dr. France's work is supported by the National Science Foundation under Grant No. CCF-1018711.

6. REFERENCES

[1] Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag (2005)
[2] Kang, K., Kim, S., Lee, J., Kim, K., Shin, E., Huh, M.: Form: A feature-oriented reuse method with domain-specific reference architectures. Annals of Software Engineering **5**(1) (1998) 143–168
[3] Batory, D.S.: Feature models, grammars, and propositional formulas. In: SPLC'05. Volume 3714 of LNCS. (2005) 7–20
[4] Schobbens, P.Y., Heymans, P., Trigaux, J.C., Bontemps, Y.: Generic semantics of feature diagrams. Comput. Netw. **51**(2) (2007) 456–479
[5] Czarnecki, K., Antkiewicz, M.: Mapping features to models: A template approach based on superimposed variants. In: GPCE'05. Volume 3676 of LNCS. (2005) 422–437
[6] Heidenreich, F., Sanchez, P., Santos, J., Zschaler, S., Alferez, M., Araujo, J., Fuentes, L., amd Ana Moreira, U.K., Rashid, A.: Relating feature models to other models of a software product line: A comparative study of featuremapper and vml*. TAOSD VII, Special Issue on A Common Case Study for Aspect-Oriented Modeling **6210** (2010) 69–114
[7] Czarnecki, K., Helsen, S., Eisenecker, U.: Staged Configuration through Specialization and Multilevel Configuration of Feature Models. Software Process: Improvement and Practice **10**(2) (2005) 143–169
[8] Reiser, M.O., Weber, M.: Multi-level feature trees: A pragmatic approach to managing highly complex product families. Requir. Eng. **12**(2) (2007) 57–75
[9] Hartmann, H., Trew, T.: Using feature diagrams with context variability to model multiple product lines for software supply chains. In: SPLC'08, IEEE (2008) 12–21
[10] Hartmann, H., Trew, T., Matsinger, A.: Supplier independent feature modelling. In: SPLC'09, IEEE (2009) 191–200
[11] Grünbacher, P., Rabiser, R., Dhungana, D., Lehofer, M.: Structuring the product line modeling space: Strategies and examples. In: VaMoS'09. (2009) 77–82
[12] Benavides, D., Segura, S., Ruiz-Cortés, A.: Automated Analysis of Feature Models 20 years Later: a Literature Review. Information Systems, Elsevier (2010)
[13] Thüm, T., Batory, D., Kästner, C.: Reasoning about edits to feature models. In: ICSE'09, IEEE (2009) 254–264
[14] Acher, M., Collet, P., Lahire, P., France, R.: Composing Feature Models. In: SLE'09. LNCS (2009) 62–81
[15] Acher, M., Collet, P., Lahire, P., France, R.: Comparing Approaches to Implement Feature Model Composition. In: ECMFA'10. Volume 6138 of LNCS., Springer (2010)
[16] FAMILIAR: FeAture Model scrIpt Language for manIpulation and Automatic Reasoning: `http://nyx.unice.fr/projects/familiar/`
[17] Acher, M., Collet, P., Fleurey, F., Lahire, P., Moisan, S., Rigault, J.P.: Modeling Context and Dynamic Adaptations with Feature Models. In: Int'l Workshop Models@run.time at Models 2009 (MRT'09). (2009)
[18] Acher, M., Collet, P., Lahire, P., France, R.: Managing Variability in Workflow with Feature Model Composition Operators. In: SC'10. LNCS, Springer (June 2010)
[19] Tun, T.T., Boucher, Q., Classen, A., Hubaux, A., Heymans, P.: Relating requirements and feature configurations: A systematic approach. In: SPLC'09, IEEE (2009) 201–210
[20] `http://www.pure-systems.com/pure_variants.49.0.html`
[21] `wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/`
[22] `http://www.splot-research.org/`
[23] FaMa: `http://www.isa.us.es/fama/`
[24] Xtext: `http://www.eclipse.org/Xtext/`
[25] Boucher, Q., Classen, A., Faber, P., Heymans, P.: Introducing TVL, a text-based feature modelling language. In: VaMoS'10. (2010) 159–162