Managing Feature Models with FAMILIAR: a Demonstration of the Language and its Tool Support

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

Developing software product lines involves modeling a large number of features, usually using feature models, that represent different viewpoints, sub-systems or concerns of the software system. To manage complexity on a large scale, there is a need to separate, relate and compose several feature models while automating the reasoning on their compositions. This demonstration gives an overview of a Domain-Specific Language, FAMILIAR, that is dedicated to the management of feature models. Its comprehensive programming environment, based on Eclipse, is also described. It complements existing tool support (i.e., FeatureIDE).

Keywords

Domain-Specific Language, Feature Models, Product Lines

1. INTRODUCTION

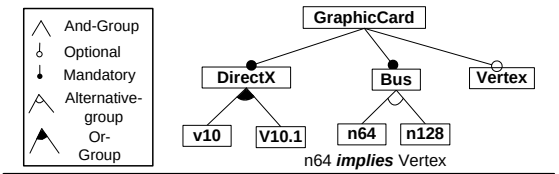
A software product line (SPL) can be seen as a family of software products that are built in a prescribed manner from a common set of core development assets [1]. Feature models (FMs) [2,3,4] are extensively used to compactly represent product commonalities and variabilities when engineering SPL. FMs can be used to describe software features at various levels of abstraction and thus can be used to manage variability of artifacts that are produced in different development phases [1, 5, 6]. Currently SPLs are becoming increasingly complex and easily span multiple organizations, so that feature models with thousands of features are now observed [7, 8, 9, 10, 11]. It has been shown that building, manipulating, and evolving these large FMs using current technologies is challenging and error-prone [12, 13].

Considering the need to manage large and complex FMs in a scalable way, our research work focuses on providing sound basis, language and tool support to help the stakeholders that have to manipulate FMs. 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 provide SPL developers with both the means to control when and how composition and analysis mechanisms are applied, and the capability to replay and reuse such procedures, we have developed a language that is dedicated to FM manipulations. This domain-specific language (DSL), named FAMILIAR (for FeAture Model scrIpt Language for manIpulation and Automatic Reasoning [16, 17]), can be used together with FM editors and reasoning tools to support large scale management of FMs. The next section gives background information on FMs and discusses the motivation behind the presented language. Section 3 presents the main constructs of FAMILIAR while Section 4 describes its programming environment, based on Eclipse, which complements existing tool support (i.e., FeatureIDE [18]). The current applicative case studies are also described together with the agenda of the demonstration.

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[1].




```
GraphicCard: DirectX Bus [Vertex] ; // Vertex is optional
DirectX : (v10 | v10.1)+ ; // Or-group
Bus : (n64 | n128) ; // Alternative-group
n64 -> Vertex ; // Constraints
```

Textual notation

Figure 1: Modeling a graphic card family

An FM can be described graphically or textually (see Figure 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

[1]We consider only FMs in their basic, propositional form [19]. We do not consider other notations and richer formalisms, for example, cardinality-based FMs [20] or FMs with attributes [12, 21]

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)* Additional constraints represented as propositional formulas must hold. A configuration is defined as a set of selected features, e.g., {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 Feature Model Management

In some previous work, we have proposed and implemented composition operators over FMs [14, 15] so that the principles of separation of concerns can be applied while developing FMs. The validation of these proposals on different SPLs (Video Surveillance Systems [22] and Medical Imaging Workflows [23]) establishes their relevance, but also shows that the large scale handling of multiple SPLs (e.g., SPLs for systems of systems) requires managing sequences of different manipulations. These are typically evolving FM structures and their configurations, inserting new features, extracting sub-FMs while maintaining constraints, and reasoning about intermediate results during composition. We then advocate that in order to manage large FMs in a scalable way, one need an approach that enables at the same time i) *the separation and composition of concerns in FMs* and ii) *rigorous reasoning about FMs.*

Separating concerns inside an SPL enables developers to understand and analyze them independently of others. Mechanisms must then be provided to master the building of large FMs from smaller ones and to reuse feature structures across different product lines (like in the consumer electronics domain [9]). Currently, organizations must more and more manage variability in product parts provided by external suppliers. 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 discussed above.

Besides support for separating and composing FMs must be coupled with support for reasoning about FMs before, during, and after composition. An FM is defined not only by its structure, but also by its configuration set and manually analyzing complex FMs is an error-prone and tedious activity. The need for tools automating relevant aspects of the reasoning process has been determined [12]. There are some approaches to systematically analyze FMs and relations between them. For instance, a classification has been proposed to characterize the relationship between two FMs in terms of sets of configuration [13] and an algorithm is also defined to compute the kind of relations between them. 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 [3,12]. To the best of our knowledge, there is no approach that enables one to control when and how reasoning tools are applied in an FM development environment while supporting separation of concerns over FMs.

2.3 Rationale for a DSL

We identify at least three relevant solutions to meet the evoked requirements. 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 [24], FeatureIDE [18], SPLOT [25], etc.). Nevertheless, they do not fully support the composition of several separated FMs. A conceivable solution would be to integrate our FM composition operators as additional functionalities inside a mainstream graphical editor but our case studies [22,23] indicate that manipulating several FMs with composition operators strongly 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 (see Section 4).

As editors like FeatureIDE and frameworks such as FAMA [26] 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, we expect such a DSL to be more easily usable by FM stakeholders with a more favorable learning curve.

3. LANGUAGE IN A NUTSHELL

The FAMILIAR DSL is an executable scripting language that supports manipulating and reasoning about FMs. We give here an overview of 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[2]. 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*. Types have accessors for observing the content of a variable. For example, one can get the parent feature of a FM, its root, its name (unique within an FM [3,4]), the set of its subfeatures, etc. An illustrative script is given below (lines **1-4** define two variables of type *Feature Model*, *gc1* and *gc2*):

```
1 gc1 = FM ( GraphicCard: DirectX Speed MemoryBus [Multi];
2    DirectX: (v10.1 | v10); Speed: (n800|n1000); MemoryBus: n128;)
```

[2]The notions of *reference* and *value* are similar to the ones used in programming languages.

```
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
9  gc3 = FM ( GraphicCard: DirectX Speed Bus Multi;
10              DirectX: (v11|v10.1) ; Speed: n1000 ; Bus: n256;)
11 f1 = parent v11 // f1 refers to feature named 'DirectX' in gc3
12 f2 = root gc3 // f2 refers to feature named 'GraphicCard' in gc3
13 s1 = name f2 // s1 is a string "GraphicCard"
14 fs = children f1 // set of features named 'v11' and 'v10.1' in gc3
```

3.2 Operations on FMs

The language provides operations for renaming and removing features in FMs:

```
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
```

It must be noted 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 and returns **true** on success.

The language also allows developers to create FM configurations, and then **select**, **deselect**, or **unselect** a feature. To **select** (resp. **deselect**) a feature means that the derived product will (resp. not) include the feature. Each of these configuration manipulation operations returns **true** on success:

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

Besides 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  cmp1 = 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 addition, FAMILIAR provides a basic **if then else** conditional construct and a **foreach** loop to iterate over a set.

3.3 FM Composition

The **insert** operator produces an FM by inserting an FM into another base or target FM. The operator takes three arguments, i.e. the FM to be inserted, the feature in the base/target FM where the insertion is to take place and the operator determining the form of the insertion. The base FM is modified if the insertion succeeds:

```
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
```

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.

When multiple *views* on a SPL must be handled, it is likely that FMs 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 the sets of configurations of FM_1 and FM_2.

The variability information associated with features in the merged FM is different according to the merge mode and the preserved properties. The reader will find all details on merge modes and additional properties regarding notably the hierarchy of input FMs in [15] and in the online reference manual of FAMILIAR [17]. The following examples illustrates the *intersection* mode:

```
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 (*gc_inter*). We check this FM against an expected FM in line **8**. For brevity's sake we do not illustrate here the union and strict union modes.

Another form of composition can be applied using constraints between features so that separated FMs are interrelated. Constraints usually reduce the sets of valid configurations. To be consistent with the rest of the language, the developer can only specify *internal* constraints within

an FM, but some compositional facilities are provided to maintain expressiveness. As a result, the **aggregate** operator enables one to create a new FM as a straightforward composition of a set of the interrelated FMs. It is created with a *synthetic* root and a copy of each FM linked to the others thanks to an *And*-operator with the mandatory status. The use of **aggregate** is illustrated Lines **3** and **6** below. Constraints can also be specified independently of any FM: the name of the feature mentioned in the constraint definition is only *mapped* on demand to the aggregated FM. The mapping may be performed when aggregate is used (line **6**) or later on using the operator **map** (line **5**). When the mapping is done, all the features mentioned are set to false in their propositional logic representation so that a simplification may occur. This ensures well-formed constraints within an FM.

```
1 fm1 = FM ( A : B H [C] ; H: (E|F) ; )
2 fm2 = FM ( R : S T [U] ; T : (W|Y)+ ; )
3 fm3 = aggregate { fm1 fm2 }
4 cst1 = { C implies W; U implies W; U excludes (E or C) ; }
5 map fm3 with cst1
6 fm4  = aggregate { fm1 fm2 } withMapping cst1
```

Note that some operations of FAMILIAR, not presented here (**configs**, **deads**, **autoSelect**, **cleanup**, etc.), are documented online in the reference manual [17].

3.4 Modularization mechanisms

All FAMILIAR statements are organized in scripts and modularization mechanisms then allow for the creation and use of multiple scripts in a single SPL project, as well as their reuse in other projects.

To handle conflicting names of variables, FAMILIAR relies on namespaces to allow disambiguation. 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
```

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 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.

Moreover 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
2  parameter target : FeatureModel
3  parameter fmToInsert : FeatureModel
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 variability op
9  assert (removeFeature target.f) // the feature must exist
10 insert fmToInsert into parentF with operatorF
11 hide ft f parentF operatorF // temp vars not needed anymore
```

4. TOOL SUPPORT AND DEMONSTRATION

Implementation. FAMILIAR [17] is developed in Java language using Xtext, a framework for the development of DSLs. As shown in Figure 2, we provide an Eclipse text editor (see ①) and an interpreter that executes the various scripts. The interpreter can be used in an interactive mode (see ②). We provide multiple notations for specifying FMs (SPLOT [25], GUIDSL/FeatureIDE [3, 18], a subset of TVL [21] as well as the notation used in the paper). Off-the-shelf SAT solvers (i.e., SAT4J) and BDD library (i.e., JavaBDD) are internally used to perform some FAMILIAR operations (e.g., merging of FMs, configuration operations). 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 feature models to other artefacts (e.g., code, models [5, 6]).

Textual and Graphical Facilities. 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, 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, for example, during configuration process. The integration of the language to the FeatureIDE graphical editor has been done to support experimentation, for example, all graphical edits are synchronized with variables environment (see ③) and all interactive commands are synchronized with the graphical editors (see ④). The complementarity of the textual and visual techniques is currently being investigated using the FAMILIAR environment.

Case Studies. The next step of our work is to empirically assess FAMILIAR. First, we are currently developing several medium-sized varied case studies to experiment and validate the different language constructs. On a larger scale two case studies are under incremental development and validation. 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 [23]. Various

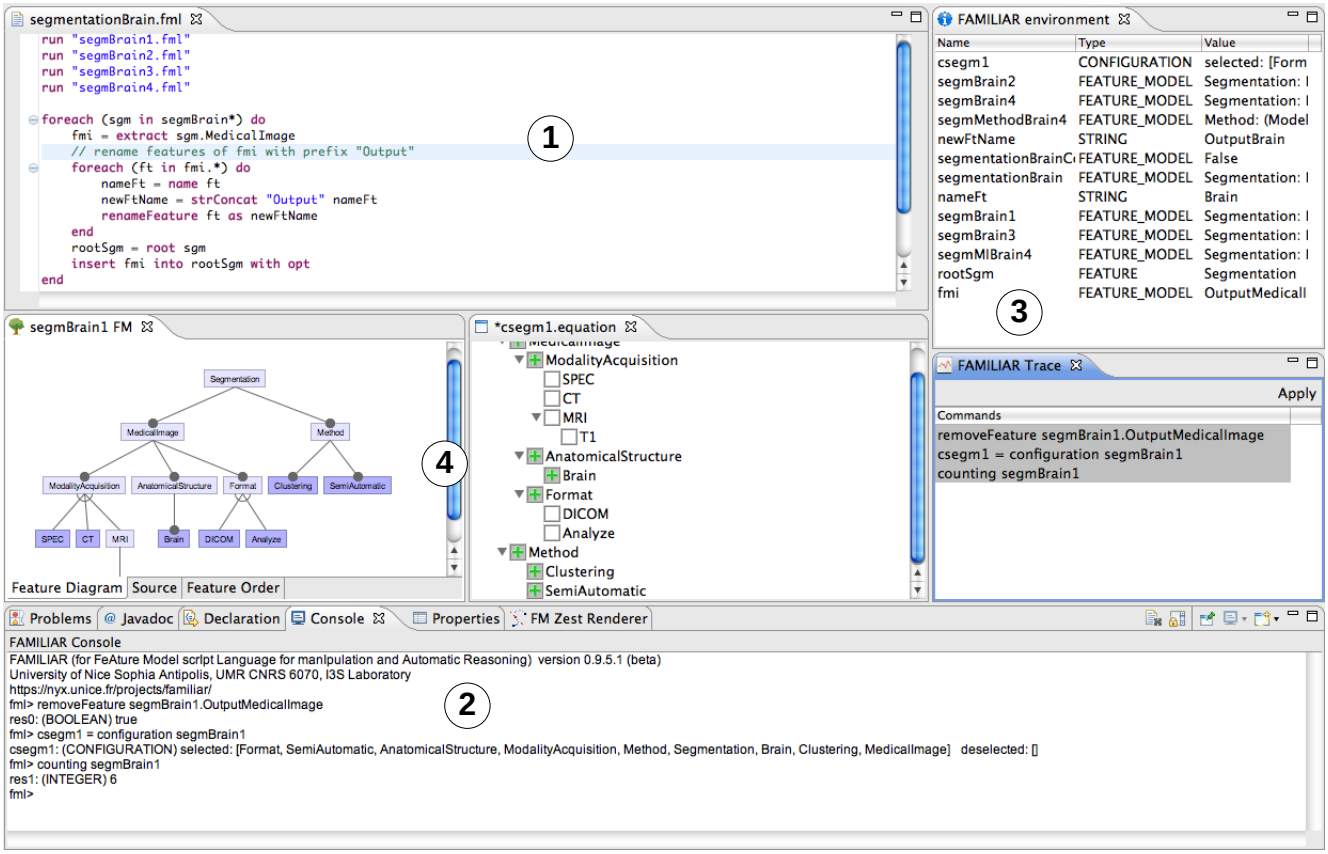


Figure 2: FAMILIAR environment

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 in which several components are composed into a processing chain to be deployed in various contexts while being adaptable at runtime [22]. In both case studies, FAMILIAR is extensively used to implement the management of several multiple SPLs.

Demonstration Proposal.

During the workshop tool support session, we propose to present the main features of the FAMILIAR language and of its supporting environment through some salient examples extracted from our different case studies. Some screen casts showing some of the environment capabilities are already available at `https://nyx.unice.fr/projects/familiar/wiki/screencasts`. To complement the proposed demonstration, we plan to make available all the corresponding screen casts on the FAMILIAR web site. We expect this presentation to foster feedbacks and discussions on the relevance of the language itself, of its positioning as a DSL, and of its possible integration in an end-to-end tool chain for large scale management of SPLs.

Acknowledgments

Dr. France's work is supported by the National Science Foundation under Grant No. CCF-1018711. The authors thank Charles Vanbeneden and Foudil Bendjabeur for their work with TVL and FeatureIDE.

5. REFERENCES

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

[2] K. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh, "Form: A feature-oriented reuse method with domain-specific reference architectures," *Annals of Software Engineering*, vol. 5, no. 1, pp. 143–168, 1998.

[3] D. S. Batory, "Feature models, grammars, and propositional formulas," in *SPLC'05*, ser. LNCS, vol. 3714, 2005, pp. 7–20.

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

[5] K. Czarnecki and M. Antkiewicz, "Mapping features to models: A template approach based on superimposed variants," in *GPCE'05*, ser. LNCS, vol. 3676, 2005, pp. 422–437.

[6] F. Heidenreich, P. Sanchez, J. Santos, S. Zschaler, M. Alferez, J. Araujo, L. Fuentes, U. K. amd Ana Moreira, and A. Rashid, "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*, vol. 6210, pp. 69–114, 2010.

[7] K. Czarnecki, S. Helsen, and U. Eisenecker, "Staged Configuration through Specialization and Multilevel Configuration of Feature Models," *Software Process: Improvement and Practice*, vol. 10, no. 2, pp. 143–169, 2005.

[8] M.-O. Reiser and M. Weber, "Multi-level feature trees: A pragmatic approach to managing highly complex product families," *Requir. Eng.*, vol. 12, no. 2, pp. 57–75, 2007.

[9] H. Hartmann and T. Trew, "Using feature diagrams with context variability to model multiple product lines for software supply chains," in *SPLC'08*. IEEE, 2008, pp. 12–21.

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

[11] P. Grünbacher, R. Rabiser, D. Dhungana, and M. Lehofer, "Structuring the product line modeling space: Strategies and examples," in *VaMoS'09*, 2009, pp. 77–82.

[12] D. Benavides, S. Segura, and A. Ruiz-Cortés, "Automated Analysis of Feature Models 20 years Later: a Literature Review," *Information Systems, Elsevier*, 2010.

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

[14] M. Acher, P. Collet, P. Lahire, and R. France, "Composing Feature Models," in *SLE'09*, ser. LNCS, 2009, pp. 62–81.

[15] ——, "Comparing Approaches to Implement Feature Model Composition," in *ECMFA'10*, vol. 6138 of LNCS. Springer, 2010.

[16] ——, "A Domain-Specific Language for Managing Feature Models," in *Symposium on Applied Computing (SAC)*, ser. , Programming Languages Track. Taiwan: ACM, Mar. 2011.

[17] FAMILIAR: FeAture Model scrIpt Language for manIpulation and Automatic Reasoning, "http://nyx.unice.fr/projects/familiar/."

[18] C. Kästner, T. Thüm, G. Saake, J. Feigenspan, T. Leich, F. Wielgorz, and S. Apel, "FeatureIDE: A tool framework for feature-oriented software development," in *ICSE'09*. IEEE, 2009, pp. 611–614.

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

[20] K. Czarnecki, S. Helsen, and U. Eisenecker, "Formalizing Cardinality-based Feature Models and their Specialization," in *Software Process Improvement and Practice*, 2005, pp. 7–29.

[21] Q. Boucher, A. Classen, P. Faber, and P. Heymans, "Introducing TVL, a text-based feature modelling language," in *VaMoS'10*, 2010, pp. 159–162.

[22] M. Acher, P. Collet, F. Fleurey, P. Lahire, S. Moisan, and J.-P. Rigault, "Modeling Context and Dynamic Adaptations with Feature Models," in *Int'l Workshop Models@run.time at Models 2009 (MRT'09)*, 2009.

[23] M. Acher, P. Collet, P. Lahire, and R. France, "Managing Variability in Workflow with Feature Model Composition Operators," in *SC'10*, ser. LNCS. Springer, Jun. 2010.

[24] D. Beuche, "Modeling and building software product lines with Pure::Variants," in *SPLC'08*, Limerick, Ireland, 2008, pp. 358–358.

[25] http://www.splot-research.org/.

[26] P. Trinidad, D. Benavides, A. Ruiz-Cortés, S. Segura, and A. Jimenez, "FAMA framework," in *SPLC'08*, Limerick, Ireland, 2008, pp. 359–359.