Domain Specific Languages for Managing Feature Models: Advances and Challenges

Philippe Collet

Domain Specific Languages for Managing Feature Models: Advances and Challenges

Philippe Collet

Univ. Nice Sophia Antipolis, CNRS, I3S, UMR 7271, 06900 Sophia Antipolis, France
Philippe.Collet@unice.fr

Abstract. Managing multiple and complex feature models is a tedious and error-prone activity in software product line engineering. Despite many advances in formal methods and analysis techniques, the supporting tools and APIs are not easily usable together, nor unified. In this paper, we report on the development and evolution of the FAMILIAR Domain-Specific Language (DSL). Its toolset is dedicated to the large scale management of feature models through a good support for separating concerns, composing feature models and scripting manipulations. We overview various applications of FAMILIAR and discuss both advantages and identified drawbacks. We then devise salient challenges to improve such DSL support in the near future.

1 Introduction

Following a *Software Product Line* SPL paradigm offers benefits such as shortened time-to-market, economies of scale and increased quality by reducing defect rates [1,2]. This paradigm basically relies on a factoring process, identifying common artifacts and managing what varies among them. These artifacts typically range from product descriptions (documentations, tabular data), requirements to models, programs and even tests. Modeling variability and managing the resulting models is a critical activity within the SPL paradigm. To deal with it, a widely used approach is to organize variability around features, which are domain abstractions relevant to stakeholders, typically being increments in program functionality [3]. Inside a SPL, a *Feature Model* (FM) is used to describe, through a compact AND-OR graph with propositional constraints, all identified features and their valid combinations [4–6]. Developments around formal semantics, analysis and reasoning techniques, as well as tool support [3–5, 7] currently make FM a *de facto* standard for managing variability.

All these advances also led to a wider usage of variability models. As one can use FM to model variability of very different kinds of concerns [3], the inherent complexity of the relations between these concerns has to be handled. With FM of hundreds to thousands of features, understanding the organization of variabilities and their complex relation rules is getting harder and harder. Reports also showed that the maintenance of a single large FM is not really feasible as some analysis techniques reach their limits, and is also not advisable as the resulting FM would be too complex to be understandable [8–13].

Tackling these issues, our research team initiated in previous work [14–18] the foundations for applying the principle of *Separation of Concerns* (SoC) to feature modeling on a large scale. Composition operators for FMs were first developed [15,16]. They notably preserve semantic properties expressed in terms of configuration sets of the composed FMs. They are complemented by a slicing operator, which produces a projection of an FM [14], and a differencing operator between FMs [18].

At that time, these operators could have been implemented using or extending one of the several Java APIs that were available (FaMa [19], FeatureIDE [20], SPLAR [21], etc.), as they provide some operations using different kinds of solvers (BDD, CSP, SAT). But with the aim to provide a better support when dealing with several feature models at the same time, we decided to build a *Domain-Specific Language* (DSL) that would provide both reasoning operations and new compositions, while focusing only on the domain concepts, i.e., feature models, features and configurations. This DSL, named FAMILIAR (for FeAture Model scrIpt Language for manIpulation and Automatic Reasoning) [22], also provides support for importing and exporting FMs, as well as for writing parameterized scripts. The language has been used in various case studies [23–25], ranging from forward to reverse engineering, with different domains and varied stakeholders. It has also evolved with extended merging techniques [26], better reverse engineering mechanisms [27], but also an additional Java API and a new implementation as an internal DSL in Scala.

In this paper, we take a step back from the development of the FAMILIAR ecosystem. After summarizing its main features, of which details can be found in references mentioned above, our contributions consist in:

- Discussing observed benefits in different case studies, while determining several recurring drawbacks. They mainly concern the fine-grained bridging with analysis and reasoning tools, the connection to other artifacts and the maintenance of the DSL itself.
- Identifying several challenges that this DSL centric approach is currently facing, from mechanisms and scope issues to the facilitation of different usages.

2 Background

2.1 Feature Modeling

The FODA method [8] first introduced the notions of feature models (FMs) together with a graphical representation through feature diagrams. An FM is structured around a *hierarchy* of features, getting into increasing detail with sub-levels, and different variability mechanisms related to feature decomposition and inter-feature constraints. In the hierarchy, the subfeatures of a feature can be *optional* or *mandatory* or can form *Xor* or *Or*-groups. Propositional constraints, typically implies or excludes rules, can be specified to express more complex dependencies between features wherever in the hierarchy.

The expressiveness of feature modeling also comes from the fact that an FM defines a set of valid feature configurations. During the configuration phase, features are selected and some rules ensure the validity of a configuration (e.g. automatic parent selection, satisfied constraints) [8]. A configuration of an FM g is defined as a set of selected features. $[\![g]\!]$ denotes the set of valid configurations of the FM g, being a set of sets of features.

FMs and propositional logic have been semantically related [5]. The set of configurations of an FM can be described by a propositional formula defined over a set of Boolean variables, in which each variable corresponds to a feature. Translating FMs into logic representations typically enables automated analysis [7].

2.2 Domain-Specific Languages

A DSL is a computer language of limited expressiveness focused on a particular domain [28]. Contrary to general purpose languages, which are aimed at handling most problems in software development, a DSL can only handle one specific aspect of a system. It is usually a small, simple and focused language [29].

In different technical domains (Unix, databases with SQL, etc.), DSLs have been used for a very long time. With their strong relation with model-driven engineering techniques, they are now getting more attention with usages in different areas related to software, being business-oriented or still technical. DSLs bring value as they can facilitate both communication with domain experts [30, 31] and programming activities in comparison with a basic Application Programming Interface (API).

But designing a DSL is not an easy task and many design trade-offs, from the scope of the language to its implementation and future maintenance, are to be made [28,30]. These languages can take the form of plain *external* DSLs, with their own custom syntax, parser and processing engine, which make a domain-specific tooling, or the form of better crafted APIs, known as *fluent* APIs, or even moving towards embedded or *internal* DSLs built on top of a host language. Numerous advances towards language workbenches [32] have been made to support the development of external DSLs. Conversely, recent advances in language design allow for easier embedding with host languages being extensible in very flexible ways [33].

3 The FAMILIAR Ecosystem

As discussed in the introduction, the FAMILIAR language was created to provide an appropriate support for the FM composition operators (see Section 3.4) that enable the large scale management of FMs following separation of concerns principles. When studying numerous examples and different case studies in which these composition operators were going to be applied [15], we observed that manipulating several FMs requires to describe and replay sequences of operations on them. We thus focused the development towards a textual language, FAMILIAR, which can define such operations in executable scripts. The DSL

functionalities comprise FM authoring and accessing operations, main reasoning operations, and the (de-)composition mechanisms. As the developed FM merging operations are restricted to propositional FMs (no feature attributes or other extensions), we also aligned the DSL on operations at the same level. Finally we decided to build an external DSL to restrict the possible manipulations to the envisioned set, and to facilitate learning and usage for different kinds of users.

FAMILIAR is available at `http://familiar-project.github.io`, with associated documentation. The reader can also refer to [22] for a presentation of FAMILIAR and to [14] for a summary of operators, more illustrations of their usage. For the sake of brevity, we do not here discuss all related work. Basically, FAMILIAR composition operators for FMs such as merging were original as they handle each operation at the semantic level (reasoning on configuration sets, see Section 3.4). The FAMILIAR language itself differs from other textual languages for feature modeling, such as Clafer [34] or TVL [35], by its capabilities to write scripts that handle several FMs at the same time.

We now overview the main constructs and data types of the FAMILIAR *external* DSL. Tool support and variants of FAMILIAR through a Java API or as a Scala based internal DSL are discussed at the end of this section.

3.1 Language Basics

FAMILIAR is a typed language that supports primitive and complex types. New types cannot be created, as the various provided types aim at supporting manipulation of FMs through a reduced but expressive set of elements. Complex types are domain-specific (*Feature Model*, *Configuration*, *Feature*, *Constraint*, etc.) or generic *Set*. Primitive types are quite common, with *String* (feature names are strings), *Boolean*, *Enum*, *Integer* and *Real*.

Operators are defined for each type, and runtime type checking is performed by the FAMILIAR interpreter. For example, the operator **counting** acts on a *Feature Model* and returns an *Integer* value. Basic arithmetic, logical, set and string operators are also provided. User-defined *variables* are also provided. In the listing below, line 1 defines a variable of type *Feature Model*. Accessors are provided for observing the content of a variable. A classical **if then else** and a loop control structure (i.e., **foreach**) complement the constructs.

```
1 mi1 = FM ( MI: Mod [Anon];  Mod: (PET | CT) ;)
2 n = counting mi1 // n is an integer
3 f1 = parent PET // f1 refers feature 'Mod' in mi1
4 f2 = root mi1 // f2 refers feature 'MI' in mi1
5 fs = children f1 // feature set {'PET','CT'} in mi1
```

3.2 Modularization

Identifiers in FAMILIAR refer to a variable identifier or to a feature in an FM. Inside one FM, feature names are supposed to be unique. The language relies on namespaces to allow disambiguation of variables having the same name. A

namespace is associated to each FM variable so that the name of such a variable followed by "." can be used to refer to a feature name, if needed.

Furthermore FAMILIAR provides modularization mechanisms that allow for the creation and use of multiple *scripts* in a single SPL project, supporting reusability of scripts. Namespaces are also used to logically group related variables of a script, making the development more modular. The listing below illustrates 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, we access to the set of all variables of *script_declaration* using a classical wildcard pattern.

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

Also, a script can be parameterized using a list of **parameter**s, a parameter recording a variable and, optionally, the type expected. Parameterized scripts are typically used to develop reusable analysis procedures for FMs and configurations. Apart from this reuse, we also found that FAMILIAR can also be used as a target language, by generating scripts handling specific tasks in SPL toolchain (checking compatibility through merging, building catalogs of descriptions). All applications discussed in Section 4 have used a combination of generated scripts and developer written ones.

3.3 Operators

For importing and exporting FMs, different FM formats are supported, including FeatureIDE, S2T2, SPLOT, subsets of TVL and FaMa. A concise notation, largely inspired from FeatureIDE [20] and the feature-model-synthesis project, is also provided. The listing below covers the main syntactic elements. In line 1, the variable *fm0* represents a FM in which MI is the root feature. Mod and Anon are child-features of MI: Mod is mandatory and Anon is optional. PET and CT are child-features of Mod and form a Xor-group. Sx and Sy are child-features of PET and form an Or-group. A cross-tree constraint is shown in line 4, as PET **excludes** Anon.

```
1 fm0 = FM ( MI: Mod [Anon];
2          Mod: (PET | CT) ;)  // Xor-group
3                         PET : (Sx|Sy)+ ;  // Or-group
4          PET excludes Anon ; // constraint )
```

FAMILIAR also allows to create FM configurations, and then **select**, **deselect**, or **unselect** a feature. Each of these configuration manipulation operations returns true if the feature does exist and if the choices conform to the FM constraints. Based on well-known applications of solvers (BDD and SAT), several operators support reasoning about FMs and their configurations. **isValid** checks whether a configuration is consistent according to its FM. Applied to a FM,

isValid determines its *satisfiability*. Besides the **isComplete** operation checks whether all features of a configuration have been chosen, i.e., selected or deselected.

The integration objective of FAMILIAR is also shown by functionalities to compare FMs. Based on the algorithm and terminology used in [6], the **compare** operation determines whether an FM is a refactoring, a generalization, a specialization or an arbitrary edit of another FM. Results from the differentiation computation between two FMs [18] are also provided through a **diff** operation.

3.4 Decomposition and Composition

The main objective of FAMILIAR is to support large-scale combinations of FMs, through decomposition and composition operations. The key feature of the main composition operations (merge, slice, diff) is that they rely on a clear semantics based on the represented configuration sets. Moreover, defining the operations through the propositional logic counterpart of the FMs allows to automatically take into account cross-tree constraints, which cannot be easily handled by syntactic techniques.

Regarding decomposition, a first basic mechanism is to **extract** a sub-tree of an FM, including cross-tree constraints involving features of the subtree. This operator is purely syntactical as it ignores cross-tree constraints that involve features not present in the sub-tree. The *semantic* counterpart of **extract** is the **slice** operator that returns a partial view of an FM according to a criterion of interest (a set of features). The semantics of the operation is based on the *projected* set of configurations of the selected features. This set is represented as its propositional logic formula and automatically takes into account cross-tree constraints. The projection is done through some logic reasoning and the result of the **slice** is a FM reconstructed from the projected set. The reader can refer to [14, 17] for formal definition and implementation details.

Two forms of composition, aggregate and merge, are supported by the FAMILIAR language. The **aggregate** operator is intended to be used when separated FMs do not have features in common, i.e., features with the same name. The operator supports cross-tree constraints, written in propositional logic, over the set of features so that the different FMs can be inter-related. The input FMs are simply put under a synthetic root as mandatory children and the propositional constraints are added to the resulting FM.

On the other hand, **merge** operators are dedicated to the composition of FMs with similar features. In this case, the operators can be used to merge the overlapping parts of the input FMs in a new FM. Variants of the merge operators defer on the production mode, e.g., merging in intersection mode computes a FM corresponding to the set of common configurations of the input FMs. Consequently the semantics of the **merge** operator variants mainly relies on the configuration sets of the input FMs (cf. Table 1). Different applications of these merge variants are mentioned in Section 4.

The default implementation of this operator computes the resulting propositional formula [16] and restores as much as possible the parent-child relationships

Mode	Semantic 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. Main merge variants in FAMILIAR

of the input FMs in the merged FM. To do so, it relies on the synthesis algorithm from [36] to build back a hierarchy. Recently, new forms of composition have been explored with differences in the expressed configurations and the ontological semantics [26]. Two new implementations have been devised and implemented, one relying on the slice operator, the other one using a local synthesis approach. This provides a range of merging variants that have different impacts on the resulting quality of the FM, the capacity to reason on it or to compose it.

3.5 Tool Support

The first version of FAMILIAR was developed in Java using Xtext[1], a framework for the development of external DSLs. Xtext facilities were used to provide a FAMILIAR script parser, an Eclipse text editor and a stand-alone console. They are all connected to the FAMILIAR kernel that deals with the main manipulated concepts (feature model, configuration, etc.), but also with transformations from feature models to the different internal and external representations (cf. Figure 1). To foster interoperability, different languages and framework format are supported through import/export methods (FeatureIDE, S2T2, SPLOT). Some of them (TVL, FaMa) are going beyond propositional FMs with feature attributes or non-boolean constructs. They are then only partially supported. This support enables FAMILIAR outputs to be processed by third party tools. For example, a connection with the graphical editor and configurator of the FeatureIDE framework [20] allows us to synchronize graphical edits and interactive FAMILIAR commands.

One of the goals was to make some existing analysis techniques available in FAMILIAR, focusing on the most important ones when several FMs have to be manipulated or composed. Consequently some FAMILIAR internal code is directly reusing or adapting several implementations, notably feature model synthesis [36] for hierarchy reconstruction of FMs, FeatureIDE [20] code for FM comparison, SPLAR for different analysis operations [21]. To perform over propositional formulas, the kernel follows a lazy strategy to compute the formulas only when needed. It relies on SAT4J for SAT solving and JavaBDD for BDDs. As they expose different advantages and drawbacks, these techniques can be switched with an annotation in FAMILIAR in many operators (except merge, which is implemented only with BDDs). A default implementation is also set for each operator. More details can be found in [22].

[1] `http://www.eclipse.org/Xtext/`

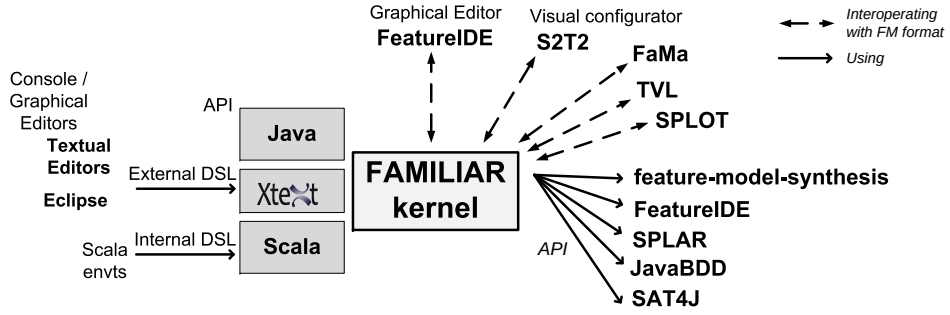

Fig. 1. Current stable FAMILIAR ecosystem

While FAMILIAR was more and more used in different applications and case studies (see next section), the source code was made open and available on github, so that the toolset can be jointly managed by three research teams, namely the Triskell team (INRIA - IRISA - University of Rennes 1), the MODALIS team (I3S laboratory - Université Nice Sophia Antipolis - CNRS) and at Colorado State University. Different extensions were then developed. The console has been extended with an interactive graphical editor, so that feature models can be directly edited or configured in sync with a text console. As FAMILIAR was also integrated in many applicative toolchains, we finally develop a Java API from the kernel to facilitate these integration tasks. Finally, we recently explored the internal DSL way to provide integration capabilities with a syntax closer to the original FAMILIAR language. We thus developed an internal DSL on top of the Scala[2] language, which provides a flexible syntax and supports mechanisms such as implicit type conversions, call-by-name parameters and mixin classes. Ongoing work notably comprise development for bridging with a CSP library and providing a web console.

4 Applications

We now report on our experience in applying FAMILIAR in various case studies, classifying them in forward and reverse engineering scenarios. They all deal with large and multiple FMs, as well as complex relationships between FMs and assets at different levels (various concerns on the same artifact, different abstraction levels, representation of different SPLs).

4.1 Forward Engineering

Scientific workflow: multiple compositions. FAMILIAR was first used to support a tooled process for assisting medical imaging experts in the error-prone activity of constructing scientific workflows [24]. These workflows are built from

[2] `http://www.scala-lang.org`

many highly customizable software services (e.g., intensity correction, segmentation), which encapsulate code from different suppliers. Separated FMs are then used to describe the variability of the different artifacts, i.e., services and workflow, with several functional and non-functional concerns, (e.g., input/ouput port, image type, used algorithm).

From a built catalog (using the merge union operator on separate descriptions of services), the workflow design process is facilitated at each step, with the capability to choose from different competing services, connect the select one in the workflow. Through automated reasoning, configuration choices and constraints on and between services are checked (using the merge intersection operator) and propagated among the workflow (using generated scripts), ensuring an overall consistent composition.

Video-surveillance: end-to-end multi-level variability. FAMILIAR has also been used on a different kind of workflow, with a more stable architecture but with more variability concerns at different levels [23]. The aim was to tame the complexity of the configuration process of a video-surveillance software pipeline. Each step was also considered as an SPL so that the variability (components, algorithms, parameters, tasks) of the underlying software platform was represented together with the variability of the hardware parts (e.g., camera capabilities). The application requirement variability was then separately captured in a domain FM, aggregating information on many context elements (e.g., lighting conditions) and expected tasks (e.g., intrusion detection). The two resulting FMs are finally related by constraints (using the aggregate operator).

Salient properties can then be checked (using parameterized scripts), such as reachability, i.e., for each high-level configuration of the domain, there exists at least one valid configuration in the software platform. The organization of the variability also allows for step-wise specialization at both levels and automatic propagation in all FMs, drastically reducing the configuration process. The remaining variability is kept at runtime to make the application self-adaptive, handling for example day/night switches.

Digital signage: multiple product lines. More recently, FAMILIAR has started to be used in the heart of an industrial-strength digital signage system developed by a start-up company and organized as a Multiple Software Product Line (MSPL) [37]. The information broadcast relies on an innovative web architecture allowing for easy aggregation of information sources and highly customizable rendering on multiple displays. Each element in the information flow is handled by a subsystem SPL represented by an FM, and a domain metamodel relates all SPLs and keeps a set of constraints between the FMs.

In this context, FAMILIAR is used for the variability definition (using merge union on all descriptions of the product instances), but also to compute the relationships between the FMs (generating appropriate scripts). As the model instance of the MSPL varies at configuration time (e.g., when a new source is added), the number of configurations also evolves. In this context, appropriate

FAMILIAR scrips allow for automatic propagation and consistency checking so that at any time, the final user is ensured to manipulate a consistent product under configuration.

Benefits. These applications illustrate different benefits of using FAMILIAR. In all of them, repositories of FMs are built and organized as reusable FAMILIAR scripts merging FMs that document some artifacts. Querying the repository is also supported by FAMILIAR with merge and slice operators. In the scientific workflow case, another DSL was designed to map services with their variability definition, and FAMILIAR was then used as an embedded language.

Generally, a Model-Driven Engineering (MDE) approach is used together with FAMILIAR and scripts are generated by the SPL toolchain to automatize some checking or propagation (e.g. at service connection, at configuration time, when the model evolves, etc.). Depending on the complexity of this coupling, the FAMILIAR Java API is more or less used in conjunction with the external DSL. Another benefit is the capability to implement more efficiently interesting properties such as realisability or usefulness when several FMs are inter-related [9,14].

Drawbacks. In the first two applications, the variability reasoning relies on **ad hoc bridges** or model-to-text transformations. The complete semantics of the solution is thus scattered through the SPL tool chain. Moreover, as there is no simple mechanism to compose external DSLs, embedding FAMILIAR in another DSL is implemented through some hacks in the Xtext back-end. Consequently very few code parts can be reused if one needs to embed FAMILIAR in another context. This is partly solved in the MSPL approach as a model drives the variability part, but still the connection semantics between the metamodel and the variability models could have been better captured.

As for the usage of FAMILIAR during execution, the adaptive part of the video-surveillance system calling the interpreter led to **performance issues at runtime**. Integrating the variability-based adaptation logic in the application engine was also very hard and it seems that a internal DSL approach would have largely simplified this task. Furthermore even at configuration time, the only means to change the implementation of the most complex operators is through a simple parameter. An average FAMILIAR user will have not enough knowledge to make the appropriate choices, especially if a script involves several operations.

Another **lack of flexibility** also appeared when we had to compute some metrics on FMs that were not present in the original DSL definition. Some of them were available in the Java API while they were needed in the DSL itself. On the other hand, implementing them in the Scala-based flavor of FAMILIAR was very fast and they were directly available in the extended language.

4.2 Reverse Engineering

Plugin-based systems: software architecture. FAMILIAR was also used to reverse engineer a variability model representing a software architecture with

plugins [25]. This was applied on several successive versions of the implemented systems.

Each time, the architectural FM was obtained by extracting variability information from both the architecture and the plugin dependencies. This creates an over-approximation of the variability, which is fixed by a slice operation made on the pure architectural part of the FM [25]. Moreover, this extracted FM was compared with another FM representing the intention of the software architect. Using several steps, scripted in FAMILIAR, the two views were reconciled to form a stable FM, which was then used to follow the evolution of the different versions.

Product descriptions: tabular data. We also explored the semiautomatic extraction of variability models from one of the most used descriptions of products, that is tabular data defining product features. A front-end enables one to give some directives on how to interpret variability and how to build the hierarchy of the resulting FM.

From the extraction tool, several FAMILIAR scripts are generated, leading to one FM per product, and all FMs are then merged in union mode to obtain an exact representation of the variability. From the first application on several public product matrices [38], this technique was applied and adapted in different contexts, such as web configurators [39] or competing visualization APIs for dashboards [40]. It is also used in the digital signage MSPL evoked in the previous section, so to populate it from different input SPLs.

Benefits. These extraction applications show again some benefits in coupling reusable parts with generated scripts in FAMILIAR. As parts of the extraction procedures have to be *ad hoc* to be adapted to the input data, the simple syntax of the external DSL was a clear advantage so to easily generate FMs. As more cases were studied, the need for a more finely parameterized operation to build a FM hierarchy arose and it was integrated in the DSL [27]. The experiment on software architecture was also the opportunity to make a software architect use the DSL to discover *hidden* features.

Drawbacks. In the extraction scenarios **ad hoc bridges** are again present. As FAMILIAR was designed to move away from a general purpose programming language, it offers only basic control structures and nothing to handle the input data. Again, the absence of a clear interface of what could or should be produced from the analyzed input to produce the resulting FM is hampering reuse between extraction chains.

Similarly the previously identified **lack of flexibility** is also characterized by the required evolution on the *ksynthesis* operator, which drives the FM hierarchy building. A first change was thus made on the whole external DSL chain, but as many different techniques are currently experimented on this hot topic of variability extraction, a more flexible evolution process is clearly needed.

5 Challenges Ahead

The development of the FAMILIAR DSL started with composition of several FMs as the main requirement. The experience built-up through its usage in different domains, life-cycle stages and with different stakeholders clearly show that the DSL is meeting this requirement. On the other hand, we identify here the main challenges that must be tackled to provide a better support to a larger variability engineering community.

Managing more explicit mechanisms. Several mechanisms inside FAMILIAR should be made more explicit and more configurable. A first obvious location in the DSL architecture is the management of reasoning back-ends. Handling the variability of back-ends is already done in some variability tool sets [19, 41]. For example FaMa [19] manages the different analyses and reasoners with a feature model capturing functional capabilities and a few non-functional properties. The challenge for FAMILIAR is to go beyond such organization, so that new solvers can be easily integrated (CSP and SMT solvers are the primary targets) and that both functional and non-functional properties can be captured and inter-related into feature models. This would also better organize the heuristics of used algorithms, like in the SPLAR Java API [21]. In addition, results from performance comparison between solvers for feature modeling operations [42] may serve as starting point.

The description of the other challenges will also show that a systematic and uniform approach should be followed to master all configurable properties in FAMILIAR, that is, not only for reasoning back-ends. The recent implementation of some variants of the merge operation is an example [26], but this is actually the case in the kernel of the DSL and in all its interaction points (extraction of variability, internal representation, relations to other models).

Extending the scope of the DSL. The second challenge is related to the advances that were made thanks to FM composition. In all the applications, FAMILIAR was an appropriate engine to deal with variability in conjunction with many different software artifacts, but the connection with these artifacts was quite often cumbersome in terms of software engineering. The move towards an internal DSL in Scala should partially solve this issue, but exploring how to facilitate the management of relationships between feature models and the whole model-driven engineering steps seems an interesting track to follow.

First this should allow to make advances in the relation between the semantics of artifact composition and the semantics of FM composition. In our recent experience on the MSPL of digital signage systems, we used a combination of metamodels and feature models that seems to be related of what is available in Clafer [34]. Still Clafer is focused on understanding domain models, whereas we completely define and implement the MSPL toolchain down to the code generation level. Different extensions of feature models should be introduced in the DSL, but this should be especially organized in terms of operations and inter-relations between the extensions. This point is related to the previous challenge,

as each extension of feature models is likely to need a fine-tuned usage of the available reasoning back-ends.

Extending the FAMILIAR scope is also needed to facilitate variability manipulation on a larger scale, especially in (semi-)automatically extraction scenarios. Currently, there is some lazy strategy implemented to reduce the transformation to propositional logic, but the available internal representations should be extended so to handle cases where only a feature set is desirable or feasible, for example when very large feature models are split and their hierarchy partially flattened [43]. The ideal functionality would handle a continuum of representations, from feature sets to feature models and the different representations needed by different reasoning approaches.

Facilitating different usages. The last challenge consists in providing the appropriate customised variants of the DSL for the different users and tasks that would be then facilitated. With an extended scope and more explicit mechanisms, different scenarios must be envisaged. Extraction processes should be supported with recurring patterns being provided in the DSL. The highly functional flavor of programming provided in Scala should enable to design a small but powerful toolkit for this purpose. Conversely the different mappings between feature models and other models, the associated realization techniques, as well as the transformation processes to different back-ends should be abstracted and organized in the same way.

Besides, one *usage* not to be neglected is the visualization and comprehension of these large and inter-related feature models. The current FAMILIAR implementation relies on interoperability formats, so that other visualization tools, such as S2T2 can be used. If different usages are supported, appropriate visualizations has to be envisaged and relationships between the DSL and third-parties toolkits should be supported as well.

Finally, these different specific parts should cleverly rely on the different internal representations discussed in the previous challenge. This, together with an efficient runtime interpreter in Scala, would be also very useful to provide a scalable variability support when the DSL is heavily exploited at runtime.

6 Conclusion

The FAMILIAR language was first developed to manipulate and compose feature models in the large, relying on formal underpinnings and bridging some existing APIs. Separation of concerns and reasoning facilities are made available through an external DSL, which has been evolved with an additional Java API and recently, an Scala based internal implementation.

We have reported several applications of the FAMILIAR toolset, ranging from semi-automatic extraction of feature models from product descriptions or software architectures, to more forward engineering cases, with scientific workflow, video-surveillance software, and digital signage systems organized in multiple

software product lines. Based on these varied experiences, we summarized obtained benefits, but especially focused on identified drawbacks: *ad hoc* integration in toolchains or applications, lack of flexibility, or performance issues during heavily runtime usage. We then devised three inter-related challenges to improve such DSL support, i.e. *i)* managing more explicit mechanisms, both internally in the used algorithms and externally when dealing with reasoning back-ends, *ii)* extending the scope of the DSL to better support extraction procedures and downstream engineering stages, and *iii)* facilitating different usages through appropriate DSL extensions.

Ongoing work aims at tackling these challenges. First steps consist in exploring how Scala facilities can help in easily integrating variability in the FAMILIAR architecture, so that other needed functionalities can be nicely and efficiently provided. We notably plan to bridge an extended version of FAMILIAR with the SIGMA family of DSLs for manipulating EMF models [44], which is also implemented in Scala. We expect to partly cover some functionalities provided by implementations of the CVL variability standard [45], but with a more lightweight and decoupled approach. Being able to easily integrate a small service provided by the FAMILIAR DSL in any toolchain is kept as a prime requirement.

Acknowledgments. The author would like to thank all the people that have worked on developing FAMILIAR (`http://familiar-project.github.io`), and especially Mathieu Acher, Philippe Lahire and Robert B. France, who were at the roots of it.

References

1. Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag (2005)
2. Clements, P., Northrop, L.M.: Software Product Lines : Practices and Patterns. Addison-Wesley Professional (2001)
3. Apel, S., Kästner, C.: An overview of feature-oriented software development. Journal of Object Technology (JOT) **8**(5) (July/August 2009) 49–84
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., Wasowski, A.: Feature diagrams and logics: There and back again. In: SPLC'07, IEEE (2007) 23–34
6. Thüm, T., Batory, D., Kästner, C.: Reasoning about edits to feature models. In: ICSE'09, ACM (2009) 254–264
7. Benavides, D., Segura, S., Ruiz-Cortes, A.: Automated analysis of feature models 20 years later: a literature review. Information Systems **35**(6) (2010)
8. 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
9. Metzger, A., Pohl, K., Heymans, P., Schobbens, P.Y., Saval, G.: Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. In: RE'07. (2007) 243–253

10. Dhungana, D., Grünbacher, P., Rabiser, R., Neumayer, T.: Structuring the modeling space and supporting evolution in software product line engineering. Journal of Systems and Software **83**(7) (2010) 1108–1122
11. Zaid, L.A., Kleinermann, F., Troyer, O.D.: Feature assembly: A new feature modeling technique. In Parsons, J., Saeki, M., Shoval, P., Woo, C.C., Wand, Y., eds.: ER. Volume 6412 of Lecture Notes in Computer Science., Springer (2010) 233–246
12. Dhungana, D., Seichter, D., Botterweck, G., Rabiser, R., Gruenbacher, P., Benavides, D., Galindo, J.A.: Configuration of multi product lines by bridging heterogeneous variability modeling approaches. In: SPLC'11, IEEE (2011)
13. Hubaux, A., Tun, T.T., Heymans, P.: Separation of concerns in feature diagram languages: A systematic survey. ACM Computing Surveys (2012)
14. Acher, M., Collet, P., Lahire, P., France, R.: Separation of Concerns in Feature Modeling: Support and Applications. In: AOSD'12, ACM (2012)
15. Acher, M., Collet, P., Lahire, P., France, R.: Composing Feature Models. In: Software Language Engineering (SLE'09). Volume 5969 of LNCS. (2009) 62–81
16. Acher, M., Collet, P., Lahire, P., France, R.: Comparing approaches to implement feature model composition. In: ECMFA'10. Volume 6138 of LNCS. (2010) 3–19
17. Acher, M., Collet, P., Lahire, P., France, R.: Slicing Feature Models. In: Proc. of ASE'11 (short paper), ACM (2011)
18. Acher, M., Heymans, P., Collet, P., Quinton, C., Lahire, P., Merle, P.: Feature Model Differences. In: 24th International Conference on Advanced Information Systems Engineering (CAiSE'12). LNCS, Springer (2012)
19. Trinidad, P., Benavides, D., Ruiz-Cortes, A., Segura, S., Jimenez, A.: FAMA framework. In: Int'l Software Product Line Conference (SPLC '08), Limerick, Ireland (2008) 359–359
20. Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G., Leich, T.: FeatureIDE: An extensible framework for feature-oriented software development. Science of Computer Programming (SCP) (2012)
21. Mendonca, M., Branco, M., Cowan, D.: S.p.l.o.t.: software product lines online tools. In: OOPSLA'09 (companion), ACM (2009)
22. Acher, M., Collet, P., Lahire, P., France, R.: Familiar: A domain-specific language for large scale management of feature models. Science of Computer Programming (SCP) Special issue on programming languages **78**(6) (2013) 657–681
23. Acher, M., Collet, P., Lahire, P., Moisan, S., Rigault, J.P.: Modeling variability from requirements to runtime. In: ICECCS'11, IEEE (2011) 77–86
24. Acher, M., Collet, P., Lahire, P., Gaignard, A., France, R., Montagnat, J.: Composing multiple variability artifacts to assemble coherent workflows. Software Quality Journal (Special issue on Quality Engineering for SPLs) (2011)
25. Acher, M., Cleve, A., Collet, P., Merle, P., Duchien, L., Lahire, P.: Extraction and Evolution of Architectural Variability Models in Plugin-based Systems. Software & Systems Modeling (SoSyM) (July 2013) 27 p.
26. Acher, M., Combemale, B., Collet, P., Barais, O., Lahire, P., France, R.: Composing your Compositions of Variability Models. In: ACM/IEEE 16th International Conference on Model Driven Engineering Languages & Systems (MODELS'2013). Volume 8107 of LNCS., Miami (USA), Springer (September 2013) 352–369
27. Acher, M., Heymans, P., Cleve, A., Hainaut, J.L., Baudry, B.: Support for reverse engineering and maintaining feature models. In: VaMoS'13, ACM (2013)
28. Fowler, M.: Domain Specific Languages. Addison-Wesley Professional (2010)
29. Hermans, F., Pinzger, M., van Deursen, A.: Domain-Specific languages in practice: A user study on the success factors. In: 12th International Conference, MODELS. Springer (2009) 423–437

30. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. **37**(4) (2005) 316–344
31. Kosar, T., Mernik, M., Carver, J.: Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments. Empirical Software Engineering **17**(3) (2012) 276–304
32. Erdweg, S. et al.: The state of the art in language workbenches. In: Software Language Engineering. Volume 8225 of Lecture Notes in Computer Science. Springer International Publishing (2013) 197–217
33. Chafi, H., DeVito, Z., Moors, A., Rompf, T., Sujeeth, A.K., Hanrahan, P., Odersky, M., Olukotun, K.: Language virtualization for heterogeneous parallel computing. In: Proceedings of the ACM international conference on Object oriented programming systems languages and applications. OOPSLA '10, ACM (October 2010)
34. Bak, K., Czarnecki, K., Wasowski, A.: Feature and meta-models in clafer: mixed, specialized, and coupled. In: SLE'10. LNCS, Springer (2011) 102–122
35. Classen, A., Boucher, Q., Heymans, P.: A text-based approach to feature modelling: Syntax and semantics of TVL. Science of Computer Programming, Special Issue on Software Evolution, Adaptability and Variability **76**(12) (2011) 1130–1143
36. Andersen, N., Czarnecki, K., She, S., Wasowski, A.: Efficient synthesis of feature models. In: Proceedings of SPLC'12, ACM Press (2012) 97–106
37. Urli, S., Mosser, S., Blay-Fornarino, M., Collet, P.: How to Exploit Domain Knowledge in Multiple Software Product Lines? In: Fourth International Workshop on Product LinE Approaches in Software Engineering at ICSE 2013 (PLEASE 2013). PLEASE, San Francisco, USA, ACM (May 2013) 4
38. Acher, M., Cleve, A., Perrouin, G., Heymans, P., Vanbeneden, C., Collet, P., Lahire, P.: On extracting feature models from product descriptions. In: VaMoS'12, ACM (2012) 45–54
39. Abbasi, E.K., Acher, M., Heymans, P., Cleve, A.: Reverse Engineering Web Configurators. In: 17th European Conference on Software Maintenance and Reengineering (CSMR), Antwerp, Belgique, IEEE (February 2014)
40. Logre, I., Mosser, S., Collet, P., Riveill, M.: Sensor Data Visualisation: a Composition-based Approach to Support Domain Variability. In: ECMFA'2014. , York, Springer (September 2014) 16
41. Cordy, M., Classen, A., Heymans, P., Schobbens, P.Y., Legay, A.: Provelines: A product line of verifiers for software product lines. In: Proceedings of the 17th International Software Product Line Conference Co-located Workshops. SPLC '13 Workshops, New York, NY, USA, ACM (2013) 141–146
42. Pohl, R., Lauenroth, K., Pohl, K.: A performance comparison of contemporary algorithmic approaches for automated analysis operations on feature models. In: Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering. ASE '11, Washington, DC, USA, IEEE Computer Society (2011) 313–322
43. Dintzner, N., Van Deursen, A., Pinzger, M.: Extracting feature model changes from the linux kernel using fmdiff. In: Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems. VaMoS '14, New York, NY, USA, ACM (2014) 22:1–22:8
44. Křikava, F., Collet, P., France, R.: Manipulating Models Using Internal Domain-Specific Languages. In: Symposium on Applied Computing (SAC), Programming Languages Track(SAC), short paper. , Gyeongju (Korea), ACM (March 2014)
45. Haugen, O., Wąsowski, A., Czarnecki, K.: Cvl: Common variability language. In: Proceedings of the 17th International Software Product Line Conference. SPLC '13, New York, NY, USA, ACM (2013) 277–277