

Reverse Engineering Architectural Feature Models

Mathieu Acher, Anthony Cleve, Philippe Collet, Philippe Merle, Laurence Duchien, Philippe Lahire

▶ **To cite this version:**

Mathieu Acher, Anthony Cleve, Philippe Collet, Philippe Merle, Laurence Duchien, et al.. Reverse Engineering Architectural Feature Models. 5th European Conference of Software Architecture (ECSA), Sep 2011, Essen, Germany. pp.220-235, 10.1007/978-3-642-23798-0_25 . inria-00614984

HAL Id: inria-00614984
https://inria.hal.science/inria-00614984

Submitted on 17 Aug 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Reverse Engineering Architectural Feature Models

Mathieu Acher[1], Anthony Cleve[2], Philippe Collet[1],
Philippe Merle[3], Laurence Duchien[3], and Philippe Lahire[1]

[1] Université de Nice Sophia Antipolis - I3S (CNRS UMR 6070), France
{acher,collet,lahire}@i3s.unice.fr
[2] PReCISE Research Centre, University of Namur, Belgium
acl@info.fundp.ac.be
[3] INRIA Lille-Nord Europe, Univ. Lille 1 - CNRS UMR 8022, France
{philippe.merle,laurence.duchien}@inria.fr

Abstract. Reverse engineering the variability of an existing system is a challenging activity. The architect knowledge is essential to identify variation points and explicit constraints between features, for instance in feature models (FMs), but the manual creation of FMs is both time-consuming and error-prone. On a large scale, it is very difficult for an architect to guarantee that the resulting FM ensures a safe composition of the architectural elements when some features are selected. In this paper, we present a comprehensive, tool supported process for reverse engineering architectural FMs. We develop automated techniques to extract and combine different variability descriptions of an architecture. Then, alignment and reasoning techniques are applied to integrate the architect knowledge and reinforce the extracted FM. We illustrate the reverse engineering process when applied to a representative software system, FraSCAti, and we report on our experience in this context.

1 Introduction

1.1 Problem

As a majority of software applications are now large-scale, business-critical, operated 24/7, distributed and ubiquitous, their complexity is increasing at a rate that outpaces all major software engineering advances. To tame it, *Software Product Line* (SPL) engineering is one of the major trends of the last decade. An SPL can be defined as "*a set of software-intensive systems that share a common, managed set of features and that are developed from a common set of core assets in a prescribed way*" [9]. SPL engineering aims at generating tailor-made variants for the needs of particular customers or environments and promotes the systematic reuse of software artifacts. An SPL development starts with an analysis of the domain to identify commonalities and variabilities (i.e., differences) between the members of the SPL. A common way is to describe variability of an SPL in terms of features, which are domain abstractions relevant to stakeholders. A *Feature Model* is used to compactly define all features in an SPL and their valid combinations [6,20,11].

When SPL engineering is conducted right from the start on a software architecture, it is feasible to manage variability through one or more *architectural* feature models and then associate them to the software architecture [19]. The major architectural variations are then associated to some of the features and allows for some safe compositions in the architecture when features are selected to configure a software product from the line. One of the resulting properties of crucial importance is the guarantee that the variability is not only preserved but also kept consistent across all used artefacts [10,4,17].

But in many cases, legacy systems must be taken into account, or more often, the software system is developed incrementally. It is not a SPL at the beginning, but it becomes so complex, with many configuration and extension points, that its variability must be handled with SPL engineering techniques. In this context, the task of constructing feature models, which is intrinsically difficult, becomes very arduous for software architects, especially if they are presented with unrelated documents from which to derive the architectural feature model. It is then necessary to obtain a consistent feature model from the actual architecture. On a large scale both automatic extraction from existing parts and the architect knowledge should be ideally combined to achieve this goal.

1.2 FraSCAti: the Need for Handling Variability

In this paper, we illustrate our proposal and report on experiments with a case study on the FraSCAti platform [13]. It is an open-source implementation of the Service Component Architecture (SCA) standard [23], which allows for building hierarchical component architectures with the potential support of many component and service technologies.

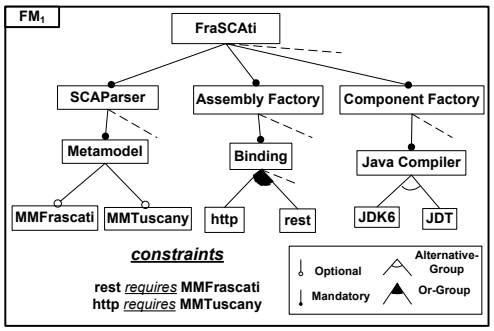
Started three years ago, the development of the FraSCAti platform begun with a framework, first validated by a basic implementation of the standard, and then incrementally enhanced. After four major releases, it now supports several SCA specifications (Assembly Model, Transaction Policy, Java Common Annotations and APIs, Java Component Implementation, Spring Component Implementation, BPEL Client and Implementation, Web Services Binding, JMS Binding), and provides a set of extensions to the standard, including binding implementation types (Java RMI, SOAP, REST, JSON-RPC, JNA, UPnP, etc.), component implementation types (Java, OSGi, Java supported scripting languages, Scala, Fractal), interface description types (Java, C headers, WSDL, UPnP), runtime API for assembly and component introspection/reconfiguration [21]. As its capabilities grew, FraSCAti has also been refactored and completely architected itself with SCA components.

With all these capabilities, the platform has become highly (re-)configurable in many parts of its own architecture. It notably exposes a larger number of extensions that can be activated throughout the platform, creating numerous variants of a FraSCAti deployment. For example, some variations consist in one or more specific components bound to many other mandatory or optional parts of the platform architecture. It then became obvious to FraSCAti technical leads

that the variability [4] of the platform should be managed to pilot and control its evolution as an SPL.

1.3 Feature Modeling

The management and modeling of variability is a central activity in SPL engineering. We chose to rely on a particular kind of variability model, *Feature Models* (FMs), based on their wide adoption, the existence of formal semantics, reasoning techniques and tool support [6,20,11,7]. FMs compactly represent product commonalities and variabilities in terms of features. FMs can be used to describe software features at various levels of abstraction and thus can be used throughout the software lifecycle on different artefacts [10,16,4].



(a) an excerpt of a possible architectural FM

$$
\begin{aligned}
\phi_1 \;=\; & FraSCAti \\
\wedge\; & FraSCAti \Leftrightarrow AssemblyFactory \\
\wedge\; & FraSCAti \Leftrightarrow ComponentFactory \\
\wedge\; & FraSCAti \Leftrightarrow SCAParser \\
\wedge\; & SCAParser \Leftrightarrow Metamodel \\
\wedge\; & AssemblyFactory \Leftrightarrow Binding \\
\wedge\; & ComponentFactory \Leftrightarrow JavaCompiler \\
\wedge\; & JavaCompiler \Rightarrow JDK6 \vee\; JDT \\
\wedge\; & \neg\, JDK6 \vee\; \neg JDT \\
\wedge\; & MMFrascati \Rightarrow Metamodel \\
\wedge\; & MMTuscany \Rightarrow Metamodel \\
\wedge\; & http \Rightarrow Binding \\
\wedge\; & rest \Rightarrow Binding \\
\wedge\; & Binding \Rightarrow rest \vee http \\
\wedge\; & rest \Rightarrow MMFrascati \\
\wedge\; & http \Rightarrow MMTuscany
\end{aligned}
$$

(b) propositional logic encoding

Fig. 1. Feature Model and Propositional Logic Encoding

FMs hierarchically structure application features into multiple levels of increasing detail. Fig. 1(a) shows an example of an FM. This FM is an extract of a *architectural FM* that is dedicated to the description of the FraSCAti platform, whereas FM are also widely used to capture and structure requirements. It must be noted that the FM that we discuss all along this paper are only architectural ones. As in typical SPLs, not all combinations of features or *configurations* (see Definition 1) are valid. Variability defines what the allowed configurations are. When decomposing a feature into subfeatures, the subfeatures may be *optional*, *mandatory* or may form *Or* or *Alternative*-groups (e.g., $JDK6$ and JDT form an Alternative-group, *http* and *rest* form an Or-group). An additional mechanism to specify variability is to add constraints (expressed in propositional logic), which may cut across the feature hierarchy (e.g., *rest* requires $MMFrascati$). The validity of a configuration is determined by the semantics of FMs, e.g. $JDK6$

[4] We use here the term *variability* as in the definition from [24]: "software variability is the ability of a software system or artefact to be efficiently extended, changed, customized or configured for use in a particular context."

and JDT are mutually exclusive and cannot be selected at the same time. A valid configuration is obtained by selecting/deselecting features from parents to children while following the rules imposed by the operators (e.g., exactly one subfeature must be selected in an Alternative) and the constraints. For example, $\{FraSCAti, SCAParser, AssemblyFactory, ComponentFactory, Metamodel, MMFrascati, Binding, rest, JavaCompiler, JDT\}$ is a valid configuration of the FM shown in Fig. 1(a).

Definition 1 (SPL, Feature Model) *A software product line SPL_i is a set of products described by a feature model FM_i. The set of features of FM_i is denoted $\mathcal{F}_{FM_i}$. Each product of SPL_i is a combination of features and corresponds to a valid configuration of FM_i. A configuration c of FM_i is defined as a set of features selected, i.e., $c = \{f_1, f_2, \ldots, f_m\} \subseteq \mathcal{F}_{FM_i}$. $[\![FM_i]\!]$ denotes the set of valid configurations of the feature model FM_i. We note ϕ_i the propositional formula of FM_i.*

The set of configurations represented by an FM can be described by a propositional formula defined over a set of Boolean variables, where each variable corresponds to a feature [6,11]. Figure 1(b) also shows the mapping of the FM to a propositional formula. The propositional formula can be used to automatically reason about properties of an FM (e.g., see [7]). In particular, if an assignment to its Boolean variables is satisfiable, then the selection/deselection of the corresponding features respects the rules evoked above.

1.4 Reverse Engineering FraSCAti as an SPL

In order to manage the FraSCAti platform as an SPL, we needed to capture its variability from the existing architecture. Several software artefacts (SCA composite files, Maven descriptors, unformal documents) describe FraSCAti architecture, but variability, though, is not explicitly represented.As the FraSCAti main *software architect* (SA) had an extensive expertise in the architecture and in its evolution, it was decided to make him model the architecture he has in mind with variation points (see left part of Fig. 2). As a domain expert, he had the ability to elicitate the architectural variation points and explain rationale behind these decisions. To follow separation of concerns principles, it was also decided to separate the variability description from the architectural model itself. The principle is to model the variation points of the architecture, to represent them as features in an architectural FM, and finally to describe the links between the features and the architectural elements. An important property is then to ensure consistency between FM and architectures [17], even if not all variability elements can be captured in an FM.

This task resulted in a manually created FM and it was clearly daunting, time-consuming and error-prone, requiring substantial effort from the SA. In this case as in all large scale architectures, it is very difficult to guarantee that the resulting FM ensures a safe composition of the architectural elements when some features are selected. Another approach thus relies on an automated extraction, so that an architectural FM that represents variability of the architecture is automatically extracted from the appropriate artefacts (see right part

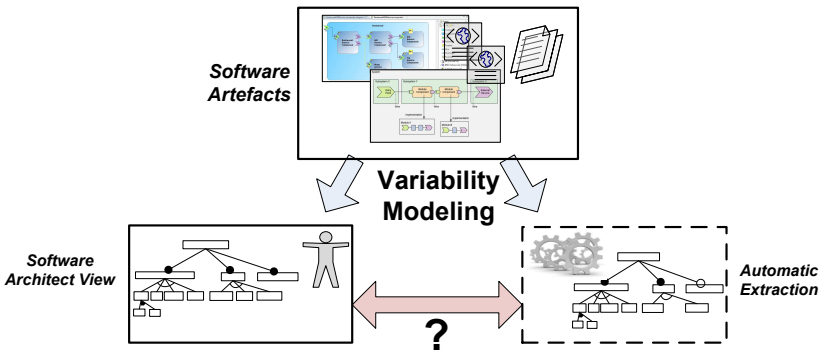


Fig. 2. Variability Modeling from Software Artefacts

of Fig. 2).This operation clearly saves time and reduces accidental complexity, but the accuracy of the results directly depends on the quality of the available documents and of the extraction procedure. This approach is notably followed in recent reverse engineering work which is doing large scale variability extraction from the Linux kernel [22].

The main challenge is then to reconcile these two architectural FMs into a final FM being compatible with both the SA view and the actual architecture. It must also be noted that we could have tried to somehow *integrate* the SA knowledge in the extraction process or to let him edit an extracted FM, but we argue that keeping the first two activities separated was better. It lets a highly experienced SA focus on its own variability scoping, and compare it afterwards to the extracted version. Moreover, this allows for explicitly separating the required variability of the SA from the supported variability of the actual software system, as advocated in [18].

In this paper, we present a comprehensive, tool supported process for reverse engineering architectural FMs. We develop automated techniques to extract and combine different variability descriptions of an architecture. Then, alignment and reasoning techniques are applied to integrate the architect knowledge and reinforce the extracted FM. In the remainder of this paper we describe the automated extraction process that we have applied to FraSCAti FM(Section 2). We then show how the process is completed by refinement steps that enables the architect to compare and integrate his/her knowledge, so that a consistent architectural FM is obtained (Section 3). This process is validated by experiments on the FraSCAti architecture and some lessons learned are briefly discussed. Related work are studied in Section 4 while Section 5 concludes the paper.

2 Automatic Extraction of Architectural Feature Model

Overview. Fig. 3 summarizes the steps needed to realize the process. First, a raw *architectural feature model*, noted $FM_{Arch_{150}}$, is extracted from a *150% architecture* of the system (see ①). The latter consists of the composition of the architecture fragments of *all* the system plugins. We call it a 150% architecture because it is not likely that the system may contain them all. Consequently,

$FM_{Arch_{150}}$ does include all the *features* provided by the system, but it still constitutes an over approximation of the set of *valid combinations* of features of the system family. Indeed, some features may actually *require* or *exclude* other features, which is not always detectable in the architecture. Hence the need for considering an additional source of information. We therefore also analyze the specification of the system plugins and the dependencies declared between them, with the ultimate goal of deriving inter-feature constraints from inter-plugin constraints. To this end, we extract a *plugin feature model* FM_{Plug}, that represents the system plugins and their dependencies (see ②). Then, we automatically reconstruct the bidirectional mapping that holds between the features of FM_{Plug} and those of $FM_{Arch_{150}}$ (see ③). Finally, we exploit this mapping as a basis to derive a richer architectural FM, noted FM_{Arch}, where additional feature constraints have been added. As compared to $FM_{Arch_{150}}$, FM_{Arch} represents much more accurately the architectural variability provided by the system.

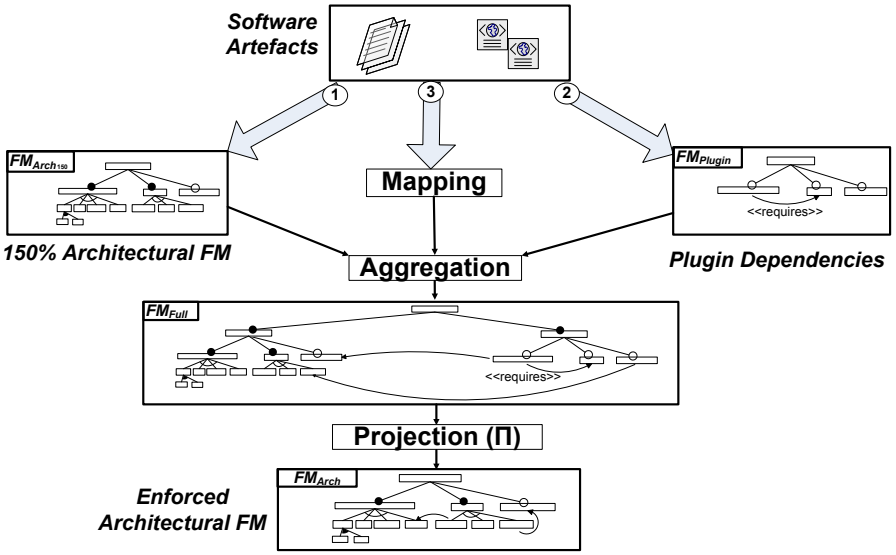


Fig. 3. Process for Extracting FM_{Arch}

2.1 Extracting $FM_{Arch_{150}}$

The architectural FM extraction process starts from a set of n system plugins (or *modules*), each defining an architecture fragment. In order to extract an architectural FM representing the entire product family, we need to consider *all* the system plugins at the same time. We therefore produce a *150% architecture* of the system, noted $Arch_{150}$. It consists of a hierarchy of components. In the SCA vocabulary, each component may be a composite, itself further decomposed into other components. Each component may provide a set of *services*, and may specify a set of *references* to other services. Services and references having compatible *interfaces* may be bound together via *wires*. Each wire has a reference

as *source* and a service as *target*. Each reference r has a *multiplicity*, specifying the minimal and maximal number of services that can be bound to r via wires. A reference having a 0..1 or 0..N multiplicity is called *optional*.

Note that $Arch_{150}$ may not correspond to the architecture of a *legal* product in the system family. For instance, several components may exclude each other because they all define a service matching the same 0..1 reference r. In this case, the architecture composition algorithm binds only one component service to r, while the other ones are left unbound in the architecture.

Since the extracted architectural FM should represent the *variability* of the system of interest, we focus on its *extension points*, that are typically materialized by *optional* references in $Arch_{150}$. Algorithm 1 summarizes the behavior of the FM extractor. The root feature of the extracted FM (f_{root}) corresponds to the

Algorithm 1 $ExtractArchitecturalFM_{150}(Arch_{150})$

Require: A 150% architecture of the plugin-based system ($Arch_{150}$).
Ensure: A feature model approximating the system family ($FM_{Arch_{150}}$).
1: $root \leftarrow MainComposite(Arch_{150})$
2: $f_{root} \leftarrow CreateFeature(root)$
3: $FM_{Arch_{150}} \leftarrow SetRootFeature(FM_{Arch_{150}}, f_{root})$
4: **for all** $c \in FirstLevelComponents(root)$ **do**
5: $\quad f_c \leftarrow CreateFeature(c)$
6: $\quad FM_{Arch_{150}} \leftarrow AddMandatoryChildFeature(FM_{Arch_{150}}, f_{root}, f_c)$
7: $\quad FM_{Arch_{150}} \leftarrow AddChildFeatures(FM_{Arch_{150}}, c, f_c, Arch_{150})$
8: **end for**

main composite (*root*) of $Arch_{150}$. The child features of f_{root} are the first-level components of *root*, the latter being considered as the main system features. The lower levels of child features are produced by the *AddChildFeatures* function (Algorithm 2). This recursive function looks for all the optional references r of component c and, for each of them, creates an optional child feature f_r, itself further decomposed through a *XOR* or an *OR* group (depending on the multiplicity of r). The child features f_{c_s} of the group correspond to all components c_s providing a service compatible with r.

2.2 Extracting FM_{Plug}

The extraction of the plugin feature model FM_{Plug} starts from the set of plugins $P = \{p_1, p_2, \ldots, p_n\}$ composing the system. This extraction is straightforward: each plugin p_i becomes a feature f_{p_i} of FM_{Plug}. If a plugin p_i is part of the system core, f_{p_i} is a mandatory feature, otherwise it is an optional feature. Each dependency of the form p_i *depends on* p_j is translated as an inter-feature dependency $f_{p_i} requires f_{p_j}$. Similarly, each p_i *excludes* p_j constraint is rewritten as an *excludes* dependency between f_{p_i} and f_{p_j}.

2.3 Mapping $FM_{Arch_{150}}$ and FM_{Plug}

When producing $Arch_{150}$, we keep track of the relationship between the input plugins and the architectural elements they define, and vice versa. On this basis, we specify a bidirectional mapping between the features of $FM_{Arch_{150}}$ and

Algorithm 2 $AddChildFeatures(FM, c, f_p, Arch_{150})$

Require: A feature model (FM), a component (c), a parent feature (f_p), a 150% architecture ($Arch_{150}$).
Ensure: FM enriched with the child features of f_p, if any.

```
for all r ∈ OptionalReferences(c) do
    MC ← FindMatchingComponents(Arch150, r)
    if MC ≠ ∅ then
        f_r ← CreateFeature(r)
        FM ← AddOptionalChildFeature(FM, f_p, f_r)
        if Multiplicy(r) = 0..1 then
            g ← CreateXORGroup()
        else if Multiplicy(r) = 0..N then
            g ← CreateORGroup()
        end if
        FM ← AddGroup(FM, f_r, g)
        for all c_s ∈ MC do
            f_cs ← CreateFeature(c_s)
            FM ← AddChildFeatureOfGroup(FM, g, f_cs)
            FM ← AddChildFeatures(FM, c_s, f_cs, Arch150)
        end for
    end if
end for
```

those of FM_{Plug} by means of *requires* constraints. This mapping allows us to determine (1) which plugin provides a given architectural feature, and (2) which architectural features are provided by a given plugin.

2.4 Deriving FM_{Arch}

Semantics and Example. We now illustrate how we can derive FM_{Arch} using $FM_{Arch_{150}}$, FM_{Plug}, the mapping between FM_{Plug} and $FM_{Arch_{150}}$, and an operation called *projection* using the example of Fig. 4.

First FM_{Plug} and $FM_{Arch_{150}}$ are *aggregated* under a synthetic root $FtAggregation$ so that root features of input FMs are child-mandatory features of $FtAggregation$. The aggregation operation produces a new FM, called FM_{Full} (see Fig. 4). The propositional constraints that relate features of FM_{Plug} to features of $FM_{Arch_{150}}$ are also added to FM_{Full}.

Definition 1 (Projection) *The projection is a unary operation on FM written as* $\Pi_{ft_1,ft_2,...,ft_n}\ (FM_i)$ *where* $ft_1, ft_2, ..., ft_n$ *is a set of features. The result of a projection applied to an FM,* FM_i*, is a new FM,* FM_{proj}*, such that:* $[\![FM_{proj}]\!] = \{\ x \in [\![FM_i]\!] \mid x \cap \{ft_1, ft_2, ..., ft_n\}\ \}$

Second, we compute the projection (see Definition 1) of FM_{Full} onto the set of features of $FM_{Arch_{150}}$ (i.e., $\mathcal{F}_{FM_{Arch_{150}}} = \{Arch, Ar1, \ldots, Ar6\}$). The projection produces a new FM, called FM_{Arch} (see Fig. 4). Formally:

$$\Pi_{\mathcal{F}_{FM_{Arch_{150}}}}\ (FM_{Full}) = \ FM_{Arch}$$

In the example of Fig. 4, the relationship between $[\![FM_{Full}]\!]$ and $[\![FM_{Arch}]\!]$ truly holds. We can notice that one configuration of the original $FM_{Arch_{150}}$ has been removed, i.e., $[\![FM_{Arch_{150}}]\!] \setminus [\![FM_{Arch}]\!] = \{Ar1, Ar2, Ar3, Ar6, Arch\}$. The reason is that the projected FM_{Arch} contains an additional constraint $Ar3 \Rightarrow Ar5$, not originally present in $FM_{Arch_{150}}$. Similarly, the constraint $Ar4 \Rightarrow$

$Ar6$ (grey tint in Fig. 4) can been deduced but is redundant with $Ar3 \Rightarrow Ar5$. As we will see in the experiments (next Section), such deductions can dramatically reduce the set of configurations of $FM_{Arch_{150}}$.

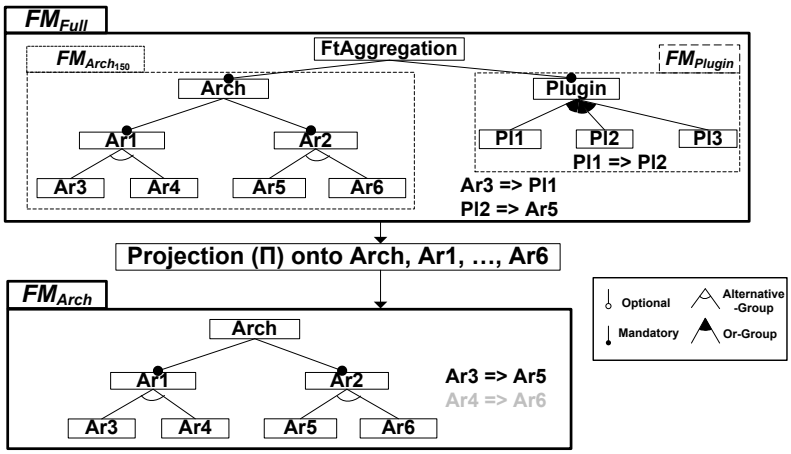


Fig. 4. Enforcing architectural FM using aggregation and projection.

Implementation of the projection. Our previous experience in the composition of FMs has shown that *syntactical* strategies have severe limitations to accurately represent the set of configurations expected, especially in the presence of cross-tree constraints [1]. The same observation applies for the projection operation so that reasoning directly at the *semantic* level is required. The key ideas of our implementation are to *i)* compute the propositional formula representing the projected set of configurations and then *ii)* reuse the reasoning techniques proposed in [11] to construct an FM from the propositional formula.

Formula Computation. For a projection $FM_{proj} = \Pi_{ft_1,ft_2,...,ft_n}(FM_i)$, the propositional formula corresponding to FM_{proj} is defined as follows[5]:

$$\phi_{proj} \equiv \exists\ ftx_1, ftx_2, \dots ftx_{m'}\ \phi_i$$

where $ftx_1, ftx_2, \dots ftx_{m'} \in\ (\mathcal{F}_{FM_i} \setminus \{ft_1, ft_2, \dots ft_m\}) =\ \mathcal{F}_{removed}$.

The propositional formula ϕ_{proj} is obtained from ϕ_i by *existentially quantifying* out variables in $\mathcal{F}_{removed}$. Intuitively, all occurrences of features that are not present in any configuration of FM_{proj} are removed by existential quantification in ϕ_i. The projection can be seen as a safe removal of a set of features (i.e., existential quantification removes a variable from a propositional formula without affecting its satisfiability).

We rely on Binary Decision Diagrams (BDDs) to compute ϕ_{proj} [8]. A BDD can be seen as a compact representation of a propositional formula. Our choice to use BDDs was driven by several elements: *i)* computing the existential quantification of BDDs and several logical operations can be performed in at most

[5] Let v be a Boolean variable occurring in ϕ. Then $\phi_{|_v}$ (resp. $\phi_{|_{\bar{v}}}$) is ϕ where variable v is assigned the value True (resp. False). Existential quantification is then defined as $\exists v\ \phi =_{def} \phi_{|_v} \vee \phi_{|_{\bar{v}}}$.

polynomial time with respect to the sizes of the BDDs involved [8]; *ii)* efficient optimized implementations of these operations are provided by off-the-shelf BDD libraries (e.g., we use the JavaBDD library [14]); *iii)* the techniques described in [11] (see below) also relies on a BDD-based implementation and *iv)* polynomial algorithms are available for computing valid domains or the counting of the number of products (see [7]).

From Formula to FM. We use the algorithm presented in [11] to transform ϕ_{proj} into an FM. More precisely, the algorithm builds a tree with additional nodes for feature groups that can be translated into a basic *feature diagram*. Importantly, the algorithm indicates parent-child relationships (i.e., mandatory or optional features) and Alternative- or Or-groups. The feature diagram, however, is an over approximation of the original formula in the sense that if we translate the synthesized feature diagram to a propositional formula, noted $\phi_{proj_{diagram}}$, then some valid assignments of $\phi_{proj_{diagram}}$ may be invalid in ϕ_{proj} (but any valid assignment of ϕ_{proj} is also valid in $\phi_{proj_{diagram}}$). We thus need to further constrain the feature diagram (as we did for the example of Fig. 4). We propose to decompose the FM FM_{proj} as a feature diagram, a set of require constraints and a set of other propositional constraints (that cut across the hierarchy of the diagram). We compute the set of require constraints by first computing the implication graph, noted I_{proj}, of the formula ϕ_{proj} over $ft_1, ft_2, ..., ft_n$. I_{proj} is a directed graph $G = (V, E)$ formally defined as:

$$V = \{ft_1, ft_2, ..., ft_n\} \qquad E = \{(f_i, f_j) \mid \phi_{proj} \wedge f_i \Rightarrow f_j\}$$

Then, the set of require constraints can be deduced by removing edges from I_{proj} being already expressed in the feature diagram (e.g., parent-child relations). Finally, if the feature diagram *plus* the require constraints are still an over approximation of ϕ_{proj} (it is not the case in Fig. 4), an additional formula $\phi_{proj_{other}}$ can be obtained so that $\phi_{proj} = \phi_{proj_{diagram}} \wedge \phi_{proj_{requires}} \wedge \phi_{proj_{other}}$ ($\phi_{proj_{requires}}$ being the logical conjunction of all require constraints of I_{proj}).

3 Refining the Architectural Feature Model: Experiments

We conduct an experiment to *i)* determine if the architectural FM designed by the SA[6], noted FM_{SA}, is *consistent* with the extracted FM FM_{Arch} (and vice-versa) ; *ii)* step-wise refine FM_{SA} based on the previous observations. We describe the techniques developed for the experiment and analyze the results.

3.1 Tool Support

For the experiment, we rely on FAMILIAR (*FeAture Model scrIpt Language for manIpulation and Automatic Reasoning*) [12,2], a domain-specific language dedicated to the management of FMs. FAMILIAR is an executable, textual language that supports manipulating and reasoning about FMs and is integrated in a comprehensive Eclipse-based environment (including graphical editors). We use FAMILIAR for two main purposes. Firstly, the extraction procedure generates FAMILIAR code to compute FM_{Arch}. Secondly, FAMILIAR provides the SA with a dedicated approach for easily manipulating FMs during the refinement process.

[6] Philippe Merle, co-author of this paper and principal developer of FraSCAti, plays the role of *software architect* (SA) in the experiment.

3.2 Experimental Results

Automatic Extraction. The $FM_{Arch_{150}}$ produced by the extraction procedure[7] contains 50 features while the FM_{Plug} contains 41 features. The aggregated FM, FM_{Full}, resulting from $FM_{Arch_{150}}$, FM_{Plug} and the bidirectional mapping contains 92 features and 158 cross-tree constraints.

We first verify some properties of FM_{Full}. By construction, we know that the projection of FM_{Full} onto $\mathcal{F}_{FM_{Arch_{150}}}$ is either a refactoring or a specialization[8] of $FM_{Arch_{150}}$ (see Definition 2).

Definition 2 (Specialization, Refactoring, Generalization, Arbitrary Edit) *Let f and g be two feature models. f is a specialization of g if $[\![f]\!] \subset [\![g]\!]$ f is a generalization of g if $[\![g]\!] \subset [\![f]\!]$ f is a refactoring of g if $[\![g]\!] = [\![f]\!]$ f is an arbitrary edit of g if f is neither a specialization, a generalization nor a refactoring of g.*

Our experiments reveal that FM_{Arch} is a specialization of $FM_{Arch_{150}}$. More precisely, $FM_{Arch_{150}}$ admits 13958643712 possible configurations ($\approx 10^{11}$), while FM_{Arch} represents 936576 distinct products ($\approx 10^6$). As expected, the projection technique significantly reduces the over approximation of $FM_{Arch_{150}}$.

To improve the understanding of the difference between two FMs, we use a *diff* operator, denoted as $FM_1 \oplus_{\setminus} FM_2 = FM_r$. The following defines the semantics of this operator.

$$[\![FM_1]\!] \setminus [\![FM_2]\!] = \{x \in [\![FM_1]\!] \mid x \notin [\![FM_2]\!]\} = [\![FM_r]\!]$$

The formula ϕ_r of FM_r is used to reason about properties of FM_r (e.g., satisfiability) and is computed as follows:

$$\phi_r = (\phi_{FM_1} \wedge not(\mathcal{F}_{FM_2} \setminus \mathcal{F}_{FM_1})) \wedge \neg(\phi_{FM_2} \wedge not(\mathcal{F}_{FM_1} \setminus \mathcal{F}_{FM_2}))$$

For example, determining the kind of relationship between two FMs (see Definition 2) can be done by reusing the algorithm presented in [25] or by using the diff operator (see Definition 3).

Definition 3 (Diff and Specialization/Refactoring) *Let f and g be FMs. f is a specialization or a refactoring of g if $(f \oplus_{\setminus} g)$ has no valid configurations since $[\![f]\!] \subseteq [\![g]\!]$ is equivalent to $[\![f]\!] \setminus [\![g]\!] = \emptyset$.*

Moreover, the diff operator can compute the difference (if any) between two FMs in terms of set of configurations. In particular, we can compute the cardinality of this set. For example, we correctly check the following relationship[9] using the tool support: $|FM_{Arch_{150}}| - |FM_{Arch_{150}} \oplus_{\setminus} FM_{Arch}| = |FM_{Arch}|$.

Techniques for Refining Architectural FM. The goal of the reverse engineering process is to elaborate an FM which *accurately* represents the valid combinations of features of the SPL architecture. The absence of a *ground truth* FM (i.e., an FM for which we are certain that each combination of features is

[7] material, details and results of the experiments are accessible at `https://nyx.unice.fr/projects/familiar/wiki/ArchFm`

[8] We rely on the terminology used in [25].

[9] where $|FM_i|$ denotes the number of configurations of FM_i, i.e., $|FM_i| = |[\![FM_i]\!]|$

supported by the SPL architecture) makes uncertain the accuracy of variability specification expressed in FM_{Arch} as well as in FM_{SA}. It is the role of the SA to determine if the variability choices in FM_{SA} (resp. FM_{Arch}) are coherent regarding FM_{Arch} (resp. FM_{SA}). In case variability choices are conflicting, the SA can *refine* the architectural FM.

We now report the problems encountered when reasoning about the relationship between FM_{Arch} and FM_{SA}. We also describe the advanced techniques we developed to assist the SA.

Reconciling FM_{Arch} ***and*** FM_{SA}**.** A first obstacle concerned the need to *reconcile* FM_{Arch} and FM_{SA} (see Fig. 5). Both FMs come from difference sources and a preliminary work is needed before reasoning about their relationship. Firstly, the *vocabulary* (i.e., names of features) used differs in both FMs and should be aligned consequently. To resolve this issue, we rely on string matching techniques (i.e., Levenshtein distance) to automatically identify features of FM_{Arch} that correspond to features of FM_{SA}. Then a renaming is applied on all corresponding features in FM_{Arch}. As an example, "MMFraSCAti" of FM_{SA} has been identified to correspond to "sca_metamodel_frascati" of FM_{Arch} and after the renaming FM_{Arch} contains the feature "MMFraSCAti". We automatically detect 32 features. The SA manually specifies the correspondence for 5 features in which the automated detection does not succeed (e.g., "MembraneFactory" corresponding to "fractal_bootstrap_class_providers"). Secondly, *granularity* details differ, (i.e., some features in one FM are not present in the other FM): FM_{SA} only contains 39 features whereas FM_{Arch} contains 50 features.

In FM_{SA} ***but not in*** FM_{Arch}**.** Two exclusive features $Felix$ and $Equinox$ are present in FM_{SA} but not in FM_{Arch}. We also observed that the two features are present in FM_{Plug} but not in $FM_{Arch_{150}}$ (and hence not in FM_{Arch}). A discussion with the SA reveals that these two plugins do not explicitly define architecture fragments in SCA. As a consequence, this variability point can simply not be identified in the architecture by the automatic extraction procedure.

In FM_{Arch} ***but not in*** FM_{SA}**.** We identified 13 features that are present in FM_{Arch} but not in FM_{SA}. Among others, two Metamodels used by the SCA parser, three Bindings, two SCA properties, two Implementations and one Interface were missing. Given the complexity of the FraSCAti project, this is not surprising that the SA forgets some features. Hence, for most of the features, the SA considers the missing features as relevant and thus adds them in FM_{SA}. For one of the missing feature, "sca_interface_java", the SA reveals that he *intentionally* ignored it in FM_{SA}, arguing that it is a mandatory feature (i.e., every FraSCAti configuration has a Java interface) and that his focus was on variability rather than commonality. We indeed verify the mandatory nature of "sca_interface_java" in FM_{Arch}. Nevertheless, the SA decides to add "sca_interface_java" in FM_{SA}. Similarly, two first-level mandatory features, "binding_factory" and "services", were missing in FM_{SA}. The SA intentionally did not include the two features since they do not convey any further variation points, but he decides to edit FM_{SA} by adding those features. Another example concerns a feature of FM_{Arch}, "juliac", that adds unnecessary details (so that

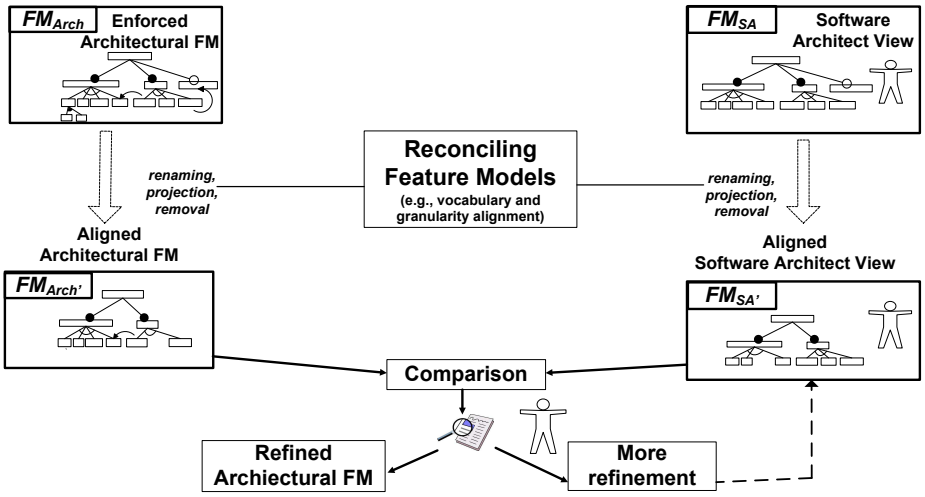


Fig. 5. Process for Refining FM_{Arch}

the way features are organized in FM_{SA} and FM_{Arch} slightly differ). Here the SA decides to remove "juliac" by projection.

Reasoning about FM_{Arch} ***and*** FM_{SA}**.** At this step, we can *compare* FM_{Arch} and FM_{SA}. A first comparison is to determine the kind of relationship between FM_{Arch} and FM_{SA} (see Definition 2). We obtain an arbitrary edit, that is, some configurations of FM_{Arch} are not valid in FM_{SA} (and vice-versa). To go further, we use the diff operator (see Definition 3) and the merge in intersection mode (see [1]). We enumerate and count the unique configurations of FM_{Arch} and FM_{SA} as well as the common configurations of FM_{Arch} and FM_{SA}. Nevertheless, the techniques appear to be insufficient to really understand the difference between the two FMs. Intuitively, we need to identify more *local* differences. A first technique is to compare the variability associated to features of FM_{Arch} and FM_{SA} that have the same name. In particular, we detect that *i)* four features are optional in FM_{Arch} but mandatory in FM_{SA} and *ii)* three sets of features belong to Or-groups in FM_{Arch} whereas in FM_{SA}, the features are all optional. A second technique is to compute the intersection and the difference of the sets of require constraints in FM_{Arch} and FM_{SA} (based on their implication graphs, see page 10).

Step-wise Refinement of FM_{SA}**.** The comparison techniques have been reiterated until having a satisfying architectural FM. Based on the comparison results, the SA have had several attitudes. Firstly, he used FM_{Arch} to *verify* the coherence of his original variability specification in FM_{SA}. Secondly, he considered that some variability decisions in FM_{SA} are correct despite their differences with FM_{Arch}. The SA imposed five variability decisions not identified by the extraction procedure. Thirdly, he edits FM_{SA}, for example, by adding some constraints only present in FM_{Arch} or by setting optional a feature originally mandatory. The extracted FM notably identifies nine "obvious" constraints not expressed in FM_{SA} and allows the SA to incrementally *correct* FM_{Arch}.

3.3 Lessons Learned

The FraSCAti experiment provide us with interesting insights into the reverse engineering of architectural FMs. First, the gap between FM_{SA} and FM_{Arch} appears to be manageable, due to an important similarity between the two FMs. However, it remains helpful to assist the SA with automated support, in particular, to establish correspondences between features of the two FMs. The most time-consuming task was to reconcile the granularity levels of both FMs. For this specific activity, tool supported, advanced techniques, such as the safe removal of a feature by projection, are not desirable but mandatory (i.e., basic manual edits of FMs are not sufficient).

Second, our extraction procedure (Section 2) yields very promising results. It recovers most of the variability expressed in FM_{SA} and encourages the SA to correct his initial model. A manual checking of the five variability decisions imposed by the SA shows that the extraction is not faulty. It correctly reproduces the information as described in the software artefacts of the project.

Third, the SA *knowledge* is required *i)* to scope the SPL architecture (e.g., by restricting the set of configurations of the extracted FM), especially when software artefacts do not correctly document the variability of the system and *ii)* to control the accuracy of the automated procedure. An open issue is then to provide a mechanism and a systematic process to reuse the SA knowledge, for example, for another version of the architectural FM of FraSCAti.

4 Related Work

Despite the importance of variability management in software engineering in general, and in software architectures in particular [5], the problem of reverse engineering the variability of *existing* systems has definitely not received sufficient attention from the research community. While our work takes an architectural perspective, the other existing approaches in the field consider different input artifacts including legacy system documentation [15] or textual requirements [3,26]. In their recent work, She *et al.* [22] propose a reverse engineering approach combining two distinct sources of information: textual feature descriptions and feature dependencies. Our approach also benefits from the combination of two (other) sources of information, namely plugin dependencies and architecture fragments. They mostly focus on the retrieval of the feature diagram (heuristics for identifying the most likely parent feature candidates of each feature, group detection, etc.) and assume that the set of valid configurations is correctly restituted, which is clearly not the case in our work. We also support the identification of feature groups (based on architectural extension points), of the right parent feature of each feature (based on architectural hierarchy) and of inter-feature dependencies (through projection of plugin dependencies).

The FM analysis and reasoning techniques used in this paper reuse and extend previous work in SPL and requirement engineering [7]. Metzger *et al.* [18] propose an approach to cross-checking *product-line* variability and *software* variability models, thus assuming that such models (or views) are available. Our approach is complementary since it allows to recover the *actually supported* variability of a software system, and since it involves the cross-analysis of architec-

tural and plugin FM. One of the key component and original contribution of our work is the projection operator we have defined and realized (see Section 2). Lopez and Eyged [16] address a related problem in the context of safe composition by checking the consistency of multi-view variability models. In particular, they check whether an FM developed by a domain expert is a specialization or a refactoring of an FM representing the variability of multiple models. Thüm *et al.* [25] reason on the nature of FM edits, and provide a classification that we rely on when comparing the extracted FM with the software architect view. The diff operator provides a compact representation of all added and removed configurations and is an extension of their work. As we have shown, reasoning about the relationship of two FMs is inappropriate until FMs are not reconciled, i.e., pre-directives (e.g., safe removal of unnecessary details) have to be applied before. The comparison operator developed in [25] considers only the *concrete* features (i.e., leaves) but our experience shows that it is important to consider *all* features of the two FMs (e.g., a non leaf feature can be concrete). Benavides et al. compared the performance of CSP, SAT solvers and BDD solvers for some reasoning operations on FMs [7]. As future work, we will investigate the use of SAT or CSP solvers to realize the diff/projection operators. A comparison with BDD-based implementations is planned to determine the most scalable solution. Another research direction is to consider feature *attributes* [7], for example, to model quality attributes of the FraSCAti architecture.

5 Conclusion

Variability management is of crucial importance in the management of large families of software systems. While feature models have long been recognized as expressive means to compactly represent software variability from different perspectives, building one of them for a large system is a complex, time-consuming and error-prone activity. In this paper, we presented a tool-supported approach to reverse engineer software variability from an architectural perspective. The reverse engineering process involves the automatically supported extraction, aggregation, alignment and projection of architectural feature models. It has the merit of combining several sources of information, namely software architecture, plugin dependencies and software architect knowledge. We successfully evaluated the proposed approach when applied to FraSCAti, a large and highly configurable plugin-based system. We showed that our automated procedures allow for producing both correct and useful results, thereby significantly reducing manual effort. We learned, however, that fully automating the process is not realistic nor desirable, since the intervention of the software architect remains highly beneficial. The ongoing evolution of the FraSCAti project will bring us an opportunity to study how to reuse the accumulated knowledge of the software architect and provide a validated, systematic process for extracting architectural variability models.

References

1. M. Acher, P. Collet, P. Lahire, and R. France. Comparing Approaches to Implement Feature Model Composition. In *ECMFA'10*, volume 6138 of *LNCS*, pages 3–19, 2010.

2. M. Acher, P. Collet, P. Lahire, and R. France. A Domain-Specific Language for Managing Feature Models. In *SAC'11*. PL Track, ACM, 2011.
3. V. Alves, C. Schwanninger, L. Barbosa, A. Rashid, P. Sawyer, P. Rayson, C. Pohl, and A. Rummler. An exploratory study of information retrieval techniques in domain analysis. In *SPLC'08*, pages 67–76. IEEE, 2008.
4. S. Apel and C. Kästner. An overview of feature-oriented software development. *Journal of Object Technology (JOT)*, 8(5):49–84, July/August 2009.
5. F. Bachmann and L. Bass. Managing variability in software architectures. *SIGSOFT Softw. Eng. Notes*, 26:126–132, May 2001.
6. D. Batory. Feature models, grammars, and propositional formulas. In *Proc. of SPLC'2005*, volume 3714 of *LNCS*, pages 7–20. Springer, 2005.
7. D. Benavides, S. Segura, and A. Ruiz-Cortes. Automated Analysis of Feature Models 20 years Later: a Literature Review. *Information Systems*, 2010.
8. K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a bdd package. In *DAC'90*, pages 40–45. ACM, 1990.
9. P. Clements and L. M. Northrop. *Software Product Lines : Practices and Patterns*. Addison-Wesley Professional, 2001.
10. K. Czarnecki and K. Pietroszek. Verifying feature-based model templates against well-formedness ocl constraints. In *GPCE'06*, pages 211–220. ACM, 2006.
11. K. Czarnecki and A. Wasowski. Feature diagrams and logics: There and back again. In *SPLC'07*, pages 23–34, 2007.
12. FAMILIAR. `http://nyx.unice.fr/projects/familiar/`.
13. FraSCAti. `https://wiki.ow2.org/frascati/Wiki.jsp?page=Main`.
14. JavaBDD. `http://javabdd.sourceforge.net/index.html`.
15. I. John. Capturing product line information from legacy user documentation. In *Software Product Lines*, pages 127–159. Springer, 2006.
16. R. E. Lopez-Herrejon and A. Egyed. Detecting inconsistencies in multi-view models with variability. In *ECMFA'10*, pages 217–232, 2010.
17. R. E. Lopez-Herrejon and A. Egyed. On the need of safe software product line architectures. In *ECSA'10*, volume 6285 of *LNCS*, pages 493–496. Springer, 2010.
18. A. Metzger, K. Pohl, P. Heymans, P.-Y. Schobbens, and G. Saval. Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. In *RE'07*, pages 243–253, 2007.
19. C. A. Parra, A. Cleve, X. Blanc, and L. Duchien. Feature-based composition of software architectures. In *ECSA'10*, volume 6285 of *LNCS*, pages 230–245. Springer, 2010.
20. P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps. Generic semantics of feature diagrams. *Comput. Netw.*, 51(2):456–479, 2007.
21. L. Seinturier, P. Merle, D. Fournier, N. Dolet, V. Schiavoni, and J.-B. Stefani. Reconfigurable SCA Applications with the FraSCAti Platform. In *SCC'09*, pages 268–275. IEEE, 2009.
22. S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. Reverse engineering feature models. In *ICSE'11*, 2011. to appear.
23. SCA standard. `http://www.osoa.org/`.
24. M. Svahnberg, J. van Gurp, and J. Bosch. A taxonomy of variability realization techniques: Research articles. *Softw. Pract. Exper.*, 35(8):705–754, 2005.
25. T. Thüm, D. Batory, and C. Kästner. Reasoning about edits to feature models. In *ICSE'09*, pages 254–264. IEEE, 2009.
26. N. Weston, R. Chitchyan, and A. Rashid. A framework for constructing semantically composable feature models from natural language requirements. In *SPLC'09*, volume 446 of *ICPS*, pages 211–220. ACM, 2009.