SoSyM manuscript No.
(will be inserted by the editor)

Extraction and Evolution of Architectural Variability Models in Plugin-based Systems

Mathieu Acher[1], Anthony Cleve[2], Philippe Collet[3], Philippe Merle[4], Laurence Duchien[4], Philippe Lahire[3]

[1] Irisa, Inria and University of Rennes 1, France
e-mail: mathieu.acher@irisa.fr
[2] PReCISE Research Centre, University of Namur, Belgium
e-mail: acl@info.fundp.ac.be
[3] Université Nice Sophia Antipolis - I3S (CNRS UMR 7271), France
e-mail: {collet,lahire}@i3s.unice.fr
[4] Inria Lille - Nord Europe, University of Lille 1 - CNRS UMR 8022, France
e-mail: {philippe.merle,laurence.duchien}@inria.fr

Revised: 30th may 2013

Abstract Variability management is a key issue when building and evolving software-intensive systems, making it possible to extend, configure, customize and adapt such systems to customers' needs and specific deployment contexts. A wide form of variability can be found in extensible software systems, typically built on top of plugin-based architectures that offer a (large) number of configuration options through plugins. In an ideal world, a software architect should be able to generate a system variant on-demand, corresponding to a particular assembly of plugins. To this end, the variation points and constraints between architectural elements should be properly modeled and maintained over time (i.e., for each version of an architecture). A crucial, yet error-prone and time-consuming, task for a software architect is to build an accurate representation of the variability of an architecture, in order to prevent unsafe architectural variants and reach the highest possible level of flexibility. In this article, we propose a reverse engineering process for producing a variability model (i.e., a feature model) of a plugin-based architecture. We develop automated techniques to extract and combine different variability descriptions, including a hierarchical software architecture model, a plugin dependency model and the software architect knowledge. By computing and reasoning about differences between versions of architectural feature models, software architect can control both the variability extraction and evolution processes. The proposed approach has been applied to a representative, large-scale plugin-based system (FraSCAti), considering different versions of its architecture. We report on our experience in this context.

Send offprint requests to:

1 Introduction

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. In order to tame such a complexity, *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*" [17]. 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 process usually starts with an analysis of the domain to identify commonalities and variabilities between the members of the SPL. It is common to express SPL variability in terms of *features*, which are domain abstractions relevant to stakeholders. For this purpose, a *Feature Model* (FM) is generally used to compactly define all features in an SPL as well as their valid combinations [61,21].

Besides large software systems are now commonly organized around a more or less explicit architecture, which defines entities, their properties and relationships. When SPL engineering principles are followed from the start, it is feasible to manage variability through one or more *architectural* FMs and then associate them to the system architecture [52]. The major architectural variations are then mapped to given features, allowing for automated composition of architectural elements when features are selected to configure a particular software product from the line. A resulting property of crucial importance is to guarantee that the variability is not only preserved but also kept consistent across all artefacts [20,12,40].

In many cases, however, one has to deal with (*legacy*) software systems not initially designed as SPLs [66,29,73,3,55]. When the system becomes more complex, with many configuration and extension points, its variability must be handled according to SPL techniques. In this context, the task of building an architectural FM is very arduous for software architects. They typically have to deal with lots of plugins (usual customizations of the Eclipse IDE are made with several hundreds of plugins, corresponding to dozens of high-level features [27, 53]), for which *safe composition* is the topmost requirement [40].

It is then necessary to recover a consistent FM from the actual architecture. On a large scale both automatic extraction from existing parts and the architect knowledge should ideally be combined to achieve this goal. In particular, a software architect should be able to determine whether her (high-level) representation complies with an automatically extracted model, and to what extent they differ from each other (e.g., in the style of reflexion models [49]). Moreover, since the software architecture and functionalities are naturally evolving over time, it is also necessary to ensure that an architectural FM is maintained consistent with these changes. In the case of modern dynamic software architectures, which are based on plugins, these modifications can be very complex to handle, especially in presence of hidden dependencies between (different versions of) plugins. In this context, evolving the architectural FM along the modified architecture is tedious. It is therefore needed to reproduce the extraction process and to reason on the new architectural FM and on its differences.

In this article, we present a comprehensive, tool supported process for reverse engineering and evolving architectural FMs. We show how techniques for FM slicing [5] and differencing [7] can be adapted and applied in the particular context of architectural FM extraction, analysis and evolution. Specifically, we develop automated techniques to extract and combine different variability descriptions of a software architecture, integrating the hierarchical decomposition of the architecture and inter-plugin dependencies. The basic idea is that variability and technical constraints of the plugin dependencies are *projected* onto an architectural model. After the extraction, alignment and reasoning techniques are applied to integrate the architect knowledge and reinforce the extracted FM. In addition to this extraction process, previously presented in [2], we also show how the process can be reiterated when the architecture evolves. This notably enables the architect to re-integrate his/her knowledge and to reason about the differences between two successive architectural FMs.

Furthermore we evaluate our proposal on different versions of FraSCAti. Overall the results show the software architect increases the *quality* of architectural FMs (i.e., better specifying variability and thus avoiding some unsafe configurations) compared to an FM that is manually designed or that does not integrate all variability descriptions of the system. Furthermore the architectural FM takes into account both the software architect viewpoint and the variability actually supported by the system. Without the FM management support exposed in the article, obtaining similar results would not be possible.

The target audience of this article is threefold. It is first relevant to software systems' modelers, providing insights on the kind of models that can be built and evolved to capture architectural variability. Software architects may also be interested in the experience report on a representative, non-trivial and still evolving plugin-based system. Finally the third target audience regroups researchers and practitioners interested in software variability extraction and management.

The remainder of this article is organized as follows. In Section 2, we give some background on the FraSCAti case study and foundations of feature modeling. Section 3 identifies two key challenges when managing the architectural variability of a plugin-based system. We also outline our contributions to face those two challenges. In Section 4, we describe in detail the automated extraction process that we have developed. The extraction is tool-supported and processes various documents until generating an architectural FM. Section 5 shows how the process is completed by refinement steps that enable the architect to compare and integrate her knowledge, with the aim to obtain a consistent architectural FM. In Section 6, we describe the tools supporting our approach. The overall process is then validated in Section 7, which presents its application to the FraSCAti architecture. We also discuss threats to validity. A related work discussion is provided in Section 8 including a comparison of the proposed FM management support with existing techniques. In Section 9 we summarize our contributions, discuss some lessons learned, and anticipate future work.

2 Background

2.1 The FraSCAti Plugin-based System Case Study

We motivate and illustrate our proposal on a case study related to the FraSCAti platform [47], an open source implementation of the OASIS's Service Component Architecture (SCA) standard [50]. SCA is a technology-agnostic component-based standard for building distributed composite service-oriented applications mixing various programming languages and frameworks (e.g., Java, C, C++, WS-BPEL, Spring Framework) for implementing business components, various interface definition languages (e.g., WSDL, Java) for describing business services, and various network communication protocols (e.g., Web Service, Java Messaging Service) for interconnecting distributed applications.

Main SCA component-based concepts are quite generic and present in numerous other component models: a *composite* is a component composed of a set of components, a *component* encapsulates a business logic implemented with a programming language/framework, a *service* and a *reference* are named interfaces respectively provided/required by a component, an *interface* is a set of methods implemented or used by a component, a *binding* explains how both service and reference are accessible via a network communication protocol, and a *wire* connects a source reference to a target service.

Started in 2007, the development of FraSCAti begun with a framework based on a basic implementation

of the standard, that has then been incrementally enhanced. After six major releases, it now supports several SCA specifications (Assembly Model, Java Common Annotations & APIs, Java Component Implementation, Spring Component Implementation, WS-BPEL Client & Implementation, Web Services Binding, JMS Binding, Transaction Policy), and provides a set of extensions to the standard, including component implementation types (SCA composite, Java, EJB, WS-BPEL, C, Spring, Fractal, OSGi, Scala, and BeanShell, FScript, Groovy, JavaScript, JRuby, Jython, XQuery, Velocity scripting languages), binding implementation types (SOAP, JMS, Java RMI, HTTP, REST, JSON-RPC, JNA, UPnP, OSGi, JGroups), interface description types (WSDL, Java, UPnP, C headers), and runtime APIs for component introspection and reconfiguration [64,65].

As its capabilities grew between releases, FraSCAti has itself been refactored and completely architected as an SCA-based application, i.e., an assembly of SCA components. The FraSCAti architecture is composed of three main SCA composites:

- The SCA parser is responsible to load business SCA composite files into memory. As the SCA composite language is extensible, its grammar is described by several meta-models (MM). Then FraSCAti supports various SCA meta-models (e.g., $MMFrascati$, $MMTuscany$).
- The Assembly Factory is responsible to check SCA composites and orchestrate their instantiation. The assembly factory is composed of several plugins for dealing with the various forms of component implementations, interface definition languages, and service bindings (e.g., *rest*, *http*).
- The Component Factory is in charge of instantiating SCA components. This factory generates and compiles Java code for component containers. This factory has two plugins for supported Java compilers (i.e., $JDK6$ and JDT).

Thanks to its new component-based architecture, different variants of FraSCAti can be built in order to meet various application requirements and target system constraints. Each SCA application running on FraSCAti could have different requirements in terms of SOA features like supporting SOAP, WSDL, WS-BPEL, REST, OSGi, JMS. All these SOA features are implemented as SCA components which are plugged to the FraSCAti architecture. Then, application developers could select all the FraSCAti plugins required for their applications. Orthogonally, the target system on which applications are deployed could impose some constraints. For instance, FraSCAti applications could be deployed on standalone Java Runtime Environments (JRE), Web application servers, or OSGi gateways. Each of these target environments is supported by a specific pluggable FraSCAti component. FraSCAti could require to compile Java code on the fly, then FraSCAti requires an embedded Java compiler. FraSCAti supports two distinct Java compilers: The standard JDK6 compiler and the Eclipse JDT compiler. FraSCAti plugins could have dependencies, e.g., the REST binding plugin requires the FraSCAti meta-model while the HTTP binding plugin requires the Tuscany meta-model. These FraSCAti plugin dependencies are captured via Apache Maven[1] XML-based descriptors.

FraSCAti version 1.5 contains around 60 plugins for a total of around 250.000 lines of code. So, FraSCAti is representative of a large plugin-based system, i.e., a system composed of plugins, each of which is implemented as a set of SCA components that adds specific abilities to FraSCAti.

With all these capabilities, the FraSCAti platform has become highly (re-)configurable in many parts of its own architecture. It exposes a larger number of extensions that can be activated throughout the platform, creating numerous variants of a FraSCAti deployment. It then became obvious to FraSCAti technical leaders that the variability[2] of the platform should be more systematically managed as an SPL in order to better drive and control its evolution.

2.2 Foundations of Feature Modeling

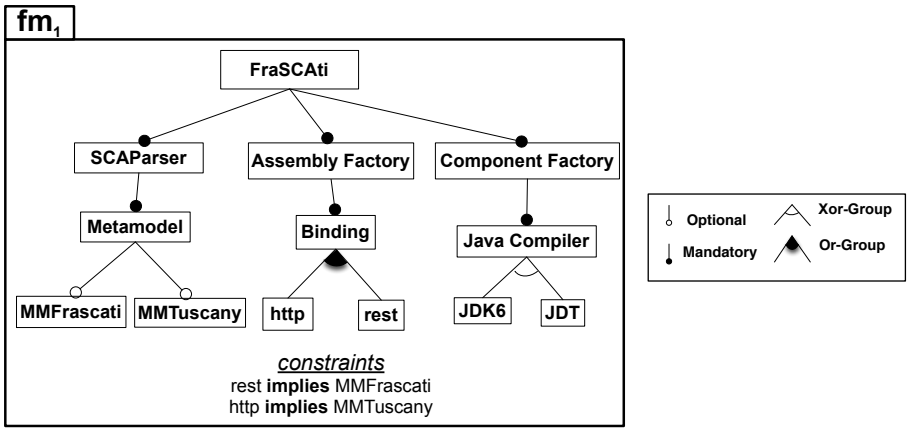
Variability modeling is a central activity in SPL engineering. We choose to rely on a particular kind of variability model, *Feature Models* (FMs), based on their wide adoption, the existence of formal semantics, reasoning techniques and tool support [61,14]. FMs compactly represent product commonalities and variabilities in terms of features [20,39,12]. The FMs that we consider all along this article usually express *architectural* variability, meaning that FMs are devoted to the modeling of the variation points (and their relationships) in a given architecture.

An FM hierarchically structures features into multiple levels of detail. The *hierarchy* of an FM is represented by a rooted tree composed of a finite set of features and a finite set of edges (edges represent top-down hierarchical decomposition of features, i.e., parent-child relations between them). As an example, Fig. 1(a) shows a excerpt of the *architectural FM* of FraSCAti as described in the previous section.

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*, *exclusive* (e.g., $JDK6$ and

[1] Maven (`http://maven.apache.org/`) is a software tool for managing a project's build, reporting and documentation

[2] We use here the term *variability* as in the definition from [67]: "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."



(a) an architectural FM (simplified from our case study)

$[\![fm_1]\!] = \{$
$C \cup \{JDT, http, MMTuscany\},$
$C \cup \{MMFraSCAti, JDK6, http, MMTuscany\},$
$C \cup \{MMFraSCAti, JDK6, rest\},$
$C \cup \{MMFraSCAti, JDK6, rest, MMTuscany\},$
$C \cup \{MMFraSCAti, JDT, rest\},$
$C \cup \{MMFraSCAti, JDK6, http, rest, MMTuscany\},$
$C \cup \{MMFraSCAti, JDT, http, MMTuscany\},$
$C \cup \{MMFraSCAti, JDT, rest, MMTuscany\},$
$C \cup \{MMFraSCAti, JDT, http, rest, MMTuscany\},$
$C \cup \{JDK6, http, MMTuscany\}$
$\}$
with
$C = \{FraSCAti, SCAParser, AssemblyFactory,$
$ComponentFactory, Metamodel, Binding,$
$JavaCompiler\}$

(b) corresponding set of configurations

$\phi_1 = FraSCAti$
$\wedge\ FraSCAti \Leftrightarrow AssemblyFactory$
$\wedge\ FraSCAti \Leftrightarrow ComponentFactory$
$\wedge\ FraSCAti \Leftrightarrow SCAParser$
$\wedge\ SCAParser \Leftrightarrow Metamodel$
$\wedge\ AssemblyFactory \Leftrightarrow Binding$
$\wedge\ ComponentFactory \Leftrightarrow JavaCompiler$
$\wedge\ JavaCompiler \Rightarrow JDK6 \vee\ JDT$
$\wedge \neg\ JDK6 \vee\ \neg JDT$
$\wedge\ MMFrascati \Rightarrow Metamodel$
$\wedge\ MMTuscany \Rightarrow Metamodel$
$\wedge\ http \Rightarrow Binding$
$\wedge\ rest \Rightarrow Binding$
$\wedge\ Binding \Rightarrow rest \vee http$
$\wedge\ rest \Rightarrow MMFrascati$
$\wedge\ http \Rightarrow MMTuscany$

(c) corresponding propositional formula

Fig. 1 Feature model, set of configurations and propositional logic encoding

JDT form an *Alternative*-group), or *inclusive* (e.g., *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. $JDK6$ and JDT are mutually exclusive and cannot be selected at the same time.

The terms FM and *feature diagram* are employed in the literature, usually to denote the same concept. In this article, we consider that a feature diagram (see Definition 1) includes a feature hierarchy (tree), a set of feature groups, as well as human readable constraints (implies, excludes).

Definition 1 (Feature Diagram) *A feature diagram $FD = \langle G, r, E_{MAND}, G_{XOR}, G_{OR}, I, EX\rangle$ is defined as follows:*

- *$G = (\mathcal{F}, E, r)$ is a rooted tree where $\mathcal{F}$ is a finite set of features, $E \subseteq \mathcal{F} \times \mathcal{F}$ is a finite set of edges and $r \in \mathcal{F}$ is the root feature;*
- *$E_{MAND} \subseteq E$ is a set of edges that defines mandatory features with their parents;*
- *$G_{XOR} \subseteq \mathcal{P}(\mathcal{F}) \times \mathcal{F}$ and $G_{OR} \subseteq \mathcal{P}(\mathcal{F}) \times \mathcal{F}$ define feature groups and are sets of pairs of child features together with their common parent feature;*
- *a set of implies constraints I whose form is $A \Rightarrow B$ and a set of excludes constraints EX whose form is $A \Rightarrow \neg B$ ($A \in \mathcal{F}$ and $B \in \mathcal{F}$).*

Features that are neither mandatory features nor involved in a feature group are optional features. A parent feature can have several feature groups but a feature must belong to only one feature group.

Similarly to [66], we consider that an FM is composed of a feature diagram *plus* a propositional formula (see Definition 2).

Definition 2 (Feature Model) *A feature model FM is a tuple $\langle FD, \psi_{cst}\rangle$, where FD is a feature diagram and ψ_{cst} is a propositional formula over the set of features $\mathcal{F}$.*

Definition 3 (Configuration semantics) *A configuration of a feature model fm_i is defined as a set of selected features $c = \{f_1, f_2, \ldots, f_m\} \subseteq \mathcal{F}_i$. $[\![fm_i]\!]$ denotes the set of valid configurations of the feature model fm_i and is thus a set of sets of features. We note ϕ_i the propositional formula of fm_i.*

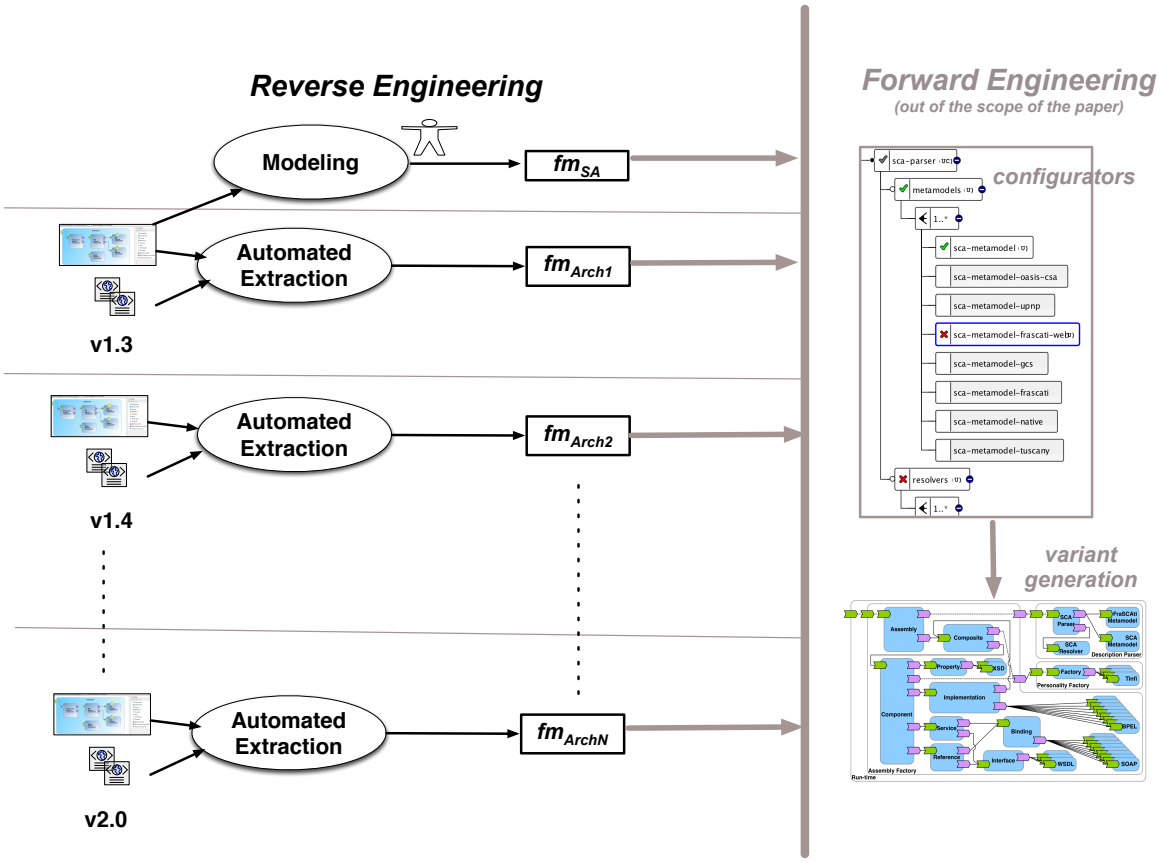


Fig. 2 Extraction and evolution of architectural variability

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 [21]. Fig. 1(c) 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 [14]). In particular, if an assignment to its Boolean variables is satisfiable, then the selection/deselection of the corresponding features respects the rules evoked above.

We chose the formalism of FMs in its Boolean (and most popular [14]) form since the expressiveness fits our needs: variability sources encountered in FraSCAti can be logically translated into such FMs.

3 On Architectural Variability Management

Several software artefacts (SCA composite files, Maven descriptors, informal documents) describe the architecture of FraSCAti, but the supported variability is not made explicit. Moreover, the raise of complexity in its maintenance and evolution led its software architect to control in a more systematic way the variability, based on product line principles.

To this aim, we needed to extract the supported variability from its actual architecture, and to reason about this variability when evolving the architecture, both for refactoring the system as an SPL or for developing new functionalities. We thus faced two challenges: a first one related to the extraction of architectural variability models; a second one about the management of the evolution of the FraSCAti architecture and of the corresponding models.

Challenge 1: *Extraction of Architectural FMs*

A first and essential step is therefore to identify and represent the variability of a system, including complex constraints between architectural elements. As exposed in Fig. 2, a variability model (i.e., an FM) can be exploited afterwards to pilot a configuration or derive specific variants[3].

Unfortunately, the task of manually creating the architectural FM is daunting, time-consuming and error-prone, requiring substantial effort from the *software architect (SA)*. In this case, as in all large-scale architectures, it is very difficult for an architect to guarantee that the resulting FM is consistent with the architecture. The scope defined by the FM should not be too

[3] The exploitation of the FM in a forward engineering process is out of the scope of this paper. Numerous generative techniques have been developed (see, e.g., [18,52,30]) and we plan to reuse them in our ongoing effort for re-engineering FraSCAti.

large (otherwise some unsafe compositions of the architectural elements are allowed) or too narrow (otherwise it translates as a lack of architectural flexibility). A *loose* FM simply hampers the systematic and consistent application of the SPL process, since the property of safe composition does not hold, allowing feature configurations that are not realizable by the architecture [48,68, 20,33,40]. In FraSCAti, an example of such a loose FM might be an FM similar to fm_1 in Fig. 1(a) but without the constraints related $REST$ and $HTTP$ technologies to specific metamodels. In this case, this would allow unsafe configurations involving some *rest* binding with the $MMTuscany$ metamodel, which are not realizable by the FraSCAti architecture.

Automatic extraction 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. In the general case, both automatic extraction from existing parts and the architect domain-specific knowledge should be ideally combined to achieve this goal.

Challenge 2: *Evolution of Architectural FMs*

As the software architectures, its elements, dynamic plugins and their relations, naturally evolve, the second challenge consists in mastering the evolution of the architectures and their variability. Software architects have to supervise and control that the evolution of the architectural FM is correct. In particular, the variability information and constraints should still be conformant with the SA knowledge or with previous versions of an architectural FM.

Several factors make this task tedious and very complex. The dynamicity of the software architectures, like in the FraSCAti case, leads to many hidden dependencies between plugins, especially when one handles evolving versions of plugins. Consequently, it is not possible to evolve the architectural FM directly and then check its consistency with the modified architecture and plugins, as this process would be very cumbersome, multiplying the changes on the FM without any real guidance.

We see this challenge as related to the first one, as the most appropriate way to tackle it is to reproduce the extraction process and to reason on the old and new architectural FMs. For our case study, the extraction/reconciliation process has to be reiterated on different versions of FraSCAti, and then, fine-grained differences between two successive architecture FM should ideally be presented to the software architect.

Overview of Our Proposal. Several sources of information can be considered when building an architectural variability model of a plugin-based system. In the case of FraSCAti, there are three possible sources. They either provide the adequate level of abstraction to manage the architecture or only focus on variability aspects. But none of them supports both and therefore are not sufficient alone to comprehensively address the **Challenge 1**:

- *the architectural model* restitutes the set of elements needed to reason about the software system and the (hierarchical) relations among them. But it usually does not contain any variability information and logical constraints between the elements.
- *the plugin dependencies* specify variation points and their logical dependencies actually supported by the architecture. But they do not reflect the architecture of the system and do not offer an adequate level of abstraction.
- *the software architect knowledge* can introduce accidental complexity (especially regarding variability) and does not necessarily reflect the software architecture as actually implemented. Yet the software architect has usually a good understanding of his/her architecture and can design the model he/she wants to reason about the software system.

To overcome this limitation, we propose to combine the different sources together. Intuitively, the variability and technical constraints of the plugin dependencies are *projected* onto the architectural model. As a result, we obtain an architectural model that is both representative of the software architecture and the variability actually supported by the system (by construction). The technical and formal details of the extraction process are described in Section 4.

The problem of integrating the software architect knowledge is addressed in Section 5. We developed techniques that allow the software architect to validate and edit (if needs be) the architectural FM extracted by the automated procedure. The key idea is to compute and present to the software architect the *differences* between two architectural FMs (e.g., the one designed by the software architect and the one extracted). The differencing techniques can also be used to address **Challenge 2** and the evolution of an architecture.

In Section 6, we describe the tooling support we developed for assisting the software architect. In Section 7, the proposed extraction and evolution techniques are applied and evaluated on different versions of FraSCAti.

4 Automatic Extraction of the Architectural Feature Model

This section addresses Challenge 1 (Extraction of Architectural FMs) discussed in Section 3. The general principle of the extraction is to combine two sources (an architectural model and a set of plugin dependencies) in order to synthesize a new integrated FM representing the features of the architecture as well as their variability and their technical constraints. Fig. 3 summarizes the steps needed to realize the extraction process.

As a first step, 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

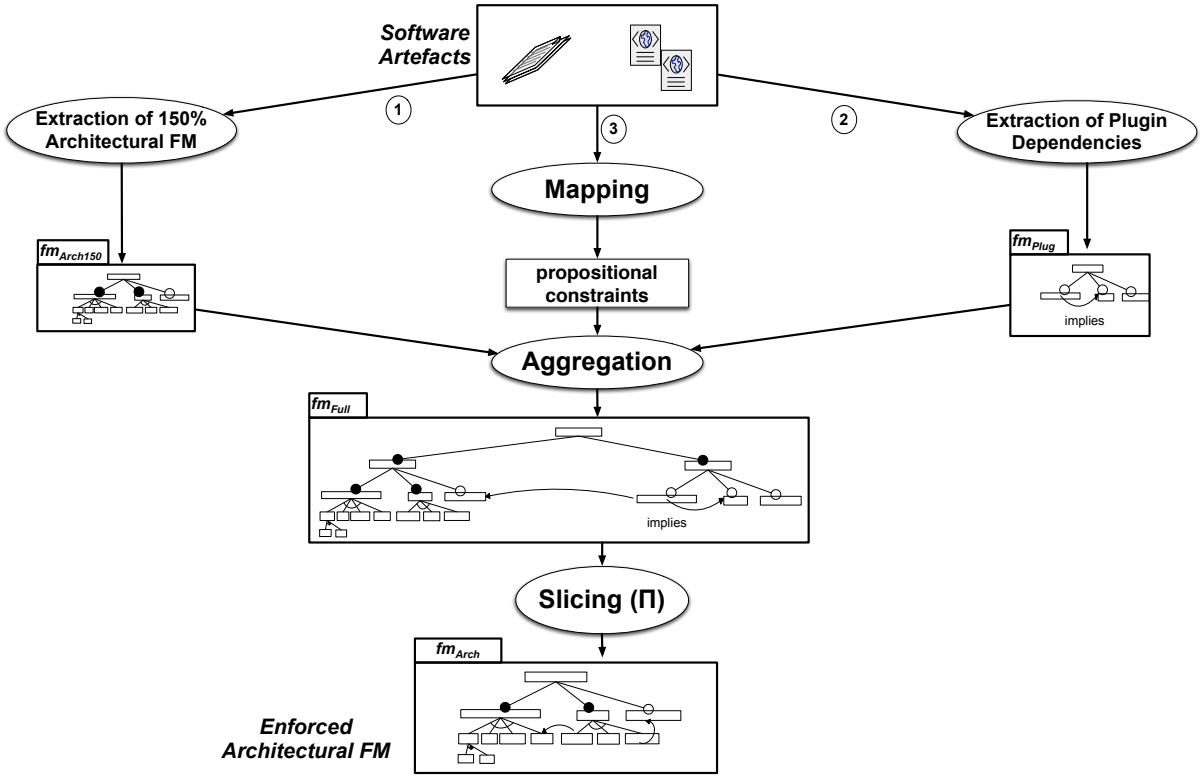


Fig. 3 Process for Extracting fm_{Arch}

plugins. We call it a 150% architecture because it is not likely that a FraSCAti configuration may contain them all. Consequently, $fm_{Arch_{150}}$ does include all the *features* provided by the FraSCAti SPL, but it still constitutes an over approximation of the set of *valid combinations* of features of the FraSCAti 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} more accurately represents the architectural variability provided by the system.

4.1 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. A reference having a 0..1 or 0..N multiplicity is *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 composition algorithm binds only one 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*, typically materialized by *optional* references (e.g., `metamodels` of composite `ScaParser`; `implementations`, `interfaces`, `bindings` and `property-types` of composite `AssemblyFactory`; `fractal-bootstrap-class-providers`, `delegate-membrane-generation`, `generators` and `compiler-provider` of composite

`ComponentFactory`). Algorithm 1 summarizes the behavior of the FM extractor.

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: $f_c \leftarrow CreateFeature(c)$
6: $fm_{Arch_{150}} \leftarrow AddMandatoryChildFeature(fm_{Arch_{150}}, f_{root}, f_c)$
7: $fm_{Arch_{150}} \leftarrow AddChildFeatures(fm_{Arch_{150}}, c, f_c, Arch_{150})$
8: **end for**

The root feature of the extracted FM (f_{root}) corresponds to the 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-level child features are produced by the $AddChildFeatures$ function (Algorithm 2).

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.
1: **for all** $r \in OptionalReferences(c)$ **do**
2: $MC \leftarrow FindMatchingComponents(Arch_{150}, r)$
3: **if** $MC \neq \emptyset$ **then**
4: $f_r \leftarrow CreateFeature(r)$
5: $FM \leftarrow AddOptionalChildFeature(FM, f_p, f_r)$
6: **if** $Multiplicy(r) = 0..1$ **then**
7: $g \leftarrow CreateXORGroup()$
8: **else if** $Multiplicy(r) = 0..N$ **then**
9: $g \leftarrow CreateORGroup()$
10: **end if**
11: $FM \leftarrow AddGroup(FM, f_r, g)$
12: **for all** $c_s \in MC$ **do**
13: $f_{c_s} \leftarrow CreateFeature(c_s)$
14: $FM \leftarrow AddChildFeatureOfGroup(FM, g, f_{c_s})$
15: $FM \leftarrow AddChildFeatures(FM, c_s, f_{c_s}, Arch_{150})$
16: **end for**
17: **end if**
18: **end for**

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 the set of all components c_s providing a service compatible with r.

Algorithm 3 specifies how to retrieve this set of *matching components* from the 150% architecture. The set of components matching a given $0..N$ reference r are obviously those providing a service bound to r via a wire. In the case of a 0..1 reference, in contrast, all compatible services are not necessarily bound to it. Thus, the matching components are all those that provide a service having an interface compatible with reference r.

Illustration Fig. 4 illustrates the extraction process when applied to FraSCAti. The left-hand side of the figure

Algorithm 3 $FindMatchingComponents(Arch_{150}, r)$

Require: A 150% architecture $Arch_{150}$, an optional reference r.
Ensure: The set MC of components defined in $Arch_{150}$ that provide a service compatible with r.
1: $MC \leftarrow \emptyset$
2: $t \leftarrow Target(r)$
3: **if** $Multiplicy(r) = 0..1$ **then**
4: $i \leftarrow Interface(r)$
5: $MC \leftarrow ComponentsWithCompatibleServiceInterface(Arch_{150}, i)$
6: **else if** $Multiplicy(r) = 0..N$ **then**
7: **for all** $w \in WiresHavingAsSource(Arch_{150}, r)$ **do**
8: $s \leftarrow TargetService(w)$
9: $c_s \leftarrow Component(s)$
10: $MC \leftarrow MC \cup \{c_s\}$
11: **end for**
12: **end if**

shows excerpts of the FraSCAti architecture expressed in SCA. The right-hand side of the figure depicts the incremental extraction of the architectural feature model $fm_{Arch_{150}}$. `FraSCAti`, as main composite of the system, becomes the root feature. The mandatory features at the first level of decomposition correspond to the first-level composites of FraSCAti, among which `sca-parser`[4]. The latter specifies a $0..N$ reference `metamodels`, which is translated into an optional child feature of `sca-parser`, that in turn serves as parent feature of an *OR* group. The child features of this group correspond to all the components that provide a service compatible with the reference `metamodels`. For instance, this is the case of the component sca-metamodel, providing a service `metamodel-provider` wired to the reference `metamodels` (then lines 7-12 of Algorithm 3 applied).

4.2 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}.

4.3 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 those of fm_{Plug} by means of *requires* constraints. This mapping allows us to determine (1)

[4] It should be noted that the extraction only focuses on the first-level composites that include some variability. It ignores all the composites that do not contain any optional references

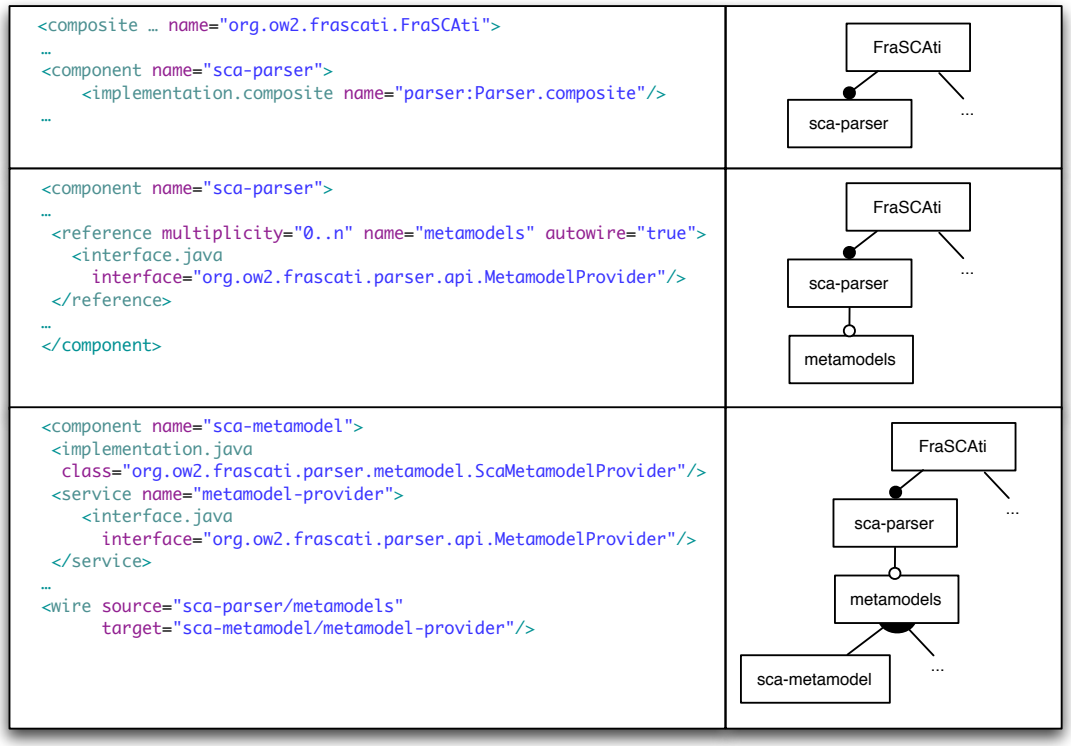


Fig. 4 Extraction of $fm_{Arch_{150}}$ applied to FraSCAti (excerpt).

which plugin provides a given architectural feature, and (2) which architectural features are provided by a given plugin.

4.4 Deriving fm_{Arch}

We now explain how we derive fm_{Arch} using $fm_{Arch_{150}}$, fm_{Plug}, the mapping between fm_{Plug} and $fm_{Arch_{150}}$, and an operation called *slicing*. We then illustrate the procedure using the example of Fig. 5. Intuitively, the variability and technical constraints induced by the plugin dependencies are *projected* onto the architectural model. In our case the use of plugin dependencies restricts the scope of the architectural FM by precluding some unauthorized configurations in $fm_{Arch_{150}}$.

4.4.1 Projecting Variability onto the Architectural Model

First the two FMs fm_{Plug} and $fm_{Arch_{150}}$ are *aggregated* under a synthetic root $FtAggregation$ so that the root features of the input FMs are mandatory child features of $FtAggregation$. The aggregation operation produces a new FM, called FM_{Full} (see Fig. 5). The propositional constraints relating features of fm_{Plug} to features of $fm_{Arch_{150}}$ are also added to FM_{Full}.

Second, we compute the projected set of configurations (see Definition 5) of FM_{Full} onto the set of features of $fm_{Arch_{150}}$ (i.e., $\mathcal{F}_{fm_{Arch_{150}}} = \{Arch, Ar1, \ldots, Ar6\}$).

To realize the projection, we use an operation called *slicing* (see Definition 4). Given a subset of features, the slicing operator produces a new FM characterizing the projected set of configurations (see Definition 5).

Definition 4 (Slicing) *We define slicing as an operation on FM, denoted $\Pi_{\mathcal{F}_{slice}}(fm) = fm_{slice}$ where $\mathcal{F}_{slice} = \{ft_1, ft_2, ..., ft_n\} \subseteq \mathcal{F}$ is a set of features (called the slicing criterion) and fm_{slice} is a new FM (called the slice).*

Definition 5 (Slice and projected set of configurations) *The result of the slicing operation is a new FM, fm_{slice}, such that: $[\![fm_{slice}]\!] = \{ x \cap \mathcal{F}_{slice} \mid x \in [\![fm]\!] \}$ (called the projected set of configurations).*

As several yet different FMs can represent a given set of configurations [66], we also take the feature hierarchy into account. In particular, we want to avoid slice FMs that are not readable and maintainable (e.g., for a software architect or for users configuring the architecture) due to an inappropriate hierarchy. Therefore we consider that the new FM produced by the slicing operation should have a hierarchy as close as possible to the hierarchy of the original FM (see Definition 6).

Definition 6 (Slice and feature hierarchy) *The feature hierarchy of the slice FM, denoted $G_{slice} = (\mathcal{F}_{FM_{slice}}, E_{slice} \subseteq E)$, is defined as follows:*

- *features include the slicing criterion except dead features (see Definition 7) of the original FM. Formally: $\mathcal{F}_{FM_{slice}} = \mathcal{F}_{slice} \setminus deads(FM)$,*

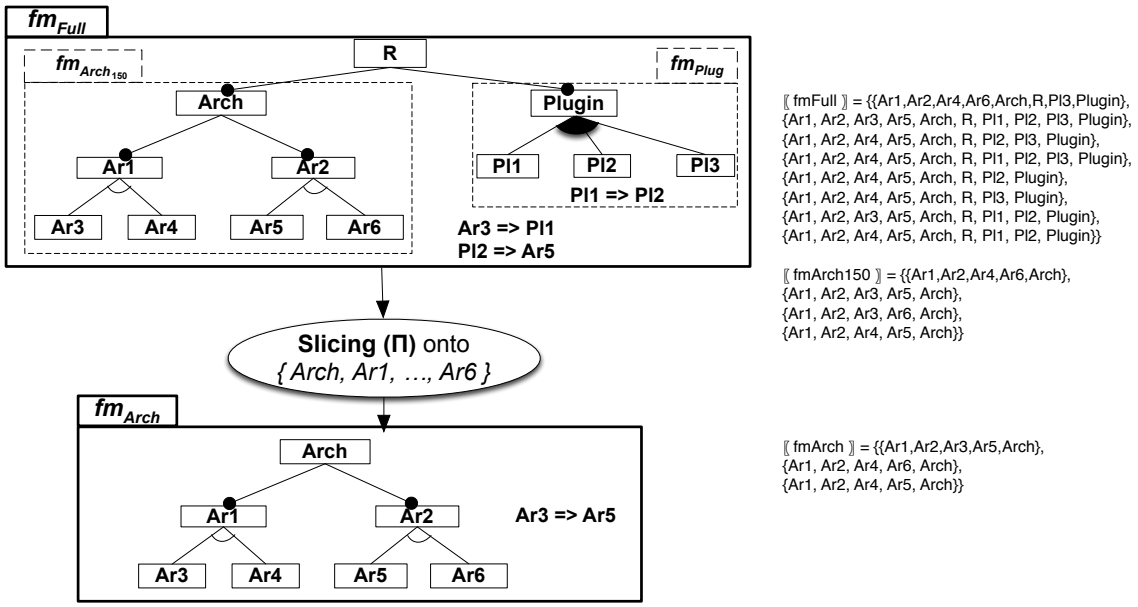


Fig. 5 Enforcing architectural FM using aggregation and slicing: an example

– *features are connected to their closest ancestor if their parent feature is not part of the slice FM. Formally:* $E_{slice} = \{e = (v, v') \mid e \in E' \wedge \nexists\, v'' \in \mathcal{F} : ((v, v'') \in E' \wedge (v'', v') \in E')\}$ *where* $G' = (\mathcal{F}', E')$ *is the transitive closure of the feature hierarchy* G *of the original FM.*

Definition 7 (Dead features) *A feature* f *of* FM *is dead if it cannot be part of any of the valid configurations of* FM*. The set of dead features of* FM *is noted* $deads(FM) = \{f \in \mathcal{F} \mid \forall c \in [\![FM]\!], f \notin c\}$.

4.4.2 Automation Our previous experience in the composition of FMs [4] has shown that *syntactical* strategies have severe limitations to accurately represent the set of configurations expected, especially in the presence of cross-tree constraints. The same observation applies for the slicing operation so that reasoning directly at the *semantic* level is required. The key ideas of our approach are to *i)* compute the propositional formula representing the projected set of configurations and then *ii)* reuse the reasoning techniques proposed in [21,11,8] to construct an FM from the propositional formula. We rely on the algorithm developed in [5] that combines this information with the known hierarchy of the slice (see Definition 6) in order to build a complete and valid FM.

4.4.3 Example In the example of Fig. 5, the resulting slice is called fm_{Arch}. As we want to focus on the variation points of the architecture, it only contains the features' name of $fm_{Arch_{150}}$. Formally:

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

We can verify that the relationship (see Definition 5) between the input FM, $[\![fm_{Full}]\!]$, and the slice FM, $[\![fm_{Arch}]\!]$, truly holds:

$$[\![fm_{Arch}]\!] = \{\, x \cap \{Ar1, Ar2, Ar3, Ar5, Arch\} \mid x \in [\![fm_{Full}]\!] \,\}$$

Importantly, we can notice that one configuration of the original $fm_{Arch_{150}}$ is no longer present in fm_{Arch}:

$$[\![fm_{Arch_{150}}]\!] \setminus [\![fm_{Arch}]\!] = \{Ar1, Ar2, Ar3, Ar6, Arch\}$$

Indeed the slice FM fm_{Arch} contains an additional constraint $Ar3 \Rightarrow Ar5$, that was not originally restituted as such in $fm_{Arch_{150}}$[5]. It should also be noted that the hierarchy of the slice correctly restitutes the hierarchical decomposition of the architecture.

This very simple example already shows two key benefits of combining different variability sources and using the slicing operator. First, constraints, not originally present in the 150% architectural FM, are automatically restitued in a new architectural variability model and can be reported back to the software architect. Second, restrictions are applied on the over approximated configurations set characterized by the 150% architectural FM. Therefore some configurations, actually not supported by the architecture, are now precluded.

5 Support for the Evolution of Architectural Feature Models

This section addresses Challenge 2 (Evolution of Architectural FMs) discussed in Section 3.

[5] Similarly, the constraint $Ar4 \Rightarrow Ar6$ could be restituted in the model (using the information of the implication graph, see above). The slicing operator does not add this constraint because of the redundancy with $Ar3 \Rightarrow Ar5$.

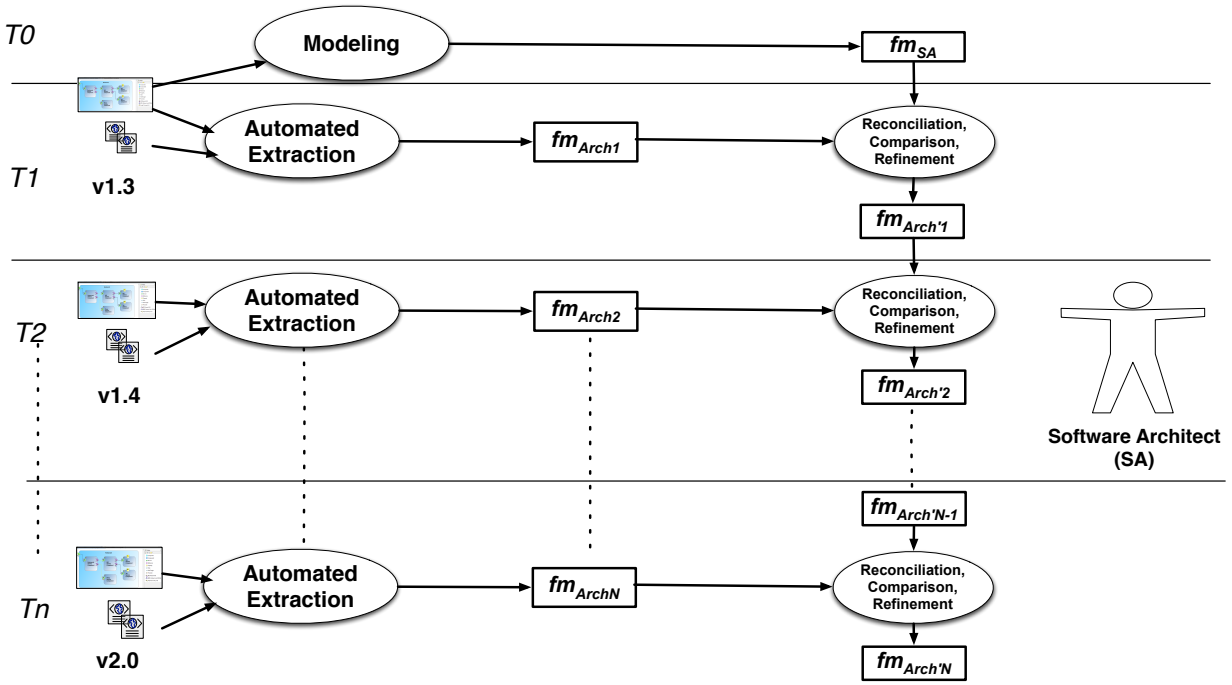


Fig. 6 Extraction process and evolution of architectural FMs

For each version of a plugin-based system like FraSCAti, the architectural FM synthesized by the extraction procedure should be validated by the *software architect* (SA). In particular, the SA should control that the variability information and the characterized set of configurations do not contradict his/her intention and knowledge of the architecture. For example, the SA may consider that the mandatory status of some features in the extracted FM is not appropriate.

The idea we defend in this article is that, for assisting the SA, the extracted FM can be compared with his/her mental representation and with older versions of architectural FMs. As a result, an appropriate support for comparing two FMs and reasoning about an *evolution* of an FM is highly needed.

First evolution. At the starting point of the re-engineering of FraSCAti as an SPL, an intentional model of the variability was elaborated by the SA. The resulting FM, denoted fm_{SA}, was the first available representation of the FraSCAti architecture (version 1.3, see Fig. 6). The extraction process previously described was then applied to produce another representation (fm_{Arch_1}) for the same version of the architecture. Therefore, fm_{Arch_1} can be seen as an *evolution* of fm_{SA} given that the FM originally elaborated by the SA has now evolved to an FM automatically extracted.

The absence of a *ground truth* FM – an FM for which we are certain that each combination of features is supported by the SPL architecture – makes uncertain the accuracy of the variability specification expressed in fm_{Arch_1} as well as in fm_{SA}. As both the software architect FM and fm_{Arch_1} may represent differently the variability of the architecture, there is need to *reconcile* and *refine* the two FMs. The result of this process is a new FM, $fm_{Arch'_1}$, that integrates the intentional variability and the SA knowledge of fm_{SA} and the explicit variability expressed by fm_{Arch_1}.

Versions and evolutions. As any software project, the FraSCAti architecture evolves. Many features and dependencies are added and removed. Naturally, the extraction procedure is reiterated on different versions (e.g., version 1.4) of a FraSCAti architecture, producing as different FMs. Nevertheless the confidence of the resulting FMs remains unclear:

- the extraction procedure may be faulty (e.g., inadequate for a specific version of FraSCAti);
- the variability and the constraints may not be correctly documented in the architecture artefacts;
- the SA knowledge may not be taken into account.

Managing the evolutions. For controlling and hopefully validating the evolution of an FM, the SA should be able to understand and exploit the *differences* between two FMs. A possible solution is to elaborate, for each version of a FraSCAti architecture, a new FM representing the current variability and then compare it with the extracted FM. Nevertheless, the elaboration from scratch of a new FM (like the SA did for version 1.3) is time-consuming and error-prone. There is an opportunity to *reuse* FMs resulting from a refinement. Then, similarly to what has been done when reasoning about fm_{SA} and fm_{Arch}, reconciliation and comparison techniques are applied. For example, as shown in Fig. 6, $fm_{Arch'_1}$ (resulting refined FM for version 1.3 of FraSCAti) can be

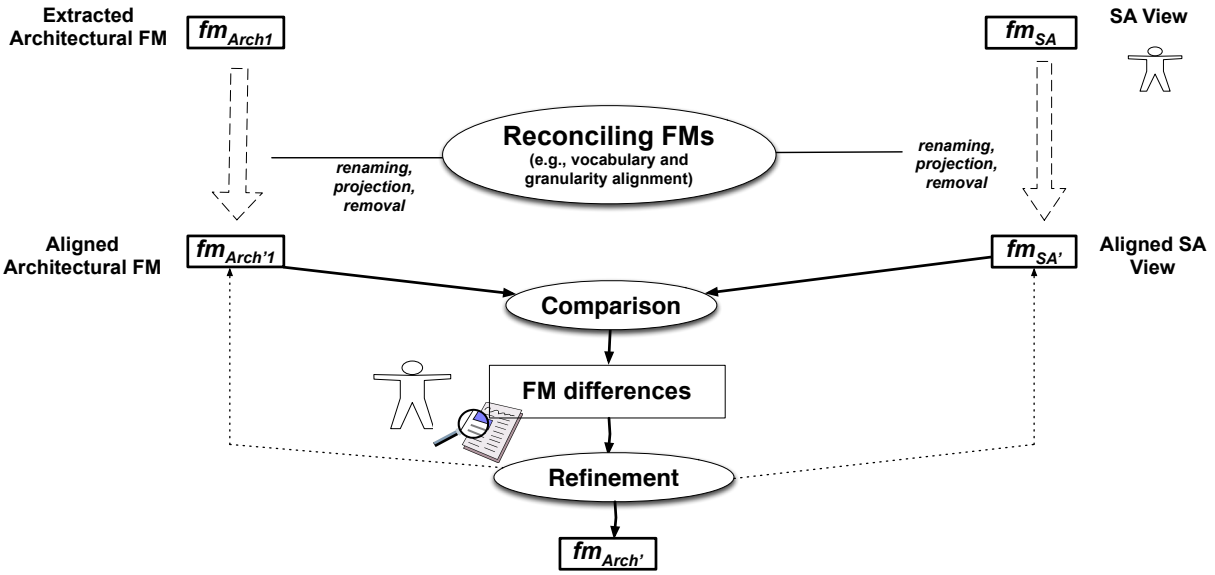


Fig. 7 Process for integrating the SA knowledge: reconciliation, comparison and refinement

compared with $fm_{Arch'_2}$ (extracted FM for version 1.4 of FraSCAti).

Support for managing evolutions. Fig. 7 presents the overall process for comparing two FMs (e.g., for comparing an extracted FM with an FM designed by the SA). In the following, we describe dedicated techniques related to the evolution of FMs for supporting the SA activities, namely reconciliation (see Section 5.1), comparison and refinement (see Section 5.2).

5.1 Reconciliation of Feature Models

Let us consider fm_{SA} and fm_{Arch} of Fig. 7. The SA should be able to determine if the variability choices in fm_{SA} comply with what is expected by himself (i.e., as specified in fm_{Arch}), and vice-versa. In case variability choices are conflicting, the SA can refine the architectural FM. Similar observations can be made when reasoning about two different versions of a FraSCAti architecture.

A first obstacle concerns the need to *reconcile* the two FMs (e.g., fm_{Arch} and fm_{SA}). Both FMs come from difference sources or versions. A preprocessing step is needed before reasoning about their relationship. Firstly, the *vocabulary* (i.e., names of features) used in both FMs may differ from each other, and should be aligned consequently. Many operations (see below) indeed assume[6] that features are identified by an unique label (i.e., name) in an FM and that two features of two FMs match if and only if they have the same name.

To avoid unexploitable differencing results, some predirectives are needed and consist in renaming features. We rely on string matching techniques (e.g., Levenshtein distance) to automatically identify corresponding features. More sophisticated matching techniques already integrated in model-based tools (e.g., see [25,37]) can also be considered but have not been used in the context of FraSCAti (see next section).

Secondly, *granularity* details differ. For example some features in one FM are not present in the other FM. The removal of features is thus needed. This questions the semantics of the removal operation. What about cross-tree constraints involving a feature that has been removed? What about variability information of the parent and children features when a feature is removed in the middle of a hierarchy?

We consider that, when a feature is removed, two values (true or false) can be assigned and should be considered accordingly. This operation cannot be done syntactically in the general case. We rely on the slicing operation previously defined that can removed a set of features while guaranteeing configuration semantics properties. For example, the removal of two features $Felix$ and $Equinox$ of fm_{SA}, leading to a new FM fm'_{SA}, corresponds to the following slicing operation:
$fm'_{SA} = \Pi_{\mathcal{F}_{fm_{SA}} \setminus \{Felix,Equinox\}} (fm_{SA})$

5.2 Comparison and Refinement of FMs: A Toolbox

At this step, we can compare the two FMs (e.g., reason about the relationship between fm_{Arch} and fm_{SA}). It means we need to compute and present *differences* of the two FMs in a comprehensible manner to the SA. The problem of FM differences is a general problem that may occur in other contexts (e.g., management of a product line offering) [7]. We present here only the techniques relevant to our specific context.

[6] This assumption is also shared by [70,63,26].

5.2.1 Principles of FM Differences Several techniques for the differencing (for short, *diff*) of FMs can be considered. Let fm_1 and fm_2 be two FMs. Roughly, the diff between fm_1 and fm_2 is the set of elements in fm_1 but not in fm_2. From a syntactical perspective, the elements to be considered in the diff may be features, feature hierarchies, feature groups or implies / excludes. We present *syntactic differencing* techniques in Section 5.2.2. Though the syntactic diff might be useful, we believe that a semantic diff for FMs should also be developed and possibly be combined with syntactic differencing. We present *semantic differencing* techniques in Section 5.2.3.

5.2.2 Syntactic Diff A general approach to model differencing is to concentrate on matching between model elements using different heuristics related to their names and structure and on finding and presenting differences at a concrete or abstract syntactic level. As previously stated, we assume that two features of two FMs match if and only if they have the same name.

In terms of feature modeling, elements of interest are features, variability information (mandatory features, feature groups, and propositional constraints) and feature hierarchy (see Definition 1 and 2). We thus consider the diff of these model elements:

- **Diff of features:** $\mathcal{F}_{diff}$ is the set of features that are in fm_1 but not in fm_2, i.e., $\mathcal{F}_{diff} = \mathcal{F}_1 \setminus \mathcal{F}_2$.
- **Diff of feature hierarchies:** several techniques can be considered (e.g., tree edit distance [15]), including the computation of E_{diff} the set of edges modeling parent-child relationships in fm_1 but not in fm_2. Formally: $E_{diff} = E_1 \setminus E_2$.
- **Diff of mandatory features:** a syntactic diff of mandatory features produces $E_{MAND_{diff}} = E_{MAND_1} \setminus E_{MAND_2}$.
- **Diff of feature groups:** It is useful to determine feature groups (Xor and Or) that are in fm_1 but not in fm_2, including $G_{XOR_{diff}} = G_{XOR_1} \setminus G_{XOR_2}$ and $G_{OR_{diff}} = G_{OR_1} \setminus G_{OR_2}$. We consider that two feature groups are equal if and only if their parent features match and their child features match.

5.2.3 Semantic Diff A practitioner rather wants to understand the difference between the two FMs in terms of *configuration semantics* (i.e., in terms of sets of configurations). We now address semantically the list of differences. We translate fm_1 and fm_2 into two formula ϕ_1 and ϕ_2. Performing at the level of abstraction for Boolean variables may produce unexploitable results for a practitioner. Stated differently, a practitioner wants to understand differences in terms of feature modeling concepts rather than in terms of a propositional formula. We thus take care of producing meaningful information based on the analysis of the two formula.

Diff of information extracted from the two formula.

A first general strategy consists in analyzing separately each formula and then performs the differences of the information produced.

- **Diff of binary implication graphs:** We consider a binary implication graph of an FM and its propositional formula ϕ as a directed graph $BIG = (V_{imp}, E_{imp})$ formally defined as follows:

$$V_{imp} = \mathcal{F} \qquad E_{imp} = \{(f_i, f_j) \mid \phi \wedge f_i \Rightarrow f_j\} \tag{1}$$

  Each binary, directed edge from feature f_i to feature f_j represents a binary implication. Based on the analysis of ϕ_1 and ϕ_2, we can produce BIG_1 and BIG_2 and then compute $BIG_{diff} = BIG_1 \setminus BIG_2$. It is then straightforward to compute the set of binary implications expressed in fm_1 but not in fm_2. As we support arbitrary propositional constraints in an FM, it should be noted that BIG_{diff} cannot be produced syntactically in the general case. Furthermore, the binary implication graph structure, reified from the propositional formula, has the advantage of exposing an information than can be directly translated in terms of feature modeling (i.e., either as a binary implication between a child feature and a parent feature or simply as a cross-tree constraint).
- **Diff of cliques in implication graphs** We extend the previous technique to n-ary biimplications. A n-ary biimplication involves n features such that $f_i \Rightarrow f_j$ for any $i, j = 1 \ldots n$. It can be obtained by computing cliques in BIG. A clique in the implication graph is a subgraph in which any two vertices are connected by an edge. A clique in the exclusion graph requires each member to have an exclusion to every other member. For the purpose of conciseness (no set of features is subsumed by other), we compute *maximal* cliques in BIG (corresponding to features that always appear together in an FM).

Reasoning about the two formula.

A second general strategy consists in producing relevant information based on the logical combinations of the two formula. We briefly present here two existing techniques [70,26] and another one we developed in previous work [7]. All these techniques are candidates for managing the evolution of FMs and have been applied on FraSCAti (see next section). Furthermore a comprehensive comparison between differencing techniques proposed in this article and in the literature is performed in Section 8.1.

Relationship between two FMs Thüm *et al.* [70] reason on the nature of FM edits, for example, when fm_1 is edited (e.g., some features are moved, added, or removed), giving fm_2. They provide a classification (see Definition 8).

Definition 8 (Kind of edits) *fm_1 is a specialization of fm_2 if $[\![fm_1]\!] \subset [\![fm_2]\!]$; fm_1 is a generalization of fm_2 if $[\![fm_1]\!] \subset [\![fm_2]\!]$; fm_1 is a refactoring of fm_2*

if $[\![fm_1]\!] = [\![fm_2]\!]$*;* fm_1 *is an arbitrary edit of* fm_2 *in other cases.*

Quotient In [26], an algorithm is presented that takes as input two formula ϕ_1 and ϕ_2 in *conjunctive normal form* (CNF) – FMs are easily converted to CNF. The algorithm finds for the quotient (i.e., difference) all clauses in ϕ_1 which are not entailed by ϕ_2 through the satisfiability checks of $\phi_2 \wedge \neg c$ (c being a clause of ϕ_1).

Diff of Formula The two previous techniques fail to comprehensively represent the difference of the two configuration sets. To raise the limitations, we develop a diff operator, noted $\oplus_\setminus$, that takes as input two FMs and produces a diff FM (denoted $fm_{diff} = fm_1 \oplus_\setminus fm_2$). The following defines the semantics of this operator:

$$[\![fm_1]\!] \setminus [\![fm_2]\!] = \{x \in [\![fm_1]\!] \,|\, x \notin [\![fm_2]\!]\} = [\![fm_{diff}]\!] \quad (M_1)$$

The computation of the diff formula, that encodes the diff set of configurations and is used for the automated synthesis of fm_{diff}, is described in [7].

5.2.4 Step-wise Refinement Once differences have been identified and understood, the SA can *edit* the two FMs:

- change the variability associated to features (e.g., set optional a mandatory feature);
- add and remove some constraints (e.g., implies constraints);
- modify the feature hierarchy.

The edits to an FM (e.g., fm_{Arch}) change its syntactic and semantic properties. Once edits are applied, the differences with another FM (e.g., fm_{SA}) should be re-computed. Therefore managing differences is a multi-step, incremental process. Edits are incrementally applied on the two FMs until obtaining a satisfying relationship between the two FMs.

6 Tool Support

We need a practical support for using the techniques previously described:

- *extraction* support: the procedure aiming to extract the variability model of the plugin-based architecture at a certain time (fm_{Arch}).
- *evolution* support: the set of FM operations designed to assist the architect in monitoring the evolution of the plugin-based architecture.

In the context of both tasks, automation and reproducibility of the operations are crucial success factors. To this end, we rely on FAMILIAR (for *FeAture Model scrIpt Language for manIpulation and Automatic Reasoning*) a domain-specific language for managing FMs [6]. The language includes facilities for aggregating and slicing FMs, editing FMs (e.g., renaming and removal of features), reasoning about FMs (e.g., validity, comparison of FMs) and their configurations (e.g., counting or enumerating the configurations in an FM). The language also integrates the differencing techniques through the form of operations over FMs (computation of candidate feature groups and implication / exclusion graphs, etc.).

FAMILIAR is an executable, textual language and comes with an Eclipse-based environment that is composed of textual editors, an interpreter that executes FAMILIAR scripts, and an interactive toplevel, connected with graphical editors (see Fig. 8). Two reasoning back-ends (SAT solvers using SAT4J and BDDs using JavaBDD) are internally used and perform over propositional formula to implement the operators. Operations can be sequentially executed while properties of the variables can be observed.

It is particularly important in our context since the process for managing differences of two FMs is incremental and interactive. Hence, complex management scenarios can be applied using FAMILIAR environment. For example, a software architect can decompose two FMs, then apply some techniques to understand local differences, edit the FMs, and reiterate the process. We will see in the next section that this kind of FM management scenario is likely to occur when managing the evolution of architectural FMs.

In summary, FAMILIAR is used for two purposes:

- the *extraction* procedure generates FAMILIAR code that is executed by the FAMILIAR interpreter to obtain fm_{Arch} for each version of FraSCAti. As a result, the procedure described in Section 4 can be realized and works as follows:
  - the extraction of $fm_{Arch_{150}}$ is supported by a dedicated Java program that makes use of the FraSCAti's SCA parser for building the 150% architecture of the plugin-based system of interest ($Arch_{150}$);
  - the plugin FM fm_{Plug} is automatically extracted from the build files. In the particular case of FraSCAti, the extractor analyzes the Maven files (i.e., `pom.xml`) associated to each system plugin, that specify inter-plugin dependencies.
  - the two FMs as well as the mapping are translated in the FAMILIAR language so that aggregate and slicing operators can be executed to compute and serialize fm_{Arch};
- FAMILIAR provides the SA with a dedicated approach for manipulating and reasoning about FMs when managing the *evolution* of FMs.

Our toolkit also includes a converter that provides bidirectional translation between FAMILIAR and different formats (SPLOT [45], FeatureIDE [36], S2T2 [60, 57], TVL [16], etc.). This allows the architectural FMs derived by our approach to be immediately visualized

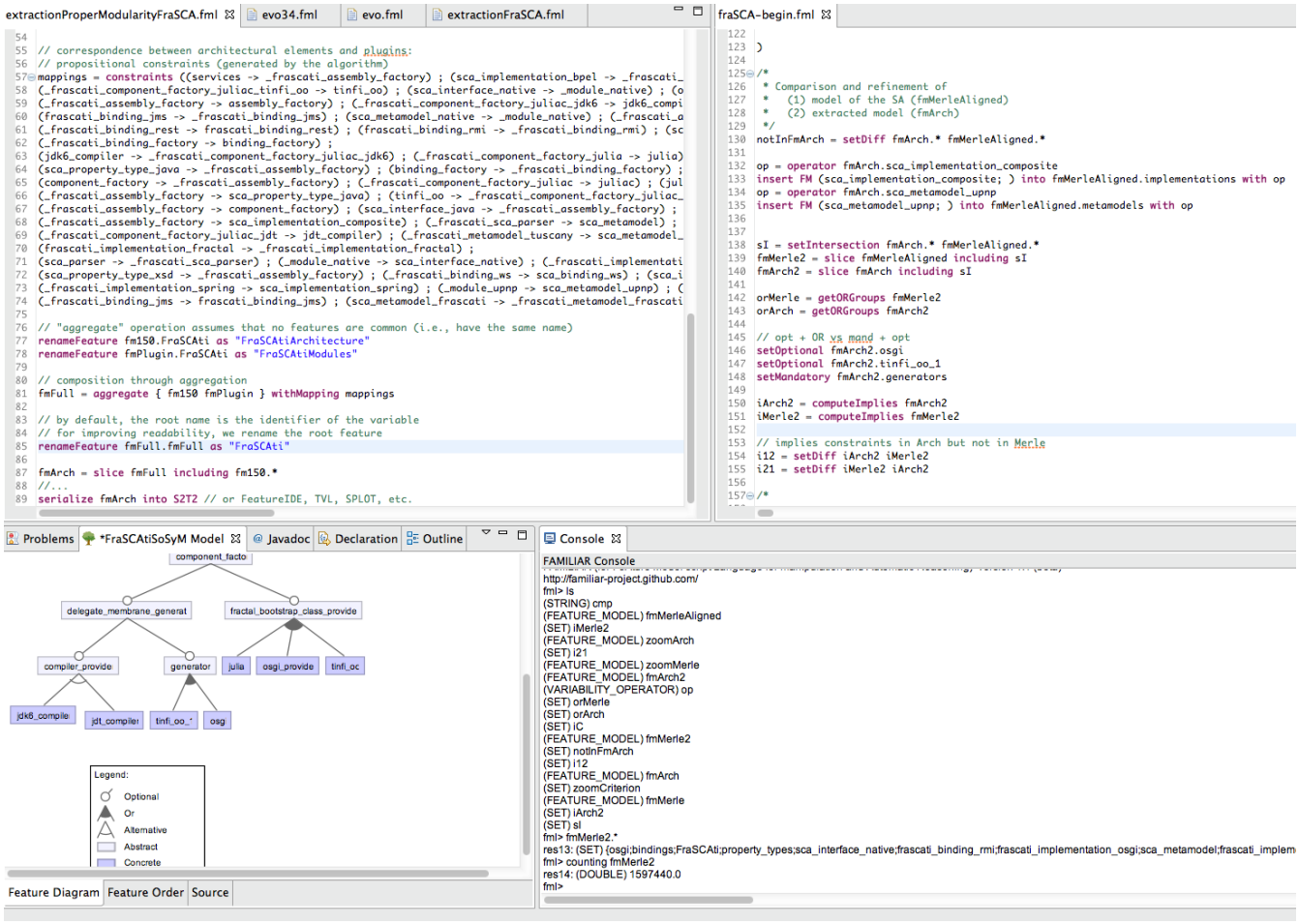


Fig. 8 FAMILIAR environment: scripts, interactive session and use of FeatureIDE editors

and used as input of subsequent software configuration or generation tasks.

7 Evaluation

7.1 Performance evaluation

The aggregation operator is purely syntactical while the extraction algorithm presented in Section 4 is a breadth-first search algorithm. Slicing and computing differences are the most costly operations in the extraction and evolution management process. Below, we analyze the complexity of these operations.

Slicing. The slicing algorithm proceeds in three steps. First, the feature hierarchy of the slice is determined by connecting features to the closest parent feature present in the slicing criterion. The computation of the feature hierarchy is immediate and basically consists in removing edges in a tree. Second, the propositional formula representing the projected set of configurations is computed by existential quantification. Third, satisfiability techniques are applied to construct a complete FM (including variability information and cross-tree constraints) based on the formula and the computed feature hierarchy. Satisfiability techniques can be realized using either BDDs or SAT solvers [21,66].

As shown in [21], the cost of FM construction is polynomial regarding the size of the BDD. We reuse the heuristics developed in [46] to reduce the size of the BDD. Our experiments with BDDs show that, in practice, the primary limit of the BDD-based implementation lies in the difficulties to construct BDD from the original FM (i.e., the original FM should not be more than 2000 features).

SAT solvers can scale for FMs with more than 2000 features. As SAT solvers require the formula to be in conjunctive normal form (CNF), the slice formula should also be in CNF. To avoid the exponential explosion of clauses, we developed specific techniques and some heuristics to determine the order in which existential quantification should be applied [5]. Using SAT, we can scale up to FMs with 10000 features in certain conditions.

FM Differences. The computation of *BIG* heavily depends on *satisfiability* checks of implications. In practice, the computation of *BIG* scales for thousands of features and can be realized using SAT solvers or BDDs [21,66].

Due to transitivity of implication, maximal cliques are actually strongly connected components in BIG, which can be found efficiently by graph traversal. We use the Bron-Kerbosch algorithm for finding all maximal cliques. In practice, the computation of cliques scales for thousands of features [66].

The synthesis of a diff FM (see Definition M_1) performs over the formula representing the diff set of configurations, denoted ϕ_{diff} [7]. As argued in [70], ϕ_{diff} is not in CNF and an exponential explosion of clauses occurs when translating to CNF, even for a small number of features. Therefore SAT solvers cannot be used directly since most of them require a CNF formula as input. Our current solution is to rely on BDDs for computing and reasoning about ϕ_{diff}, since computing the disjunction, conjunction and negation of BDDs can be performed in at most polynomial time with respect to the size of the BDD involved, even for non CNF formula.

In Practice. The order of complexity of FMs encountered in FraSCAti is manageable. FMs exhibit lots of constraints but at worst only 123 features (see Table 1, page 20) when combining $fm_{Arch_{150}}$ and fm_{Plug} for the version 1.5. At this scale, we observed no difficulty. The operations on FMs can be efficiently executed in a few seconds using our implementation of the slicing operation and differencing techniques.

7.2 Practical Evaluation

We applied the tool-supported techniques previously described on different versions of FraSCAti[7]. P. Merle, principal FraSCAti developer for six years now, plays the role of the SA in this study. Specifically, we aim at assessing them regarding the two main challenges identified in Section 3:

- **(RQ1) Extraction of variability:** Is the extraction procedure accurate or faulty? Are the properties of the produced FMs coherent with what is expected by the SA? To what extent is the SA knowledge needed for recovering the architectural variability? For this purpose, we determine the variability information inferred by the extraction procedure and analyze the differences between fm_{Arch} and fm_{SA}. We also report qualitative insights gained when the SA validates the extracted FM.
- **(RQ2) Evolution of variability:** Are the differencing techniques exploitable for the SA? Can an evolution be controlled and validated by the SA? We apply previous techniques and report similar quantitative and qualitative observations for two other versions of FraSCAti.

[7] Further details and material (including FMs and FAMILIAR scripts) about the experiment are available in [1].

The remainder of this section is organized as follows. In Section 7.2.1 we report on our results when extracting the version 1.3 of FraSCAti (the starting point of our work). In Section 7.2.2 we describe how we integrate and exploit the SA knowledge. In Section 7.2.3 we report on our results on other versions of FraSCAti. In Section 7.2.4 we answer the two research questions **(RQ1)** and **(RQ2)**.

7.2.1 Automatic Extraction (version 1.3) We applied the extraction procedure for the version 1.3 of FraSCAti.

Properties of the input FMs.

The extraction procedure produces three kinds of information:

- the FM $fm_{Arch_{150}}$ contains 50 features;
- the FM fm_{Plug} contains 41 features and 81 constraints;
- the bidirectional mapping between features of $fm_{Arch_{150}}$ and fm_{Plug} consisting in 78 propositional constraints (i.e., implies constraints).

As a result, the FM fm_{Full} resulting from the aggregation of $fm_{Arch_{150}}$, fm_{Plug} and the bidirectional mapping contains 159 cross-tree constraints and 92 features.

Comparison of the extracted FM and $fm_{Arch_{150}}$

The slicing technique of fm_{Full} onto $\mathcal{F}_{fm_{Arch_{150}}}$ produced fm_{Arch}. We observed that fm_{Arch} is a specialization of $fm_{Arch_{150}}$. More precisely, $fm_{Arch_{150}}$ admits 13 958 643 712 possible architecture configurations ($\approx 10^{11}$), while fm_{Arch} represents 936 576 distinct products ($\approx 10^6$).

A first observation is that the slicing technique significantly reduced the over approximation of $fm_{Arch_{150}}$.

To improve further our understanding and identify possible benefits of our technique, we computed the differences between fm_{Arch} and $fm_{Arch_{150}}$. We observed that:

- 12 *core*[8] features have been deduced. Those features were initially defined as optional in $fm_{Arch_{150}}$;
- some features that were initially defined as part of an Or-group in $fm_{Arch_{150}}$ are now all declared optional. More precisely, 5 Or-groups are no longer present in fm_{Arch}, while 2 Or-groups are commonly shared by fm_{Arch} and $fm_{Arch_{150}}$;
- 9 implies constraints and 5 bi-implies constraints have been deduced.

Thereby, a second observation is that a considerable amount of variability information has been inferred thanks to the extraction procedure.

[8] A feature f of FM is a core feature if it is part of all valid configurations of FM.

Comparing fm_{Arch} and $fm_{Arch_{150}}$.

The two previous observations let suggest that our extraction procedure improves the quality of the architectural FM. Yet the possible improvements have to be validated by the SA.

A first validation was done by validating the differences between $fm_{Arch_{150}}$ and fm_{Arch}. The SA notably explains why, in fm_{Arch}, many core features have been deduced and why Or-groups are no longer present.

He first observed that core features deduced by the extraction were originally part of an Or-group in $fm_{Arch_{150}}$. He then explains that the presence of core features provides a *default* (i.e., mandatory) solution. Typically, core features are related to Java (SCA implementation, SCA property type, interface), the default solution provided by FraSCAti. As a result, all other sibling optional features can be deactivated (since at least one has been selected by default), thus justifying why the features are no longer forming an Or-group of $fm_{Arch_{150}}$.

7.2.2 Integration of the SA knowledge (version 1.3) A second validation of the extraction procedure was done by validating the differences between fm_{SA} and fm_{Arch}. At the starting point of our reverse engineering effort, the SA designed an architectural FM, denoted fm_{SA} hereafter, for the same version (1.3) of FraSCAti. fm_{SA}, contains 39 features and 7 constraints.

Reconciling FMs.

A preliminary step is to reconcile fm_{Arch} and fm_{SA}, that is, dealing with possible vocabulary and granularity mismatches. We now report the problems encountered and the use of advanced techniques we present in Section 5 to assist the SA.

Vocabulary. Using Levenshtein distance, we automatically detect 32 corresponding features. As an example, MMFraSCAti of fm_{SA} has been identified to correspond to sca_metamodel_frascati of fm_{Arch}. 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).

Granularity. fm_{SA} only contains 39 features whereas fm_{Arch} contains 50 features.

First, two exclusive features *Felix* and *Equinox* are present in fm_{SA} but not in fm_{Arch}. A discussion with the SA reveals that these two plugins do not explicitly define architecture fragments in SCA. We indeed observed that the two features are present in fm_{Plug} but not in $fm_{Arch_{150}}$ (and hence cannot be present in fm_{Arch} by construction). As a consequence, the explanations of the SA validate the fact that the variation point cannot be identified by the automatic extraction procedure.

Secondly, some features are present in fm_{Arch} but not in fm_{SA}. This time, 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. Several reasons were given by the SA:

- **accidental complexity:** the SA recognizes that some features were missing in his FM. Given the complexity of the FraSCAti project, this is not surprising that the SA forgets some features. Some oversights are related to "helper" features of FraSCAti (such as the features features binding factory or juliac) that are generally not used by developers, while other oversights were qualified as more relevant from a configuration perspective (additional metamodels and binding types).
- **modeling intention:** the SA reveals that he *intentionally* ignored some features in fm_{SA}. He argued that there are mandatory features (e.g., every FraSCAti configuration has a Java interface) and that his focus was on variability rather than commonality. We indeed verify the mandatory nature of the features (e.g. sca_interface_java) in fm_{Arch} (see above).
  Another example related to the way features are modeled concerns a feature of fm_{Arch}, juliac, not modeled in fm_{SA}. By simplification, features juliac and delegate-membrane-generation have been merged by the SA into an unique feature MembraneGeneration.
- **obsolete features:** for the feature services, the SA explains that this architectural element is an empty composite that "could have been used but have not yet an interest".

Editing FMs for reconciling them. Based on these observations, the SA decided to edit FMs as follows:

- renaming directives, automatically generated or specified by the SA, were applied on fm_{SA} so that the vocabulary conforms to the one of fm_{Arch}, at least for the corresponding features;
- features forgotten by the SA were added to fm_{SA} with the same variability status as in fm_{Arch};
- the slicing operation has been applied two times for removing other features present in fm_{Arch} (resp. fm_{SA}) but not in fm_{SA} (resp. fm_{Arch}).

Managing FM differences.

Once the two FMs are aligned, we can reason about differences between fm_{Arch} and fm_{SA}. We now report *what* differencing techniques have been used and *how* they helped to manage differences between the FMs.

A first comparison is to determine the kind of relationship between fm_{Arch} and fm_{SA} (see Definition 8). We obtain an arbitrary edit, that is, some configurations of fm_{Arch} are not valid in fm_{SA} (and vice-versa). To

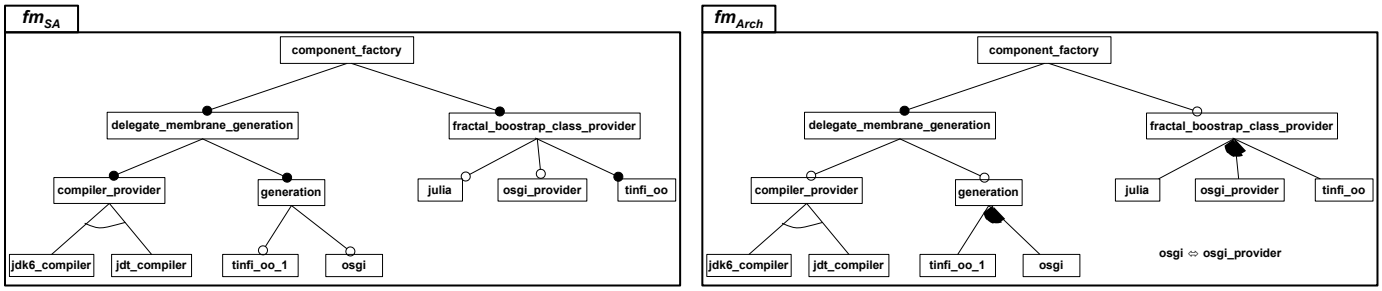


Fig. 9 Variability mismatch between fm_{SA} and fm_{Arch} (version 1.3)

go further, we use the diff operator. We enumerate and count the unique configurations of fm_{Arch} and fm_{SA}. Nevertheless, the techniques appear to be though useful not sufficient to really understand the differences between the two FMs.

Intuitively, we needed to identify more *local* differences. We used syntactic technique to compare the variability associated to features of fm_{Arch} and fm_{SA} that have the same name. We detected that:

- four features are optional in fm_{Arch} but defined as mandatory in fm_{SA};
- two sets of features belong to Or-groups in fm_{Arch} whereas in fm_{SA}, the features are all optional (see features tinfi_oo_1, osgi and julia, tinfi_oo, osgi_provider).

We observed that the variability mismatch concerns a subset of features being part of the same sub-FM of fm_{Arch} and fm_{SA}. Therefore we used the slice operator to reason on this specific part. Fig. 9 depicts the two resulting FMs.

Or-groups vs optionals. Three subtle situations of variability mismatch have been encountered and are interesting to explain:

- feature generators is optional and its children tinfi_oo_1, osgi are forming an Or-group in fm_{Arch} whereas feature generators is mandatory and its children tinfi_oo_1, osgi are all optional in fm_{SA}. At first glance, the difference seems important but the intention of the SA is actually similar to the variability expressed in fm_{Arch}. In terms of sets of configurations, fm_{SA} authorizes four combination of features {generators, tinfi_oo_1, osgi}, {generators, osgi}, {generators, tinfi_oo_1}, and {generators}. fm_{Arch} authorizes exactly the same set, except {generators}. It means that in both cases a configuration of a FraSCAti architecture may have zero or some *concrete* generators (i.e., {tinfi_oo_1, osgi}). The feature {generators} can be seen as an *abstract*[9] feature.

  As a result, the two FMs, though modeling differently the variability, have the same intention. It has been decided by the SA to keep the solution of the extraction procedure.
- feature fractal_bootstrap_class_provider is mandatory in fm_{SA} and one of its child feature tinfi_oo is mandatory. On the contrary, fractal_bootstrap_class_provider is optional in fm_{Arch}, and its children form an Or-group. The discussions with the SA reveal that, indeed, the architecture of FraSCAti authorizes a configuration *without* fractal_bootstrap_class_provider. The initial intent of the SCA was to state, that this feature is *often*[10] necessary. He explained the mandatory status of the feature tinfi_oo as a *default* implementation.
  Nevertheless, the SA recognized that fm_{Arch} accurately restitutes the flexibility of the architecture.
- the feature compiler_provider is optional in fm_{Arch} but mandatory in fm_{SA}. The SA confirms that a FraSCAti architecture has not necessarily to embed a complete Java compiler – minimal ($\leq 4Mo$) FraSCAti architecture for embedded systems can thus be derived and deployed. Therefore fm_{Arch} accurately models the variability of the feature compiler_provider.

Implications. We then used semantic techniques to compare the two FMs. We observed that the FMs involved have a large number of cross-tree constraints (i.e., binary implications between features). The so-called implies constraints are very important in the FraSCAti case study. First, the software architect specified many binary implications when elaborating the FM. Second, the extraction procedure combines different sources of information, including plugin dependencies. These dependencies are essentially expressed through *implies* constraints. Therefore we made an extensive use of the diff between binary implication graphs.

The diff between binary implication graphs has the merit of reifying the differences of the two FMs in terms of *implies* constraints. It is then easier for a software

[9] In [71], Thüm *et al.* define a feature as abstract, "if and only if it is not mapped to any implementation artifacts". They "call all other features non-abstract or concrete, i.e., a concrete feature is mapped to at least one implementation artifact". It corresponds to our case.

[10] Many constraints of fm_{Arch} involve features tinfi_oo, osgi_provider, julia, thus confirming that their parent feature fractal_bootstrap_class_provider is needed in many configurations.

architect to understand the impact of the difference: it is either an implication unintentionally not specified or an implication not documented by plugin dependencies. The major advantage of the structure of binary implication graph is the ability to derive *transitive* implications.

We identified 9 implies constraints expressed in fm_{Arch} but not in fm_{SA}. All constraints were validated by the SA, recognizing that the constraints have been forgotten. Furthermore, we observed that the 7 implies constraints originally expressed in fm_{SA} are already induced by fm_{Arch}. To avoid redundancy, we did not add them.

Step-wise Refinement. Based on the comparison results, the SA had several attitudes:

- firstly, he used fm_{Arch} to *verify* the coherence of his original variability specification fm_{SA};
- secondly, he considered that some variability decisions in fm_{SA} (resp. fm_{Arch}) are correct despite their differences with fm_{Arch} (resp. fm_{SA});
- thirdly, he edits the two FMs by adding some constraints only present in fm_{Arch} or by setting the variability.

Edits have been applied on both FMs. The comparison and editing techniques have been reiterated until obtaining a *refactoring* (see Definition 8), i.e., where no differences occur between fm_{SA} and fm_{Arch}.

7.2.3 Evolutions (versions 1.4 and 1.5) We applied the extraction procedure as well as the comparison techniques for other versions (1.4 and 1.5) of FraSCAti architecture.

Properties of FMs.

Table 1 summarizes the properties of the input FMs ($fm_{Arch_{150}}$ and fm_{Plug}), of the mapping between the two FMs and the qualitative deduction made by the slicing (i.e., number of configurations of fm_{Arch}, deduction of core features, implies and bi-implies constraints).

New features in the architectural and in the plugin parts have been added between version 1.3 (resp. 1.4) and version 1.4 (resp. 1.5) as well as constraints. We can notice that the number of plugins increased much more than the number of features in $fm_{Arch_{150}}$.

We observed similar benefits than with version 1.3 of FraSCAti. The slice architectural FM significantly reinforces the set of configurations over-approximated by the 150% FM.

Reuse of reconciliation and SA knowledge.

The FM fm_{Arch_2} of the version 1.4 of FraSCAti has been compared with the FM obtained at the end of the refinement process $fm_{Arch'_1}$, corresponding to version 1.3 of FraSCAti (see Fig. 6). A typical problem occurring in such situation is that $fm_{Arch'_1}$ and fm_{Arch_2} are not correctly reconciled. Instead of performing a new reconciliation process or modifying the extraction algorithm, we simply reused FAMILIAR directives (i.e., edits) used for the version 1.3 of FraSCAti. Another benefit is that the SA knowledge can be reused (e.g., for retaining mandatory features in the model or removing obsolete features[11]).

Understanding the evolutions. The evolutions of FraSCAti architecture consist in the introduction of new features (no features have been removed) while the same naming convention and structure of the architecture have been kept.

The three new features of version 1.4 include two core features (child features of the root) jmx, fscript and the feature frascati_implementation_resource, providing an alternative implementation for the FraSCAti architecture. We computed the kind of relationship between fm_{Arch_2} with $fm_{Arch'_1}$, leading to an arbitrary edit (see Definition 8). It is not surprising since new core features jmx, fscript preclude all previous configurations of fm_{Arch_2}. We used the slice to safely remove those features (we kept frascati_implementation_resource) and we obtained that fm_{Arch_2} is a specialization of $fm_{Arch'_1}$. The SA validates the kind of relationship because it exactly corresponds to what he had in mind: all previous valid configurations of fm_{Arch_2} have an equivalence in $fm_{Arch'_1}$ (i.e., all configurations of $fm_{Arch'_1}$ augmented with the core features jmx, fscript).

The SA validated the new set of configurations induced by the architectural evolution and corresponding to configurations expressed in fm_{Arch_2} but not originally valid in $fm_{Arch'_1}$. The diff operator (see Definition M_1, page 14) indeed revealed that all new configurations include the new feature frascati_implementation_resource.

For the version 1.5 of FraSCAti, 7 new features have been added (compared to version 1.4) and some new constraints. Features provide new functionalities such as new metamodels, implementations and bindings. All of them are optional in the architectural FM. We computed the kind of relationship between the version 1.4 and 1.5, and obtained an arbitrary edit.

As previously, the kind of relationship is counter-intuitive since the new features of version 1.5 disturb the comparison operator. To overcome this problem, we sliced the two FMs – using as slicing criteria *all* features not included in both FMs – and re-computed the kind of relationship. This time, we obtained a generalization, meaning that the FM of version 1.4 supports more configurations than the FM of version 1.5.

[11] The information gained during the reverse engineering process for the version 1.3 of FraSCAti could have been used by the SA to modify the FraSCAti artefacts or refactor the architecture. Nevertheless, such changes have not been made in the project. Therefore versions 1.4 and 1.5 share a lot of properties of version 1.3, including mandatory or obsolete features.

Version	$fm_{Arch_{150}}$	fm_{Plug}	mapping	fm_{Arch}	core features (deduced)	implies constraints (deduced)	bi-implies constraints (deduced)
1.3	50 features $\approx 10^{11}$ config.	41 features 81 constraints	78 constraints	$\approx 10^{6}$ config.	12	9	5
1.4	53 features $\approx 10^{11}$ config.	56 features 87 constraints	80 constraints	$\approx 10^{7}$ config.	12	10	5
1.5	60 features $\approx 10^{14}$ config.	63 features 96 constraints	92 constraints	$\approx 10^{8}$ config.	12	13	7

Table 1 Experimental results: properties of the FMs

To better understand why and validate this evolution, we computed the diff of binary implication graph and we identified two implies constraints, expressed in the version 1.5 but not in the version 1.4: frascati_implementation_script ⇒ fractal_bootstrap_class_providers and frascati_implementation_script ⇒ julia. The constraints explained why the FM of version 1.4 is less restrictive than the FM of version 1.5. The SA validated the evolution of version 1.5. He indeed considered that the two constraints have been improperly missed in version 1.4.

Refactoring opportunities. By analyzing the evolution of FraSCAti for the versions 1.4 and 1.5, we observed that much more features have been added in fm_{Plug} than in $fm_{Arch_{150}}$. It means that the architecture have not yet integrated the new functionalities or that the architecture should be modified to explicitly support them. Such differences will be considered and exploited in the future of the FraSCAti project in order to further enhance the flexibility of the architecture.

7.2.4 Assessment Based on our experiments with different versions of the FraSCAti project, we can draw some conclusions w.r.t **(RQ1)** and **(RQ2)**:

- **Extraction of variability** The extraction procedure deduces many constraints and drastically restricts the configuration set of $fm_{Arch_{150}}$. The SA validates the variability recovered by the procedure. It even encourages him to correct his initial model. We gain better confidence in the accuracy of the extraction procedure by reiterating the process on different versions of FraSCAti. In some specific cases though the extracted FM contains faulty variability information. In this case, we have to rely on the knowledge of the SA.
- **Evolution of variability** The differencing techniques appear to be meaningful for the SA. It allows the SA to control the properties of extracted FMs and in turn integrate his knowledge. It also allows the SA to understand and validate the evolutions of the FraSCAti architecture, for example, by controlling what implies constraints have been added and removed between two versions.

7.3 Threats to Validity

Internal Validity. A first threat to internal validity is the reliability of the proposed techniques. For example, a faulty FM management support might bias the validation of the extraction procedure by the SA. Our implementation of FAMILIAR is currently checked by a comprehensive set of tests. We also manually verified a large number of examples. Another internal threat is that our approach is semi-automated. The manual part of the study was conducted by the SA. His choices, interpretation and possible errors might have influenced the results. To mitigate this threat, the authors of the article interacted to assist the SA in using the tools and explaining the differences.

External Validity There are threats to external validity that limit our ability to generalize the results and their application in other contexts. We only consider one plugin-based system. We may not find in other plugin-based systems the same characteristics of the FraSCAti project. In particular, the SA has a strong experience and an in-depth understanding of the implemented architecture, which finally eases the alignment process. This may not be the case in other architecture-based software. The automatically extracted FM could then be more difficult to align with the SA view, the latter possibly belonging to different levels of abstraction, completeness and precision, or relying on a different vocabulary. Nevertheless, the automatically extracted FM will always constitute an accurate variability model of the system architecture, as it is (currently) implemented.

Another concern is whether FraSCAti is representative of plugin-based systems used in industry. Its representativity can be studied from two points of views: Are the used software architecture technologies representative? and is the size of the FraSCAti system representative? On the one hand, the FraSCAti architecture relies on two standards (i.e., SCA and Maven) strongly used in industry. SCA is standardized by the world-wide industrial OASIS consortium and is more and more adopted by software projects in industry like IBM WebSphere

Application Server, Eclipse SCA Tools, Apache Tuscany, etc. Apache Maven is the pillar build system used by many industrial software systems. On the other hand, FraSCAti is a system composed of more than 60 plugins and 100 components. To our knowledge, few plugin-based systems (e.g., Eclipse, Nuxeo) contain more plugins/components than FraSCAti, but FraSCAti is the bigger SCA-based application, i.e., with the higher number of plugins/components. Then, we consider that FraSCAti is representative of industrial plugin-based systems.

But these systems require to expose an explicit description of these plugins, their dependencies, and the components they contain, in order to extract their architectural FM automatically. If the explicit description is not expressed with SCA and Maven, then the extraction process (both Section 4.1 and 4.2) requires to be slightly adapted to parse these descriptors. Then most of the algorithms detailed in this article can be reused as it for this kind of plugin-based systems. However more research effort is needed to obtain evidence that our proposals could be applied on non SCA/Maven plugin-based systems, e.g., Eclipse or OSGi ones.

Other kinds of systems (i.e., not necessarily based on plugins) might benefit from our proposals. For example, the Linux kernel faces similar evolution problems (see next section). Yet more research effort is needed to obtain evidence.

8 Related Work

As part of our approach, many operations are needed for extracting and managing the evolution of architectural variability (see Section 4 and Section 5). In the first part of this section, we specifically compare the proposed FM management support with state-of-the-art techniques. In the second part of this section, we review other related works.

8.1 FM Management Support

8.1.1 Extraction: Combining FMs. The extraction process consists in synthesizing a new FM based on two variability sources. There are many attempts to compose and decompose FMs (also called FM *views*).

The original contribution of our work is that we combine the two mechanisms. The composition mechanism (aggregation) is first used to obtain an integrated FM of the two variability sources – it can be seen as a *temporary* FM. The decomposition mechanism is applied afterwards to synthesize the result, projecting the constraints of one FM to the other FM. We now review existing works in the area of FM composition and decomposition.

Composition. A few works consider some forms of composition for FMs [69,9,63,61,28,35,48,28,72,56,58]. They can be used to inter-relate several FMs (like $fm_{Arch_{150}}$ and fm_{Plug}) through constraints. None of them though propose to perform over this composed FM and, in particular, do not propose any projection mechanism. We go further and propose to combine the aggregate operator with the slicing operator for synthesizing fm_{Arch} for each version of FraSCAti.

Decomposition. In the context of feature-based configuration, techniques have been proposed to separate the configuration process in different steps or stages [19]. Hubaux et al. provide view mechanisms to decompose a large FM [32]. However they do not propose a comprehensive solution when dealing with cross-tree constraints. This is particularly important in the FraSCAti case study.

Schroeter *et al.* [62] propose mechanisms to support multi-perspectives on FMs. Their main interest is on guaranteeing *consistency* of perspectives w.r.t. configuration semantics. In the FraSCAti case study, we are interested in *synthesizing* a new perspective (i.e., fm_{Arch}) based on two perspectives ($fm_{Arch}150$ and fm_{Plug}).

8.1.2 Evolution: FM Differences. We now review two techniques that are part of the toolbox for FM differences (see Section 5.2).

Kinds of edit between two FMs. Thüm et al. [70] presented an automated and scalable algorithm to characterize the kinds of edit between two FMs (see Definition 8, page 13). In case the relationship is not a refactoring, they propose a technique to generate an example of configuration authorized in one but not in another. The techniques can be used in the context of FM differences and are part of our tooling support but have some limitations.

First, the kind of relationship between two FMs does not help to precisely *understand* the impact of a change, for example, what implies or excludes constraints have been removed and added.

The technique does not compute *all* added and removed configurations. We can compute a diff FM that compactly represents all added and removed configurations. This model can be analyzed (e.g., enumeration of all configurations), visualized or serialized.

Moreover, reasoning about the relationship of two FMs is inappropriate until FMs are not reconciled: our experience shows that pre-directives have to be applied before. In particular, we rely on the slicing operator for removing unnecessary details since basic manual edits of FMs are not appropriate.

Quotient. In [26], Fahrenberg *et al.* propose an algorithm to compute the *quotient* (see page 14). As recognized, the quotient is an approximation of the differ-

ences between two FMs whereas the diff FM is not. Another limitation is that the quotient is a set of disjunctive clauses that are difficult to understand for a practitioner. In practice, an additional step is necessary to transform these clauses into a more readable and manageable information, closer to FM constructs. From our experience, the method of quotient produces some disjunctive clauses that can be transformed into implications. Nevertheless, we observed many times that the method suffers from a lack of completeness regarding the diff of implies constraints. The diff of binary implication graphs reports more meaningul differences to the SA in the FraSCAti case study.

Divide and Conquer. Understanding FM differences can quickly become difficult for a SA when a large number of features and constraints are involved. Therefore we intensively used the slicing operator to decompose a typically large FM into sub-FMs. Differencing techniques, including the two previously described, are then applied afterwards. This divide-and-conquer strategy was crucial to understand local and fine-grained differences of FraSCAti architectural FMs.

8.2 Other Related Work

Despite the importance of variability in software systems in general, and in software architectures in particular [13], the problem of managing the architectural variability of *existing* systems has definitely not received sufficient attention from the research community. The variability management exposes two important activities: the extraction (also called recovery or reverse engineering) of variability as well as its evolution. In this section, we review existing works related to architectural variability, to its extraction and its evolution as well as its realization.

8.2.1 Extraction of Architectural Variability

Extraction of variability in general. While our work takes an architectural perspective, the other existing approaches in the field consider different input artifacts. Some approaches deal directly with source code, handling it with clone detection [74] or intermediate models built with construction primitives [77]. Some others are based on legacy system documentation [34], textual requirements [10], or identification of similar elements in specific models [76,59]. General similarity detection is also explored with syntactic techniques on source code [24], which only provide similarity information with no feature extraction, or with semantic techniques based on Formal Concept Analysis [38,75,59]. All these approaches share the usage of a *single form of input* to extract variability information. In their recent work, She *et al.* [66] propose a reverse engineering approach combining two distinct sources of information: textual feature descriptions and feature dependencies. They developed an efficient synthesis procedure to compute variability information (e.g., feature groups) and proposed heuristics for identifying the most likely parent feature candidates of each feature. Our approach also benefits from the combination of two (other) sources of information, namely plugin dependencies and architecture fragments. 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 article reuse and extend previous work in SPL engineering [14]. Metzger *et al.* [48] propose an approach to cross-checking *product-line* variability and *software* variability models, thus assuming that such models (or views) are available. Janota *et al.* propose a theoretical foundation for checking that an FM does not allow feature configurations not realizable by the architecture [33]. Lopez and Eyged [39] 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. Our approach is complementary since it allows the recovering of the *actually supported* variability of a software system, and since combines architectural and plugin FMs. One of the key component and original contribution of our work is the slicing operator we have defined and realized (see Section 4). It allows one to *project* software variability and constraints onto an architectural model.

Architecture and design recovery. This work can also be seen as a contribution to the broader theme of software architecture recovery (or reconstruction), a recent process-oriented survey is presented in [23]. Although our approach takes as input an *explicit* description of an extensible architecture, it also allows the recovery of *implicit* architectural knowledge, among which undocumented dependencies between components that are identified through the projection of inter-plugin dependencies towards the architectural feature model. The main difference of our work with respect to the architecture recovery literature resides in our original focus on architectural variability. Our goal is not to recover the complete architecture of the system, but rather to extract an accurate variability model expressing the exact set of *valid* architectural configurations that can be obtained, in the context of this article, from the composition of several system plugins. As a positive side-effect, the variability model obtained aims to enable easier configuration and safer composition from the end-user perspective. Easier configuration, because system configurations may now be expressed in terms of fine-grained system features (services), rather than as a set of possibly obscure plugins to be composed. Safer composition,

since only valid combinations of features can now been selected, therefore preventing unsafe system variants to be composed.

8.2.2 Evolution of Architectural Variability

Evolution of SPLs in general. In [22,53], Dhungana *et al.* report that evolution support becomes particularly important for engineering SPLs and other variability-intensive systems. They propose model-driven support at the feature level, using FM concepts [53]. They developed a set of operators to make evolve FMs but no *reasoning and differencing techniques* are proposed to control the evolution of the FMs. This is particularly important in our context to understand the impact of the evolution of an FM. Lotufo *et al.* study the evolution of the Linux kernel variability model [41]. They identify edit operations applied in practice and new automation challenges, including the detection of edits that break existing configurations. FM management support is also needed (e.g., to identify what are the added and removed configurations). In particular the differencing techniques exposed in this article can be reused in such context. We leave it as future work.

Reasoning about evolution. Model differencing is an important technique to manage (e.g., understanding, maintenance) evolutions of models. It has attracted research efforts in recent years, including the development of tools (e.g., see [51,37,44,44,42,43]). The bibliography [51] compiles about 300 publications in this field. Existing approaches mainly focus on *syntactical* differences. As argued in [44,26], models (e.g., FMs) that are syntactically very similar may induce very different semantics and a list of differences should be best addressed *semantically*. Recently, Maoz *et al.* tackled the problem of semantic model differencing, specifically for class and activity diagrams [42,43]. They defined and implemented two versions of semantic diff operator, *cd-diff* and *addiff*. These two contributions are specific to the semantics of class and activity diagrams. Therefore they cannot be applied in our context where we need to reason about differences of FMs.

In the field of feature modeling, Benavides et al. [14] survey a set of operations and techniques proposed for automated analysis of FMs. In this survey, no automated techniques are reported to reason about or compute differences. Two notable exceptions are the algorithms described in [70,26]. We compared and discussed these algorithms in Section 8.1, showing that they are not sufficient in the FraSCAti case study. Segura et al. propose a catalog of rules for merging FMs (union and intersection) [63]. They present syntactic mechanisms that have limitations (see a comparison in [4]) and no diff operator is considered.

9 Conclusion

Variability modeling and management is of crucial importance in the management of software systems. It is particular the case for extensible software systems, typically built on top of plugin-based architectures that offer a large number of configuration options through plugins. 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. Furthermore, as variability intensive systems and their variability evolve over time, there is an increasing need of more automated, accurate and reproducible procedures to derive feature models.

9.1 Summary of Contributions

In this article, we presented a tool-supported approach to extract and manage the evolution of software variability from an architectural perspective:

- **extraction:** the process involves the automatically supported extraction, aggregation, alignment and slicing of architectural feature models. It has the merit of combining several sources of information, namely software architecture, plugin dependencies and software architect knowledge. As a result, we contribute to projecting the implementation constraints (expressed by plugin dependencies) onto an architectural model.
- **evolution:** the process enables the software architect to validate the extracted feature models and incorporate if needs be his/her knowledge through step-wise refinement. We contributed to an advanced support for aligning concepts, computing differences between two versions of the architectural feature model, and editing feature models.

We 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, especially in the case of several successive versions of an evolving software architecture. We also showed that without the feature model management support exposed in this article, some analysis and reasoning operations (e.g., differencing techniques, slicing) would not be made possible.

9.2 Key Insights

The FraSCAti case study provides us with interesting insights into the reverse engineering of architectural feature models.

Firstly, although the gap between the intentional variability representations of the software architect and the

extracted feature models appears to be manageable (due to a rather important similarity between the feature models), it remains necessary to assist the software architect with automated support:

- the use of Levenshtein distance was sufficient to establish correspondences between features' names of the two feature models. We did not need an advanced *matching* support already integrated in model-based tools;
- the most time-consuming task was to deal with the level of details in both feature models. For this specific activity, tool supported, advanced techniques, such as the safe removal of a feature by slicing, are not desirable but mandatory (i.e., basic manual edits of FMs are not sufficient);
- automated differencing techniques are crucial to integrate the software architect knowledge and validate an evolution. A manual inspection is not feasible and the techniques proposed in the literature are useful but not sufficient.

Secondly, our extraction procedure yields very promising results. It recovers most of the variability originally specified by the software architect and encourages him to correct his initial model. A manual checking of numerous variability decisions imposed by the software architect shows that the extraction is not faulty and in line with the intention of the software architect. We reiterated the extraction procedure on different versions of FraSCAti and performed similar observations, thus gaining better confidence.

Thirdly, the software architect knowledge is required *i)* to scope the architecture (e.g., by restricting the set of configurations of the extracted feature model), especially when software artefacts do not correctly document the variability of the system and *ii)* to control and validate the automated procedure. As a result, we consider that fully automating the extraction process is neither realistic nor desirable for the next versions of FraSCAti. The integration of the software architect knowledge, along different evolutions of architectural feature models of FraSCAti, can be time-consuming and error-prone. It encourages us to develop support for assisting the software architect in reusing his previous engineering effort for older versions of FraSCAti.

9.3 Future Work

As future work, we plan to couple the presented reverse engineering process to more common forward engineering procedures. For example, the improvements made on the architectural feature model should allow the software architect to derive safer architecture variants and configurators for users.

Moreover the integration of our extraction and reasoning procedures with existing model-driven approaches to SPL engineering and evolution [54] should be further investigated, with the ultimate goal to ease the evolution of large families of software systems. In this context, we intend to couple the evolution of the architectural feature models to software development tracking information, such as commits in version control systems. Another interesting direction is to apply testing techniques to control that the variants allowed by the feature model are really safe at compile or at runtime. It could be an alternative and complementary way to validate the feature models we have reverse engineered [31].

In the long term, we plan to apply our techniques to other (kinds of) architectures and software projects (e.g., Linux). We hope the principles and the feature model management support we present in this article can be successfully applied for managing the evolution of variability-rich architectures.

References

1. `https://nyx.unice.fr/projects/familiar/wiki/FraSCAti`.
2. M. Acher, A. Cleve, P. Collet, P. Merle, L. Duchien, and P. Lahire. Reverse Engineering Architectural Feature Models. In *Proceedings of the 5th European Conference on Software Architecture (ECSA 2011)*, LNCS, pages 220–235. Springer, 2011.
3. M. Acher, A. Cleve, G. Perrouin, P. Heymans, P. Collet, P. Lahire, and C. Vanbeneden. On Extracting Feature Models From Product Descriptions. In *Proceedings of the 6th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2012)*, pages 45–54. ACM, January 2012.
4. M. Acher, P. Collet, P. Lahire, and R. France. Comparing Approaches to Implement Feature Model Composition. In *Proceedings of the 6th European Conference on Modelling Foundations and Applications (ECMFA 2010)*, volume 6138 of *LNCS*, pages 3–19, 2010.
5. M. Acher, P. Collet, P. Lahire, and R. France. Separation of concerns in feature modeling: Support and applications. In *Proceedings of the 11th International conference on Aspect-Oriented Software Development (AOSD 2012)*, pages 1–12. ACM, 2012.
6. M. Acher, P. Collet, P. Lahire, and R. France. FAMILIAR: A Domain-Specific Language for Large Scale Management of Feature Models. *Science of Computer Programming (SCP) – Special issue on programming languages (to appear)*, 2013.
7. M. Acher, P. Heymans, C. Quinton, P. Merle, P. Collet, and P. Lahire. Feature model differences. In *Proceedings of the 24th International Conference on Advanced Information Systems Engineering (CAiSE'12)*, pages 629–645. Springer, 2012.
8. Mathieu Acher, Patrick Heymans, Anthony Cleve, Jean-Luc Hainaut, and Benoit Baudry. Support for reverse engineering and maintaining feature models. In *Proceedings of the 7th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'13)*, Pisa, Italie, jan 2013. ACM.

9. V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, and C. Lucena. Refactoring product lines. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE 2006)*, pages 201–210. ACM, 2006.
10. 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 *Proceedings of the 12th International Software Product Lines Conference (SPLC 2008)*, pages 67–76. IEEE, 2008.
11. Nele Andersen, Krzysztof Czarnecki, Steven She, and Andrzej Wasowski. Efficient synthesis of feature models. In *Proceedings of SPLC'12*, pages 97–106. ACM Press, 2012.
12. S. Apel and C. Kästner. An overview of feature-oriented software development. *Journal of Object Technology (JOT)*, 8(5):49–84, July/August 2009.
13. F. Bachmann and L. Bass. Managing variability in software architectures. *SIGSOFT Softw. Eng. Notes*, 26:126–132, May 2001.
14. D. Benavides, S. Segura, and A. Ruiz-Cortes. Automated Analysis of Feature Models 20 years Later: a Literature Review. *Information Systems*, 2010.
15. P. Bille. A survey on tree edit distance and related problems. *Theoretical Computer Science*, 337(1-3):217–239, 2005.
16. A. Classen, Q. Boucher, and P. Heymans. A text-based approach to feature modelling: Syntax and semantics of TVL. *Science of Computer Programming, Special Issue on Software Evolution*, 2010.
17. P. Clements and L. M. Northrop. *Software Product Lines : Practices and Patterns*. Addison-Wesley Professional, 2001.
18. K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
19. K. Czarnecki, S. Helsen, and U. Eisenecker. Staged Configuration through Specialization and Multilevel Configuration of Feature Models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005.
20. K. Czarnecki and K. Pietroszek. Verifying feature-based model templates against well-formedness OCL constraints. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE 2006)*, pages 211–220. ACM, 2006.
21. K. Czarnecki and A. Wasowski. Feature diagrams and logics: There and back again. In *Proceedings of 11th International Software Product Lines Conference (SPLC 2007)*, pages 23–34, 2007.
22. D. Dhungana, P. Grünbacher, R. Rabiser, and T. Neumayer. Structuring the modeling space and supporting evolution in software product line engineering. *Journal of Systems and Software*, 83(7):1108–1122, 2010.
23. S. Ducasse and D. Pollet. Software architecture reconstruction: A process-oriented taxonomy. *IEEE Trans. Software Eng.*, 35(4):573–591, 2009.
24. S. Duszynski. A scalable goal-oriented approach to software variability recovery. In *Proceedings of the 15th International Software Product Line Conference, Volume 2*, SPLC '11, pages 42:1–42:8, New York, NY, USA, 2011. ACM.
25. J. Euzenat and P. Shvaiko. *Ontology matching*. Springer-Verlag, Heidelberg, 2007.
26. U. Fahrenberg, A. Legay, and A. Wasowski. Vision paper: Make a difference! (semantically). In *Proceedings of the ACM/IEEE 14th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2011)*, pages 490–500, 2011.
27. P. Grünbacher, R. Rabiser, D. Dhungana, and M. Lehofer. Model-Based Customization and Deployment of Eclipse-Based Tools: Industrial Experiences. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 247–256, Washington, DC, USA, 2009. IEEE Computer Society.
28. H. Hartmann and T. Trew. Using feature diagrams with context variability to model multiple product lines for software supply chains. In *Proceedings of 12th International Software Product Lines Conference (SPLC 2008)*, pages 12–21. IEEE, 2008.
29. E. N. Haslinger, R. E. Lopez-Herrejon, and A. Egyed. Reverse engineering feature models from programs' feature sets. In *WCRE'11*, pages 308–312. IEEE CS, 2011.
30. Florian Heidenreich, Pablo Sanchez, Joao Santos, Steffen Zschaler, Mauricio Alferez, Joao Araujo, Lidia Fuentes, Uira Kulesza amd Ana Moreira, and Awais Rashid. Relating feature models to other models of a software product line: A comparative study of featuremapper and vml*. *Transactions on Aspect-Oriented Software Development VII, Special Issue on A Common Case Study for Aspect-Oriented Modeling*, 6210:69–114, 2010.
31. Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. Towards automated testing and fixing of re-engineered feature models. In *Proceedings of ICSE'13, NIER track*, pages 1245–1248, 2013.
32. A. Hubaux, P. Heymans, P.-Y. Schobbens, D. Deridder, and E. K. Abbasi. Supporting multiple perspectives in feature-based configuration. *Software and Systems Modeling (SoSyM)*, pages 1–23, 2011.
33. M. Janota and G. Botterweck. Formal approach to integrating feature and architecture models. *Fundamental Approaches to Software Engineering (FASE)*, pages 31–45, 2008.
34. I. John. Capturing product line information from legacy user documentation. In *Software Product Lines*, pages 127–159. Springer, 2006.
35. K. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. Form: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5(1):143–168, 1998.
36. C. Kästner, T. Thüm, G. Saake, J. Feigenspan, T. Leich, F. Wielgorz, and S. Apel. FeatureIDE: Tool Framework for Feature-Oriented Software Development. In *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009)*, 2009. Formal Demonstration paper.
37. D.S. Kolovos, D. Di Ruscio, A. Pierantonio, and R.F. Paige. Different models for model matching: An analysis of approaches to support model differencing. In *Proceedings of the International Workshop on Comparison and Versioning of Software Models (CVSM 2009)*, pages 1 –6, may 2009.

38. F. Loesch and E. Ploedereder. Restructuring variability in software product lines using concept analysis of product configurations. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, CSMR '07, pages 159–170, Washington, DC, USA, 2007. IEEE Computer Society.
39. R. E. Lopez-Herrejon and A. Egyed. Detecting inconsistencies in multi-view models with variability. In *Proceedings of the 6th European Conference on Modelling Foundations and Applications (ECMFA 2010)*, pages 217–232, 2010.
40. R. E. Lopez-Herrejon and A. Egyed. On the need of safe software product line architectures. In *Proceedings of the 4th European Conference on Software Architecture (ECSA 2010)*, volume 6285 of *LNCS*, pages 493–496. Springer, 2010.
41. R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wasowski. Evolution of the linux kernel variability model. In *Proceedings of the 14th International Conference on Software Product Lines (SPLC 2010)*, pages 136–150, 2010.
42. S. Maoz, J. O. Ringert, and B. Rumpe. Addiff: semantic differencing for activity diagrams. In *Proceedings of the 2011 joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2011)*, pages 179–189. ACM, 2011.
43. S. Maoz, J. O. Ringert, and B. Rumpe. Cddiff: semantic differencing for class diagrams. In *Proceedings of the 25th European Conference on Object-Oriented Programming (ECOOP 2011)*, pages 230–254. Springer, 2011.
44. S. Maoz, J. O. Ringert, and B. Rumpe. A manifesto for semantic model differencing. In *Proc. of MODELS'10*, pages 194–203. Springer, 2011.
45. M. Mendonca, M. Branco, and D. Cowan. S.P.L.O.T.: software product lines online tools. In *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2009)*, pages 761–762. ACM, 2009.
46. M. Mendonca, A. Wasowski, K. Czarnecki, and D. Cowan. Efficient compilation techniques for large scale feature models. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering (GPCE 2008)*, pages 13–22. ACM, 2008.
47. P. et al. Merle. OW2 FraSCAti Web Site, 2008. `http://frascati.ow2.org`.
48. 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 *Proceedings of the 15th IEEE International Requirements Engineering Conference (RE 2007)*, pages 243–253, 2007.
49. G. C. Murphy, D Notkin, and K. J. Sullivan. Software reflexion models: Bridging the gap between design and implementation. *IEEE Trans. Softw. Eng.*, 27(4):364–380, April 2001.
50. OASIS. Service Component Architecture, 2007. `http://www.oasis-opencsa.org/sca/`.
51. Bibliography on Comparison and Versioning of Software Models. `http://pi.informatik.uni-siegen.de/CVSM`.
52. C. A. Parra, A. Cleve, X. Blanc, and L. Duchien. Feature-based composition of software architectures. In *Proceedings of the 4th European Conference on Software Architecture (ECSA 2010)*, volume 6285 of *LNCS*, pages 230–245. Springer, 2010.
53. A. Pleuss, G. Botterweck, D. Dhungana, A. Polzer, and S. Kowalewski. Model-driven support for product line evolution on feature level. *Journal of Systems and Software*, 2011.
54. A. Pleuss, G. Botterweck, D. Dhungana, A. Polzer, and S. Kowalewski. Model-driven support for product line evolution on feature level (available online, in press). *Journal of Systems and Software (JSS)*, 2011. http://dx.doi.org/10.1016/j.jss.2011.08.008.
55. A Rabkin and R Katz. Static extraction of program configuration options. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*, pages 131–140. ACM, 2011.
56. M.-O. Reiser and M. Weber. Multi-level feature trees: A pragmatic approach to managing highly complex product families. *Requirements Engineering*, 12(2):57–75, 2007.
57. Lero research center. S2t2 configurator. http://download.lero.ie/spl/s2t2/.
58. M Rosenmüller, N Siegmund, T Thüm, and G Saake. Multi-dimensional variability modeling. In *VaMoS'11*, pages 11–20. ACM, 2011.
59. U. Ryssel, J. Ploennigs, and K. Kabitzsch. Automatic variation-point identification in function-block-based models. In *Proceedings of the ninth international conference on Generative programming and component engineering*, GPCE '10, pages 23–32, New York, NY, USA, 2010. ACM.
60. D. Schneeweiss and G. Botterweck. Using flow maps to visualize product attributes during feature configuration. In *Proceedings of the 3rd International Workshop on Visualisation in Software Product Line Engineering (ViSPLE 2010)*, 2010.
61. P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 51(2):456–479, 2007.
62. J. Schroeter, M. Lochau, and T. Winkelmann. Multi-perspectives on feature models. In Robert B. France, Jürgen Kazmeier, Ruth Breu, and Colin Atkinson, editors, *MoDELS*, volume 7590 of *Lecture Notes in Computer Science*, pages 252–268. Springer, 2012.
63. S. Segura, D. Benavides, A. Ruiz-Cortes, and P. Trinidad. Automated merging of feature models using graph transformations. *Post-proceedings of the Second Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2007)*, 5235:489–505, 2008.
64. L. Seinturier, P. Merle, D. Fournier, N. Dolet, V. Schiavoni, and J.-B. Stefani. Reconfigurable SCA Applications with the FraSCAti Platform. In *Proceedings of the 2009 IEEE International Conference on Services Computing (SCC 2009)*, pages 268–275. IEEE, 2009.
65. L. Seinturier, P. Merle, R. Rouvoy, D. Romero, V. Schiavoni, and J.-B. Stefani. A Component-Based Middleware Platform for Reconfigurable Service-Oriented Architectures. *Software: Practice and Experience*, 42(5):559–583, May 2012.

66. S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. Reverse engineering feature models. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011)*, pages 461–470. ACM, 2011.
67. M. Svahnberg, J. van Gurp, and J. Bosch. A taxonomy of variability realization techniques: Research articles. *Software Practice and Experience*, 35(8):705–754, 2005.
68. S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe composition of product lines. In *Proceedings of the 6th International Conference on Generative Programming and Component Engineering (GPCE 2007)*, pages 95–104. ACM, 2007.
69. J. M. Thompson and M. P. E. Heimdahl. Structuring product family requirements for n-dimensional and hierarchical product lines. *Requirements Engineering*, 8(1):42–54, February 2003.
70. T. Thüm, D. Batory, and C. Kästner. Reasoning about edits to feature models. In *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009)*, pages 254–264. ACM/IEEE, 2009.
71. T. Thüm, C. Kästner, S. Erdweg, and N. Siegmund. Abstract features in feature modeling. In Eduardo Santana de Almeida, T. Kishi, C. Schwanninger, I. John, and K. Schmid, editors, *Proceedings of the 15th International Conference on Software Product Lines (SPLC 2011)*, pages 191–200, 2011.
72. T. T. Tun, Q. Boucher, A. Classen, A. Hubaux, and P. Heymans. Relating requirements and feature configurations: A systematic approach. In *Proceedings of the 13th International Software Product Lines Conference (SPLC 2009)*, pages 201–210. IEEE, 2009.
73. N. Weston, R. Chitchyan, and A. Rashid. A framework for constructing semantically composable feature models from natural language requirements. In *Proceedings of 13th International Software Product Lines Conference (SPLC'09)*, pages 211–220. ACM, 2009.
74. Y. Xue. Reengineering legacy software products into software product line based on automatic variability analysis. In Richard N. Taylor, Harald Gall, and Nenad Medvidovic, editors, *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 1114–1117. ACM, 2011.
75. Y. Yang, X. Peng, and W. Zhao. Domain feature model recovery from multiple applications using data access semantics and formal concept analysis. In *Proceedings of the 2009 16th Working Conference on Reverse Engineering*, WCRE '09, pages 215–224, Washington, DC, USA, 2009. IEEE Computer Society.
76. X. Zhang, O. Haugen, and B. Moller-Pedersen. Model comparison to synthesize a model-driven software product line. In *Proceedings of the 2011 15th International Software Product Line Conference*, SPLC '11, pages 90–99, Washington, DC, USA, 2011. IEEE Computer Society.
77. T. Ziadi, L. Frias, M. A. Almeida da Silva, and M. Ziane. Feature identification from the source code of product variants. In *16th European Conference on Software Maintenance and Reengineering, CSMR 2012, Szeged, Hungary, March 27-30, 2012*, pages 417–422. IEEE, 2012.