Feature-based Composition of Software Architectures

Carlos Parra, Anthony Cleve, Xavier Blanc, Laurence Duchien

Feature-based Composition of Software Architectures

Carlos Parra, Anthony Cleve, Xavier Blanc, and Laurence Duchien

INRIA Lille-Nord Europe, LIFL CNRS UMR 8022,
Université des Sciences et Technologies de Lille, France
{carlos.parra, anthony.cleve, xavier.blanc, laurence.duchien}@inria.fr

Abstract. In Software Product Lines variability refers to the definition and utilization of differences between several products. Feature Diagrams (FD) are a well-known approach to express variability, and can be used to automate the derivation process. Nevertheless, this may be highly complex due to possible interactions between selected features and the artifacts realizing them. Deriving concrete products typically involves the composition of such inter-dependent software artifacts. This paper presents a feature-based composition approach to automatically derive a product architecture from a given feature configuration. The proposed approach relies on the combination of Model-Driven Engineering (MDE) and Aspect-Oriented Modeling (AOM) techniques. We introduce a metamodel to reify each feature as a high-level aspect model. Product derivation is achieved by weaving the set of aspect models corresponding to a particular feature configuration. The weaving strategy is derived from an in-depth cross-analysis of both the feature interactions and the aspect model dependencies.

1 Introduction

One of the most important challenges of Software Product Line Engineering concerns variability management, i.e., how to describe, manage and implement the commonalities and variabilities existing among the members of the same family of software products. A well-known approach to variability modeling is by means of Feature Diagrams (FD) introduced as part of Feature Oriented Domain Analysis (FODA) [1] back in 1990. An FD typically consists of (1) a hierarchy of *features*, which may be *mandatory* (commonality) or *optional* (variability), and (2) a set of *constraints* expressing inter-feature dependencies. Nevertheless, deriving a concrete software product from an FD remains a highly complex process. The latter starts with the *feature configuration* step, which aims at selecting the features to include in the desired product, in strict conformance to the specified constraints. The product derivation process then necessitates the *composition* of the *software artifacts* corresponding to the selected features. This second step may be very challenging, since the fact of selecting a single feature may impact several several places in the product itself.

In order to enable the automated derivation of a product in an SPL, it is necessary to specify the corresponding artifacts that reify each feature. One way

to develop such artifacts is by means of *software components*. Given a particular configuration, the artifacts associated with the selected features are to be *composed* in order to obtain the desired product. In the context of Component-Based Software Engineering (CBSE) the typical unit of composition is the *software component* [2]. Ideally, all components are independent from each other. Nevertheless, in SPLs, each feature may be supported by *several components* which means that feature interactions may translate as dependencies and conflicts between components implementing them.

In this paper we propose an approach for feature-based architecture composition in component-based software product lines. To fill the gap between features and software components, we rely on the definition of aspect-like composition models that link every particular feature with several software components. Every model contains the information required for the composition including: (1) the locations modified by the feature, (2) the elements to be added and (3) the set of modifications to perform in order to add such elements. Their definition relies on Aspect Oriented Modeling (AOM), that consists in using the Aspect Oriented Programming (AOP) principles as part of the Model-Driven Engineering (MDE) development process [3]. We present an aspect metamodel to define the aspect models, and the mappings that enable such models to be composed by means of model transformations. Furthermore, our approach includes the combined analysis of the inter-feature constraints of the FD and the dependencies between the corresponding aspect models. We argue that such an analysis may significantly improve the composition process by allowing (1) the verification of the constraints explicitly defined in the FD, (2) the identification of implicit dependencies between the aspect models that are not defined in the FD, and (3) the derivation of a conflict-free composition strategy. The constraint analysis and composition are performed at the model level. Afterwards, the composed model is transformed into software components. We use Service-Component Architecture (SCA) [4] as target platform. SCA proposes a reconciliation between the Service Oriented Architecture (SOA) and CBSE, by defining a framework for describing the composition and the implementation of services using software components.

The main advantages of the proposed architecture composition approach as a whole are: (1) a clear separation of concerns achieved by defining independent aspect models, (2) the possibility to identify inconsistencies both in the FD and in the aspect models, (3) the definition of a feature-driven order to prevent conflicts in the process of architecture composition, and finally (4) the platform independence guaranteed by aspect models that are agnostic to the underlying technologies used for implementation.

The remainder of this paper is organized as follows. Section 2 presents a motivating example and a set of challenges for feature-based composition. Section 3 illustrates our approach in detail. In Section 4 we give some results of our experimentation and revisit the challenges identified in Section 2. Section 5 provides a related work discussion. In Section 6 we conclude the paper and anticipate future work.

2 Motivation and Challenges

In this section, we present an illustrative example and define a set of challenges for feature-based software composition.

2.1 Motivating Scenario

Let us consider the feature diagram of Figure 1. It defines a family of products with the essential functionality for an e-shopping scenario where a client connects to a server in order to find and buy items. The FODA terminology distinguishes three types of features: (1) *mandatory* features (dark circles) which are always selected (e.g. `Notification` and `Payment`), (2) *optional* features (white circles), which can be chosen or not (e.g. `Location`), and (3) *alternative* features (inverted arc), a special kind of optionality where the selection is realized among a limited set of alternatives, it can be non-exclusive (e.g. `CreditCard` and `Discount`) or exclusive (e.g. `SMS` and `Call`). In addition to that, the diagram introduces two types of constraints among features: *requires* and *excludes*. The requires constraint states that for a given feature to be selected, the required feature has to be selected before. The excludes constraint states that for a given feature to be selected, the excluded feature has to be deselected. In the feature diagram of Figure 1 there is one constraint indicating that location-filtered catalog needs one type of location to work.

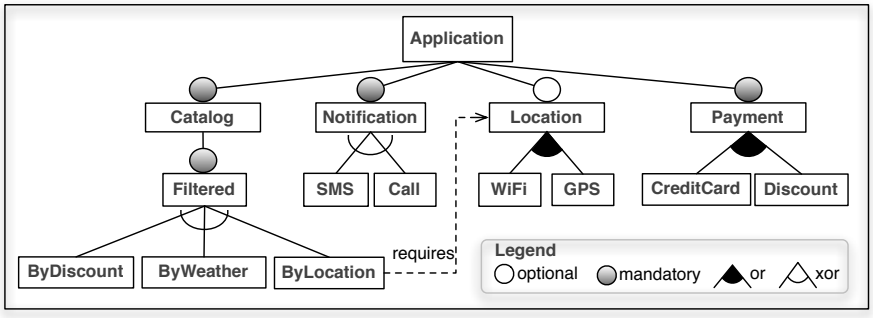

Fig. 1. A sample feature diagram.

2.2 Challenges

The main idea with a feature diagram like the one in Figure 1 is to enable software architects to derive their products based on (1) the selection of features, (2) the existence of dependencies between the selected features, and (3) the

mapping between the selected features and the supporting software artifacts. In order for software composition to fully benefit from the information contained in the feature diagram (variabilities, commonalities, and constraints), several challenges have to be faced:

1. **Ensure a clear separation of concerns:** Although feature diagrams enable the clean specification of software variability as a feature hierarchy, the mapping that holds between the features and the corresponding software artifacts may prove much more difficult to define. This is especially the case in the presence of *crosscutting* features, i.e., features that are materialized at multiple places in the final product. Possibly complex interactions between features on the one hand, and between artifacts on the other hand, further complicate the definition of the composable elements.
2. **Identify inconsistencies:** When composing multiple artifacts to form a software product, it is possible that two or more of those artifacts have conflicts regarding the elements where they are going to be composed and the requirements for the composition to take place. It may happen that implicit dependencies exist between artifacts that support independent features in the FD, and conversely. Such *inconsistencies* do not necessarily lead to composition problems but they have to be made explicit.
3. **Derive a suitable composition strategy:** This challenge corresponds to use the information at the feature and also at the artifact level to obtain the composition strategy. For example if two features have a dependency, like in the example `ByLocation` depends on any kind of `Location`, it is necessary to first compose the artifacts related to `Location` so that, `ByLocation` can reference parts of the `Location` artifacts. In other words, features have to be used to define partial orders in the composition of artifacts.
4. **Use multi-platform artifacts:** Finally, it is desirable that the artifacts that implement the features are platform-independent, this allows the SPL to have multiple targets and postpone the decision of a particular platform until later steps of the product derivation.

3 From Features to Aspect Composition

In order to obtain a software product from a set of features, we define a product derivation process with three main phases as illustrated in Figure 2: (1) *feature and aspect modeling* concerning to the language used to define both feature diagrams and aspect models, (2) *constraint analysis* dealing with the analysis of constraints at both the feature and aspect level, and finally (3) *model composition* that introduces a process to derive a single product using aspect model composition.

3.1 Feature and aspect modeling

In our approach, both the software variability and the composable software artifacts are represented as *models*. Here below, we present the two metamodels used to define feature and aspect models.

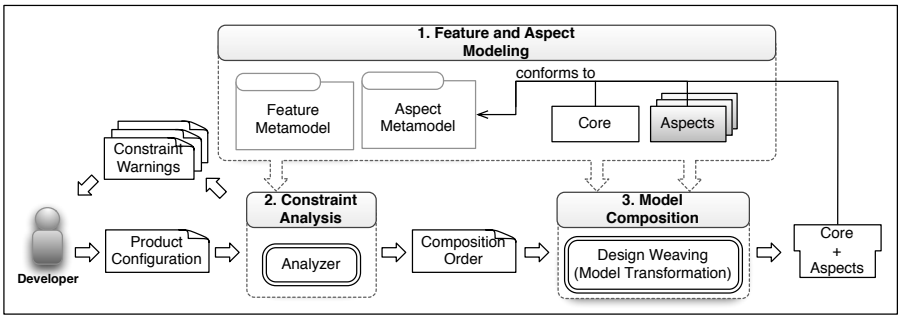

Fig. 2. Variability and Product Derivation.

Feature Metamodel Several works on feature modeling have proposed multiple extensions to the FDs initially introduced in [1]. In [5] Schobbens *et al.* survey different approaches to feature modeling and define an abstract syntax for feature diagrams that eliminate the ambiguity occurring in earlier proposals. They employ a mathematical notation to define the inter-feature relationships. A different approach to deal with ambiguity in FDs is by defining a metamodel like the one proposed by Pohl *et al.* [6]. This metamodel presents two main concepts: *variation points* and *variants*. A variation point is a representation of a variability subject, for example, the type of user interface that an application provides. A variant identifies a single option of a variation point. Using the same example, every single user interface that can be chosen for the application (e.g., rich, thin, web-based, mobile) is represented by a variant. The metamodel presented in [6] further specializes the relationships between variation points and variants, by classifying the types of relationships that may exist. They define dependencies (*optional* and *mandatory*) and constraints (*requires*, *excludes*). In this paper we define a feature metamodel inspired from the concepts that Pohl *et al.* have identified. In our metamodel, we define the same concepts and relationships using the Eclipse Modeling Framework (EMF) [7], but we change the way they are modeled, since EMF does not support the specialization or inheritance of relationships between two different meta-classes. Our feature metamodel is shown in Figure 3(Part a).

Aspect Metamodel The aspect metamodel (see part b of Figure 3) is essential in our approach, it allows us to link the three different methodologies (SPL, SCA, and AOSD) into one single model. First, the root of the metamodel is the `Aspect` which implements a `Variant` from the SPL. Second, an `Aspect` introduces the concepts needed to model a component and service based application (`Model`), and third, the `Aspect` also defines the two essential elements of any AOSD approach: the places where the weaving is realized (`Pointcut`), and the set of modifications to be performed (`Advice`).

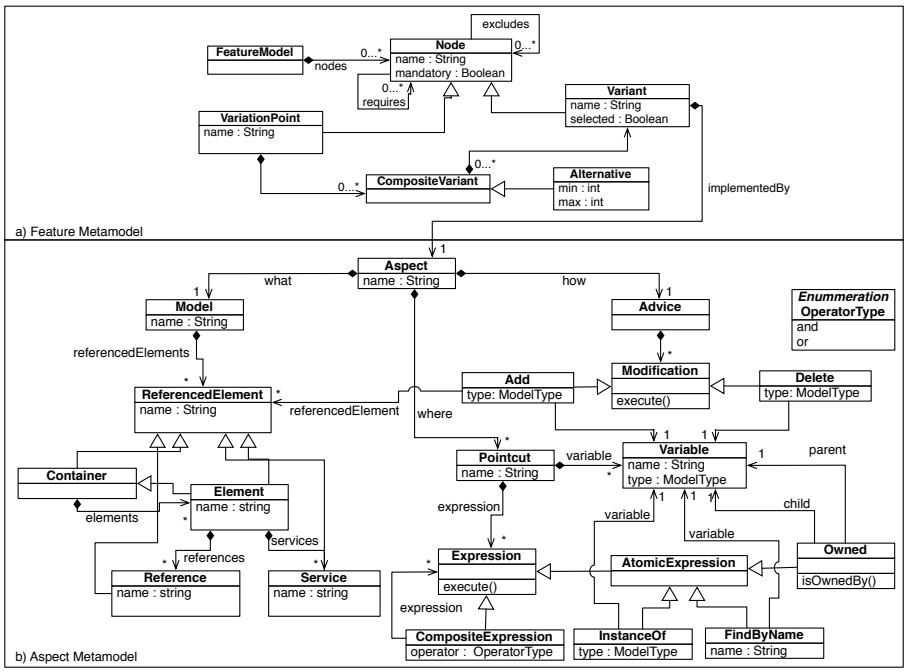

Fig. 3. Feature and aspect metamodels.

Modeling the elements to be weaved (`Model`): The `Model` part of the metamodel is used to define the *core*. It is inspired by the SCA, our target platform, and describes the structure of applications as a set of components (meta-class `Element`) that provides services (meta-class `Service`) and requires references (meta-class `Reference`). An element can contain other elements. This is expressed using the composite pattern of the meta-class `Container`. Additionally, to fully describe the architecture, the metamodel also introduces concepts like contracts, operations, objects, activities, connections, etc. Nevertheless, for space reasons, we show a reduced version of the metamodel making emphasis on the aspect information which is used in the constraint analysis. Every aspect uses the meta class `ReferencedElement` as an entry point to the `Model` part. As it can be noticed, every meta-class in the `Model` inherits from `ReferencedElement` which makes them accessible from the `Pointcut` and `Advice` definitions.

Modeling the place (`Pointcut`): We consider the `Pointcut` to be a query that returns all the model elements that have to be present in the model in order for an aspect to be weaved. A pointcut (meta-class `Pointcut`) is composed of expressions (meta-class `Expression`) and variables (meta-class `Variable`). An expression can be either composite (meta-class `Composite`) or atomic (meta-class

`Atomic`). A composite expression has an operator (meta-attribute `operator`) that defines the semantics of the composition (e.g., `and`, or). An atomic expression can be specialized in three different forms. `InstanceOf`, `FindByName` and `Owned`. `InstanceOf` is used to find an element using its type as a parameter. `FindByName` returns the elements whose name equals the `name` attribute of the expression. Finally the `Owned` expression looks for couples of elements where one of the elements (`parent`) owns the other (`child`). A variable represents a place where the elements obtained by executing an expression are stored.

Modeling the modifications (`Advice`): We consider the `Advice` to be a sequence of atomic modifications (meta-class `Modification`). There are two types of modifications supported: (1) add a new model element (meta-classes `Add`), which links an element of the model, represented as a `ReferencedElement`, and a `Variable` of the query, which represents the place where the element is going to be added, and (2) remove an existing model element (meta-classes `Remove`), which has a reference to the `Variable` representing the elements to be removed.

3.2 Constraint Analysis

The constraint analysis process takes place once the developer has configured a particular product. The feature selection is represented as a set of variants. Based on this selection, the constraint analysis aims at: (1) checking that the constraints defined in the FD are consistent with respect to corresponding inter-aspect dependencies, (2) identifying implicit composition constraints, and (3) deriving the most appropriate order of composition. This cross-model analysis utilizes the two parts: on the one hand (*left*) there are the features and their constraints, and on the other hand (*right*) there are the aspects with their own dependencies. The analysis goes in both ways: from features to aspects (*left to right*), and from aspects to features (*right to left*). Below, we specify both analyses based on the following notations:

- FD denotes the feature diagram of interest;
- $\mathcal{F} = \{F_1, F_2, \ldots, F_n\}$ denotes the set of features of FD;
- $\mathcal{P}$ denotes the set of valid products that can be derived from FD;
- $\mathcal{R} = \{(F_1, F_2) \in \mathcal{F} \times \mathcal{F} : F_1 \textit{ requires } F_2\}$ denotes the set of *requires* constraints of FD;
- $\mathcal{E} = \{(F_1, F_2) \in \mathcal{F} \times \mathcal{F} : F_1 \textit{ excludes } F_2\}$, denotes the set of *excludes* constraints of FD;
- A_F denotes the aspect model associated with a feature F;
- $\mathcal{A} = \bigcup_{F \in \mathcal{F}} A_F$ denotes the set of aspect models associated with the features of FD;
- A.`Model` denotes the `Model` part of an aspect $A \in \mathcal{A}$;
- A.`Pointcut` denotes the `Pointcut` of an aspect $A \in \mathcal{A}$;

Left to right analysis The *left to right* analysis, concerns the constraints (*requires* or *excludes*) that are *explicitly* specified in the FD. Given a valid feature

configuration, the analysis (1) checks that the related FD constraints actually translate as equivalent inter-aspect dependencies, (2) takes such dependencies as a basis to derive a correct weaving order, and (3) returns a warning for each FD constraint that has no "equivalent" at the aspect level.

- A "F_1 *requires* F_2" constraint in the FD usually implies that the pointcut of aspect A_{F_1} references some model element(s) introduced by aspect A_{F_2}. If it is the case, A_{F_2} must be woven *before* A_{F_1} when deriving the product.
- A "F_1 *excludes* F_2" constraint in the FD usually implies that the pointcuts of A_{F_1} and A_{F_2} references common model elements.

Algorithm 1 summarizes the *left to right* analysis process, which takes as inputs (1) the feature diagram FD, (2) the associated aspect models $\mathcal{A}$, and (3) a valid feature configuration p. Each *requires* constraint relative to p is analyzed (lines 2–8). If the constraint translates as a Pointcut-Model dependency, the weaving order is adapted accordingly (line 6). If such a dependency is not found, a corresponding warning is returned. The analysis of *excludes* constraints (lines 9–12) is similar, except that (1) it is based on Pointcut-Pointcut dependencies and (2) it does not impact the weaving order. Indeed, the feature configuration is supposed to be valid with respect to the explicit FD constraints.

Right to left analysis The second part of the analysis is intended to find *implicit* inter-feature constraints. Such dependencies are not specified in the FD, but hold between the corresponding aspects and, thus, may cause a conflict when realizing the composition. Similarly to the *left to right* analysis, two types of constraints are considered:

- A *requires* constraint indicates that an aspect pointcut refers to parts of the model of other aspect.
- An *excludes* constraint indicates that there are at least two pointcuts in distinct aspects with equivalent expressions. If such a situation occurs, then it is necessary to verify whether the corresponding advices are interfering with each other. Generally, aspects can be classified with respect to the interferences with each other in three categories: (1) *independent*, when their pointcuts and modifications do not affect other aspects, (2) *partially dependent*, when pointcuts may involve previously woven aspects but advices are independent, and (3) *totally dependent*, when pointcuts are dependent on previous aspects and advices may impact other aspects. In our case, it is the third category that may lead to composition conflicts. Consequently, aspects that exhibit such dependencies should not be weaved within the same product derivation. In order to determine whether the aspects are totally dependent, one must check if the modifications introduced by one aspect have a negative impact on the other. This is similar to *critical pair analysis* [8] in the domain of graph rewriting. Since there are only two types of modifications in our aspect metamodel: *add* and *delete*, the analyzer has to make sure that one aspect is not deleting an element referenced in the

Algorithm 1 Left to right analysis

Require: A feature diagram FD, the associated aspect models $\mathcal{A}$, a valid feature configuration $p = \{F_1, F_2, \ldots, F_k\} \in \mathcal{P}$
Ensure: A weaving order $\mathcal{O}$ and a set of warnings $\mathcal{W}$

$\mathcal{O} \leftarrow$ `toList`(p)
for all $(F_1, F_2) \in \mathcal{R}$ such that $F_1 \in p$ **do**
  if A_{F_1}.`Pointcut` $\cap$ A_{F_2}.`Model` $= \emptyset$ **then**
    $\mathcal{W} \leftarrow \mathcal{W} \cup \{F_1$ *does not require* F_2 *at the architectural level*$\}$
  else
    $\mathcal{O} \leftarrow$ `switchPositionIfNeeded`$(\mathcal{O}, F_2, F_1)$
  end if
end for
for all $(F_1, F_2) \in \mathcal{E}$ such that $F_1 \in p$ **do**
  if A_{F_1}.`Pointcut` $\cap$ A_{F_2}.`Pointcut` $= \emptyset$ **then**
    $\mathcal{W} \leftarrow \mathcal{W} \cup \{F_1$ *does not exclude* F_2 *at the architectural level*$\}$
  end if
end for

Algorithm 2 Right to left analysis

Require: A feature diagram FD, the associated aspect models $\mathcal{A}$, a valid feature configuration $p = \{F_1, F_2, \ldots, F_k\} \in \mathcal{P}$, an initial weaving order $\mathcal{O}$
Ensure: A flag `compositionAllowed`, a possibly adapted weaving order $\mathcal{O}$ and a set of warnings $\mathcal{W}$

`compositionAllowed` $\leftarrow$ *true*
for all $F1 \in p$ **do**
  for all $F_2 \in \mathcal{F}$ such that A_{F_1}.`Pointcut` $\cap$ A_{F_2}.`Model` $\neq \emptyset$ **do**
    if $(F_1, F_2) \notin \mathcal{R}$ **then**
      $\mathcal{W} \leftarrow \mathcal{W} \cup \{F_1$ implicitly *requires* F_2 *at the architectural level*$\}$
    end if
    if $F_2 \in p$ **then**
      $\mathcal{O} \leftarrow$ `switchPositionIfNeeded`$(\mathcal{O}, F_2, F_1)$
    else
      `compositionAllowed` $\leftarrow$ *false*
      $\mathcal{W} \leftarrow \mathcal{W} \cup \{F_1$ *implicitly requires a non-selected feature* $(F_2)\}$
    end if
  end for
end for
for all $F1 \in p$ **do**
  for all $F_2 \in \mathcal{F}$ such that A_{F_1}.`Pointcut` $\cap$ A_{F_2}.`Pointcut` $\neq \emptyset$ **do**
    if $(F_1, F_2) \notin \mathcal{E} \wedge$ `totallyDependent`(A_{F_1}, A_{F_2}) **then**
      $\mathcal{W} \leftarrow \mathcal{W} \cup \{F_1$ implicitly *excludes* F_2 *at the architectural level*$\}$
      if $F_2 \in p$ **then**
        `compositionAllowed` $\leftarrow$ *false*
        $\mathcal{W} \leftarrow \mathcal{W} \cup \{F_1$ *implicitly excludes a selected feature* $(F_2)\}$
      end if
    end if
  end for
end for

other aspect. If it does, the developer is warned about an implicit *excludes* constraint missing in the FD.

The *right to left* analysis is formalized in Algorithm 2. In case an implicit *requires* constraint is detected (lines 2–14), the behavior of the analyzer varies depending on whether the product configuration includes the required feature F_2 or not. If F_2 is selected, a warning is returned and the composition can be achieved according to an appropriate weaving order (line 8). If, in contrast, F_2 is not part of the configuration, then the composition is aborted (lines 10–11). Regarding the detection of implicit *excludes* constraints, the analyzer behaves in the other way around. In this case, indeed, the *presence* of excluded features F_2 in the configuration causes the composition to be aborted (lines 20–21), while their absence leads to a warning only (line 18).

Defining the composition order The composition order is derived from the analysis in both ways. To obtain it, the analysis tool traverses the list of features in the same order as they were selected, and, whenever a feature requires (implicitly or explicitly) other feature, it is moved in the list to the position right after the feature being required. This is done in both the *left to right* algorithm (line 6) and the *right to left* algorithm (line 8). This order guarantees that the pointcuts of features requiring other features are correctly executed during the composition.

3.3 Composition of Aspects

In general terms, the aspect composition consists of successive calls to a single generic model transformation (*weaver*). This transformation takes as inputs the *core* model M and an aspect A to be weaved, and returns a single model representing the composition of the core and the aspect. The transformation itself relies on the metamodel of Figure 3 (Part b). It consists in iterating over the set of modifications specified in the *Advice* of A in order to execute each of them.

The places where each modification takes place are defined by the associated `Pointcut`. The execution of this pointcut on the core model iterates over its `Expressions`, which can be either atomic or composite. Atomic expressions correspond to `FindByName`, `InstanceOf` and `Owned`. Each atomic expression returns the collection of *core* model elements that match their conditions. A composite expression is evaluated by accumulating and combining the result of each atomic expression. The way the resulting elements are combined depends on the composite operator. The `AND` operator is interpreted as the *intersection* of the model elements, whereas the `OR` operator translates as their *union*.

At the end of the pointcut execution, all the places impacted by the aspect have been identified. Then the modifications specified by the aspect can be applied. In the case of an `Add` modification, the elements of the aspect are added to the *core* model. Applying a `Delete` consists in removing the elements found in the pointcut from the *core* model.

The transformation finishes when all modifications specified in the advice have been performed. The global weaving process repeats until all the aspects corresponding to the variants selected in the feature configuration have been composed with the core model.

4 Experimentation and Discussion

In order to test the constraint analysis introduced in the previous section, we have implemented the sample SPL introduced with the FD in Section 2 and applied our constraint analysis. There are in total 9 variants (`ByDiscount`, `ByWeather`, `ByLocation`, `SMS`, `Call`, `Wifi`, `GPS`, `CreditCard`, and `Discount`) which are realized with individual aspect models. The total number of valid products that are derivable from such diagram is 66, that is 72 in total minus 6 that do not respect the requires constraint. In Table 4 we have selected a subset of 10 products to illustrate the result of the analysis. For each product we present the list of selected variants (v1-v9), the results of the *left2right* (l2r) and *rigth2left* (r2l) algorithms, the order of composition (Result), and the execution time (Time) in milliseconds.

As it can be seen from the results, the analysis for each product takes slightly short times for this small FD. Nevertheless, the more variants there exist, the more aspects to verify for each product with consequences in performance, but such an overload is related to the nature of the product family itself. Additionally, since this process is executed during the design phase, time and performance are less critical than correctness and conflict-free composition.

Table 1. Constraint Analysis Results.

Product	**v1**	**v2**	**v3**	**v4**	**v5**	**v6**	**v7**	**v8**	**v9**	***L2R***	***R2L***	**Result**	**Time(ms)**
1	✓	–	–	✓	–	–	–	✓	–	–	–	{v1,v4,v8}	242
2	–	✓	–	–	✓	–	–	–	✓	–	HI(v9,v4)	Not allowed	229
3	–	–	✓	✓	–	✓	–	✓	–	*Order*	–	{v6,v3,v4,v8}	240
4	–	–	✓	–	✓	✓	–	✓	–	*Order*	–	{v6,v3,v5,v8}	236
5	–	–	✓	–	✓	–	✓	–	✓	*Order*	HI(v9,v4)	Not allowed	231
6	–	✓	–	✓	–	✓	–	✓	–	–	–	{v2,v4,v6,v8}	234
7	✓	–	–	–	✓	–	✓	✓	–	–	–	{v1,v5,v7,v8}	242
8	–	✓	–	✓	–	–	✓	✓	–	–	–	{v2,v4,v7,v8}	270
9	–	–	✓	–	✓	–	–	✓	–	*Order*	HI(v9,v4)	Not allowed	255
10	✓	–	–	✓	–	–	–	✓	✓	–	HI(v9,v4)	{v1,v4,v9,v8}	244

Regarding the results of the analysis, we notice that the *left to right* algorithm modifies the *order* of composition of the products 3,4,5, and 9. On the other side, the *requires* constraint between the variant ByLocation and the variation point Location has an equivalent dependency in the aspect level. However, as

previously stated, even if there was no equivalent dependencies, the constraint does not necessarily represent an error since it may come from a business rule.

On the other side, the *right to left* analysis shows that the aspect for the variant 9 (Discount) has a dependency (presented in the table as HI for *Hidden Includes*), with the aspect realizing the variant 4 (SMS). As a result, products 2,5 and 9 are not allowed for composition. In the case of product 10, the analysis shows the same dependency, but the composition is allowed since the variant 4 (SMS) is selected. Additionally, the order does not need to be changed since the variant 4 (SMS) is already placed before the variant 9 (Discount).

Regarding the implementation, we have used the tools provided by EMF. There are four metamodels in total: the feature and aspect metamodels introduced in Section 3, and additionally, there are two metamodels for SCA and Java respectively. The constraint analysis algorithms as well as the model transformations are written in Java and use the EMF API to import and manipulate the models. We have made this choice over other model platforms for two main reasons: first, we wanted to let the aspect developers to decide how the aspect has to be composed. Our weaver is generic and allows aspects to define any combination of advices and pointcuts. This gives aspects great expressivity and at the same time, we guarantee that every aspect, modeled with the metamodel presented in Figure 3, can be processed by the weaver. And second, by using our own definition and semantics for the modification operations (Add and Remove), we are able to generate equivalent reconfiguration scripts that can be executed at runtime. With this property we aim at defining a dynamic product derivation using the same aspect models.

4.1 Discussion

Let us now revisit the feature-based architecture composition challenges identified in Section 2 for discussing the tool-supported approach proposed in this paper. Regarding challenge 1, our modeling approach contributes to a clear separation of concerns at three levels: variability expression, architecture definition, and feature-architecture mapping specification. We benefit from the complementary capabilities of Feature Modeling, Component-Based/Service-Oriented Architecture and Aspect-Oriented Modeling. The constraint analysis algorithms allows the detection of inconsistencies (challenge 2) in the FD as well as in the aspect models. This prevents the composition from taking place unless all the constraints are respected. Additionally, the algorithms take explicit and implicit features interactions as a basis to derive a conflict-free composition strategy (challenge 3) that ensures that aspects are weaved in the appropriate order for any given configuration. Finally, our aspects are platform-independent models (challenge 4). In our case, model transformations have been implemented towards a particular platform (SCA and Java) to enable the SPL to deal with dynamic product derivation as explained in [9]. Nevertheless, such aspect models can be transformed towards different component-based platforms, in which case, the analysis and composition processes remain valid.

5 Related Work

This section discusses the complementarity of our approach with respect to previous work on feature-based software composition, aspectual feature modeling and aspect-oriented model composition.

Feature-based software composition In [10] van der Storm presents a generic approach to feature-based software composition, with a particular focus on the feature configuration phase. Feature descriptions are mapped to related software artifacts through a formal model. This mapping indicates which artifact(s) should be included in the product if a feature is selected. A scalable configuration technique, based on binary decision diagrams (BDDs) [11], is developed. The BDDs, derived from the feature interactions specified in the feature model, aim to lead to valid configurations only. Several possible methods are identified for the composition process itself, each supporting a different level of granularity. In contrast, we assume that a valid product configuration is available, and we contribute to the subsequent feature composition process.

Voelter and Groher [12] present an approach based on the combination of aspect-oriented and model-driven software development. This approach supports the explicit separation and modeling of variability in feature models. In the implementation of this approach, an AOP framework enables product derivation to be performed using a weaving process described in a workflow. Kuhlemman *et al.* [13] presents a tool-supported approach to support safe composition of *non-monotonic* features, i.e., features that *add* and *remove* code. In particular, the authors verify that all valid combinations of features can be composed without errors. Considering each feature implementation as an increment in program functionality, software composition is seen by the authors as the application of successive *feature transformations* that add features to a program (by adding and/or removing code). The authors use SAT technologies to check *configurable sequences* of feature transformations. They show that automated support is indispensable due to the rapidly growing complexity of the analysis.

Our approach also enables the automated composition of (non-monotonic) features, but it considers the architecture level rather than the code level. Furthermore, in contrast with Kuhleman *et al.*, we do not assume that the feature composition order is encoded in the feature model by reading from right to left. Our analysis technique aims at deriving an adequate composition order based on both explicit and implicit feature interactions.

Lee *et al.* [14] addresses the challenge of software composition in the presence of feature dependencies. They suggest the use of *aspect-oriented* implementation patterns for such dependencies. This approach allows a clear separation of feature dependencies from feature implementations, thereby increasing the reusability of the latter. The authors mainly focus on *dynamic* feature interactions as those identified in [15], whereas we consider *structural* dependencies between features. Czarnecki *et al.* [16] present an automated procedure for verifying that a given feature configuration will lead to a *correct* product model. The notion of correctness they consider is *well-formedness*: they verify that the resulting product model conforms to the meta-model of the target modelling language. In con-

strast, we aim to check that the configured product can be composed. When possible, we derive a conflict-free composition strategy allowing all the selected features to be correctly supported. The analysis of implicit feature dependencies is essential in this context. For instance, failing to identify an implicit *requires* constraint may lead to an incomplete, yet well-formed, product model.

Aspectual feature modeling Griss [17] presents a conceptual framework for feature-based and aspect-oriented product line engineering. The key idea is to use aspects for implementing the features identified as common and variable in a product line. Lee *et al.* [18] go a step further by proposing a set of detailed guidelines on how feature-oriented programming and aspect-oriented programming can be combined in order to enhance the reusability, adaptability and configurability of software product line artifacts. They aim at addressing the so-called *invasive change* problem. This problem is due to the fact that the code implementing a particular feature may be scattered across multiple components, and consequently adding or removing a feature may have an impact on several source code locations. Our work also aims at addressing this problem by considering both inter-feature dependencies and inter-aspect dependencies.

More recently, Apel *et al.* [19] introduce the notion of *aspectual feature module* (AFM), which constitutes a proposal of the symbiosis of Feature-Oriented Programming and Aspect-Oriented Programming. An AFM encapsulates the roles of collaborating classes and aspects that together contribute to implementing a feature. According to this view, a feature implementation regroups a collection of artifacts among which classes, class refinements and aspects. The use of aspects in an AFM brings the benefit from AOP's modularization capabilities. In our approach, we also from aspect modularization, but in our case we do not mix aspects and classes to implement a feature. We aim at defining independent aspects that are woven with a core. Since our aspects are self-contained, they include a model part, which defines the components and services that are latter transformed into configuration files and classes.

Aspect-oriented model composition Zhang *et al.* [20] show that the explicit specification of aspect precedence at the modeling level allows to mitigate the problem of aspect interference in AOM. The precedence declarations enable the composition mechanism to automatically derive an appropriate weaving order. Our approach relies on this principle and also takes into account the mutual dependency between feature interactions and related aspect precedence.

Morin *et al.* [21] consider the introduction of variability at a higher level of abstraction. They present a generic approach to weaving variability in metamodels, by means of a reusable variability aspect. This aspect allows the description of the variability concepts and the relationships between them, in a metamodel-independent manner. Such an aspect can then be woven using standard AOM techniques in order to include variability in a given domain-specific metamodel. The authors then show how to compute a feature diagram from an instance model with variability. In contrast, our approach takes feature diagrams as input for aspect-based architecture composition.

6 Conclusion

This paper presented a comprehensive approach to feature-driven composition of software architectures. This approach allows the automated derivation of product architectures from feature configurations, by combining MDE and AOM techniques. The composition process is realized through transformation-based model weaving and is guided by the explicit and implicit dependencies that exist between the selected features. Our proposal relies on a clear separation of concerns enabled by the underlying variability and aspect metamodels. Our method allows to identify implicit dependencies and conflicts between features, and takes such feature interactions as a basis to derive an appropriate architecture composition strategy. The overall approach is implemented in a generic SPL framework that enables the composition and deployment of both component-based and service-oriented architectures on various platforms. In the near future, we intend to consolidate the promising results obtained so far, following two main directions. First, we want to explore the reusability of our approach in the context of *dynamic* feature (de)selection. We believe that it could be extended to support the derivation of context-aware, self-adaptive systems. Second, we intend to evaluate the application of our feature-based composition techniques to larger software systems. We already identified FraSCAti [22], a configurable SCA platform, as a good candidate for such an experiment.

Acknowledgments The CAPPUCINO project is funded by the Conseil Régional Nord-Pas-de- Calais, Oseo/ANVAR, and the Fonds Unique Interministériel. This work was also supported by Ministry of Higher Education and Research, *Nord-Pas de Calais* Regional Council and FEDER through the *Contrat de Projets Etat Region* (CPER) 2007-2013. This research was carried out during the tenure of an ERCIM "*Alain Bensoussan*" Fellowship.

References

1. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute (November 1990)
2. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. 2nd edn. Addison-Wesley Professional (2002)
3. Jézéquel, J.M.: Model driven design and aspect weaving. Software and System Modeling **7**(2) (2008) 209–218
4. Open SOA: Service component architecture specifications (November 2007) www.osoa.org/display/Main/Service+Component+Architecture+Home.
5. Schobbens, P.Y., Heymans, P., Trigaux, J.C.: Feature diagrams: A survey and a formal semantics. In: 14th Int. Requirements Engineering Conference (RE'06). (2006) 136–145
6. Pohl, K., Böckle, G., Linden, F.J.v.d.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag (2005)
7. The Eclipse Foundation: Eclipse Modeling Framework Project (EMF) (2010) http://www.eclipse.org/modeling/emf/.

8. Plump, D.: Hypergraph rewriting: critical pairs and undecidability of confluence. (1993) 201–213
9. Parra, C., Blanc, X., Duchien, L.: Context Awareness for Dynamic Service-Oriented Product Lines. In: Proceedings of the 13th International Software Product Line Conference (SPLC'09). (2009) 131–140
10. der Storm, T.V.: Generic feature-based software composition. In: 6th Int. Symposium on Software Composition (SC'07). Volume 4829 of LNCS., Springer (2007) 66–80
11. Bryant, R.E.: Symbolic Boolean manipulation with ordered binary-decision diagrams. ACM Computing Surveys **24**(3) (1992) 293–318
12. Voelter, M., Groher, I.: Product line implementation using aspect-oriented and model-driven so ftware development. In: 11th Int. Software Product Line Conference (SPLC'07), IEEE CS (2007) 233–242
13. Kuhlemann, M., Batory, D., Kästner, C.: Safe composition of non-monotonic features. In: 8th Int. Conference on Generative Programming and Component Engineering (GPCE'09), ACM (2009) 177–186
14. Lee, K., Botterweck, G., Thiel, S.: Aspectual separation of feature dependencies for flexible feature composition. In: 33rd Annual IEEE Int. Computer Software and Applications Conference, IEEE CS (2009) 45–52
15. Lee, K., Kang, K.C.: Feature dependency analysis for product line component design. In: 8th Int. Conference on Software Reuse: Methods, Techniques and Tools (ICSR'04). Volume 3107 of LNCS., Springer (2004) 69–85
16. Czarnecki, K., Pietroszek, K.: Verifying feature-based model templates against well-formedness ocl constraints. In: 5th Int. Conference on Generative Programming and Component Engineering (GPCE'06), ACM (2006) 211–220
17. Griss, M.L.: Implementing product-line features by composing aspects. In: 1st Conference on Software Product lines : experience and research directions (SPLC'00), Kluwer Academic Publishers (2000) 271–288
18. Lee, K., Kang, K.C., Kim, M., Park, S.: Combining feature-oriented analysis and aspect-oriented programming for product line asset development. In: 10th Int. Software Product Line Conference (SPLC'06), IEEE CS (2006) 103–112
19. Apel, S., Leich, T., Saake, G.: Aspectual feature modules. IEEE Transactions on Software Engineering (TSE) **34**(2) (2008) 162–180
20. Zhang, J., Cottenier, T., van den Berg, A., Gray, J.: Aspect composition in the motorola aspect-oriented modelling weaver. Journal of Object Technology **6** (August 2007) 89–108 Special issue on Aspect-Oriented Modelling.
21. Morin, B., Perrouin, G., Lahire, P., Barais, O., Vanwormhoudt, G., Jézéquel, J.M.: Weaving variability into domain metamodels. In: 12th Int. Conference on Model Driven Engineering Language and Systems (MoDELS'09). Volume 5795 of LNCS., Springer (2009) 690–705
22. Seinturier, L., Merle, P., Fournier, D., Dolet, N., Schiavoni, V., Stefani, J.B.: Reconfigurable sca applications with the frascati platform. In: 6th IEEE International Conference on Service Computing (SCC'09). (sep 2009) 268–275