Unifying design and runtime software adaptation using aspect models

Carlos Parra, Xavier Blanc, Anthony Cleve, Laurence Duchien

Unifying Design and Runtime Software Adaptation Using Aspect Models

Carlos Parra[a]

Xavier Blanc[b]

Anthony Cleve[a]

Laurence Duchien[a]

[a] *INRIA Lille-Nord Europe, LIFL CNRS UMR 8022, Université de Lille 1, Lille, France*
[b] *Université Bordeaux 1, Bordeaux, France*

Abstract

Software systems are seen more and more as evolutive systems. At the design phase, software is constantly in adaptation by the building process itself, and at runtime, it can be adapted in response to changing conditions in the executing environment such as location or resources. Adaptation is generally difficult to specify because of its cross-cutting impact on software. This article introduces an approach to unify adaptation at design and at runtime based on Aspect Oriented Modeling. Our approach proposes a unified aspect metamodel and a platform that realizes two different weaving processes to achieve design and runtime adaptations. This approach is used in a Dynamic Software Product Line which derives products that can be configured at design time and adapted at runtime in order to dynamically fit new requirements or resource changes. Such products are implemented using the Service Component Architecture and Java. Finally, we illustrate the use of our approach based on an adaptive e-shopping scenario. The main advantages of this unification are: a clear separation of concerns, the self-contained aspect model that can be weaved during the design and execution, and the platform independence guaranteed by two different types of weaving.

Key words: Aspect Oriented Modeling, Software Product Lines

1. Introduction

Forty years after the first conference on Software Engineering [1] and almost twenty years after the IEEE Computer Society has standardized this discipline, Soft-

Email addresses: carlos.parra@inria.fr (Carlos Parra), xavier.blanc@labri.fr (Xavier Blanc), anthony.cleve@inria.fr (Anthony Cleve), laurence.duchien@inria.fr (Laurence Duchien)

ware Engineering is still struggling to produce large software systems [2, 3]. As it was already pointed out by the NATO conference and formalized by M.M. Lehman [4], two main reasons for this situation are (1) the fact that software systems have an intrinsic increasing complexity, but also (2) that such systems are living entities with a continuous infinite life cycle.

Software Product Line (SPL) engineering was mainly introduced to deal with the intrinsic increasing complexity [5]. SPL focuses on variability management and aims at deriving different products from a same product family. In SPL, feature diagrams express the variability of a same product family by defining its variants and its variation points [6, 7]. Based on a feature diagram, one of the most difficult SPL challenges is *Product Derivation* (PD), which is the complete process of building a particular product belonging to the product family [8]. The PD defines how assets are selected according to a given feature configuration, and specifies how those assets are composed in order to build the desired product.

Aspect-Oriented Software Development (AOSD) and Model-Driven Engineering (MDE) can be used to face the issue of continuous infinite life cycle. AOSD and MDE follow the well-known separation of concerns principle, which has been proven to provide many benefits, including reduced complexity, improved reusability, and easier evolution [9]. Thanks to AOSD, software systems can be modularized using orthogonal aspects that are woven at the production time [10]. MDE deals with levels of abstraction and considers any software artifact produced at any step of the development process as a valuable asset by itself to be reused across different systems and implementation platforms[11].

The Aspect Oriented Modeling (AOM) initiative introduces the AOSD principles in the MDE development process, particularly in the composition and transformation phases [12] [13]. AOM is currently being used within more and more SPL approaches [14, 15, 16, 17] in order to compose assets selected from a feature diagram. Those approaches however only contribute to the design phases of the software life cycle. The system features and their corresponding assets are modeled using AOM techniques. The product derivation process is supported by automatic model compositions and transformations. As a consequence, they are used to build software systems that, once deployed, cannot be easily evolved.

New concerns like ubiquitous computing, mobile and autonomous systems emphasize on dynamic adaptation, which is usually achieved by performing structural or behavioral changes at runtime. Some SPL approaches contribute to the runtime phase [18, 19]. They are based on rules specifying the contextual changes that trigger the dynamic adaptation of a software system. Although some of those approaches make use of aspects to specify and realize dynamic adaptation, they define new mechanisms that differ from those involved in existing SPL approaches focusing on the design phase. As a consequence, there exist no unified AOM-based SPL approach that covers the complete life cycle, from design to runtime.

In this article, we propose such a unified approach, the goal of which is to support the complete software life cycle: from feature selection and initial product derivation, to runtime adaptation in response to changes of the execution environment. Our approach considers that the creation of the initial product is performed through design adaptations. Once created and deployed, the product can then be subject to dynamic

adaptations. Our approach unifies design and runtime adaptations by representing both categories of adaptation as executable aspect models.

The approach is built on two main entities. First, an unified aspect metamodel is used to support the definition of both design and runtime adaptations. Second, a platform transparently realizes the adaptations expressed by means of the unified metamodel. This platform is composed of two independent mappings that support weaving at design or at runtime. The weaving at design time links the elements of the aspect in the design model. The result of such a weaving can then be further transformed into platform-specific models and code. The weaving at runtime dynamically links the elements of the aspect to the running system.

The main advantages of our approach are: (1) a clear separation of concerns achieved by defining aspect models, (2) the possibility of weaving such aspects at different phases of the application lifecycle (design and runtime), and (3) the platform independence guaranteed by aspects that are agnostic to the underlying technologies used for each weaving. The proposed approach has been implemented and validated in a Dynamic SPL called CAPucine [20].

The remainder of this article is structured as follows. Section 2 presents the motivation behind the definition and implementation of aspect models at design and at runtime. Section 3 describes our approach in detail. In Section 4 we illustrate with an e-commerce example how this approach can be used in the context of a dynamic SPL. The advantages and disadvantages of our approach are discussed in Section 5. Section 6 presents the related work. In Section 7, we draw our conclusions and anticipate future work.

2. Motivation

In all software processes, existing software can be adapted either at the design phase or at the runtime phase. For each of those two phases, dedicated technologies are used to specify and realize the adaptations. For instance, design languages provide adaptation mechanisms such as inheritance or composition; and runtime platforms that support dynamic adaptation provide API's to dynamically change connections between running components.

Design adaptations are often considered to be of completely different nature than runtime adaptations. Design adaptations are motivated by design goals whereas runtime adaptations are motivated by changes of the software's environment. Moreover, design adaptations are considered as permanent adaptations that cannot be rolled back whereas runtime adaptations are considered as impermanent.

However, whatever the technology and whatever the phase, a software adaptation is always initiated by a particular motivation and is always realized through modifications of some software artifacts. Therefore, from a specification point of view, design and runtime adaptation are not so different. We then argue that a single unified language should be provided to specify both of them. Based on this language, a platform should be realized to execute the weaving of design and runtime aspects transparently.

Having only one unified language to specify design and runtime adaptations offers several advantages. First, it formalizes similarities and differences that exist between

the two kinds of adaptation. Second, it may serve as a basis to transform design adaptations into runtime ones and vice versa. Transforming design adaptations into runtime one allows one to delay the realization of some design adaptations to the runtime phase. Transforming runtime adaptation into design one prevents the realization of adaptation mechanisms that have been defined regarding specific environment's changes that may never arise at runtime. Third, unifying the specification of modifications done by both aspects is the first step to compute analysis between aspects, such as dependency analysis for example.

Having a platform that executes the weaving of design and runtime aspects transparently offers several advantages. First, it supports the whole life cycle from the initial creation of the product (driven by feature selection) to its dynamic adaptation (driven by changes of its environment). Second, it then maintains traceability links between the motivations (features selection or changes of the product's environment) and the adaptations of the software artifact. Third, it can be used as a flexibility cursor to reach a tradeoff between a development cycle that will be fully design oriented (without any runtime adaptation), and a development cycle that will be fully runtime oriented (without any feature selection).

In the next section, we present our approach that provides such a unified language and its corresponding platform.

3. Approach

Our approach allows software systems to be adapted at design time and runtime. We consider an adaptation as the addition or removal of *optional* parts to a fixed base *core*. Our approach is based on SPL principles. Feature diagrams (FD) are used to model *commonalities* (i.e. core), and *variabilities* (i.e. optional), among the members of the same family of software products. An FD typically consists of a hierarchy of *features*, which may be *mandatory* or *optional*. To realize each feature and fill the gap between features and implementation, we rely on a metamodel for defining aspect-like composition models. Such models link every particular feature with an optional set of components that may be part of a product. 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 those elements.

The platform weaves the models in two different ways for both *design* time, and *runtime*. This duality of the platform is due to the motivations behind design and runtime weaving. While in design, the platform takes into account the decisions from the developers that want to derive a product, at runtime, the weaving is triggered with context events of changes in the environment, without human intervention. A product is then adapted via a design choice through a feature selection, or a runtime choice in reaction to context events.

Our approach is based on three main phases for building and/or adapting a product: aspect modeling, design weaving, and runtime weaving. Figure 1 illustrates such phases.

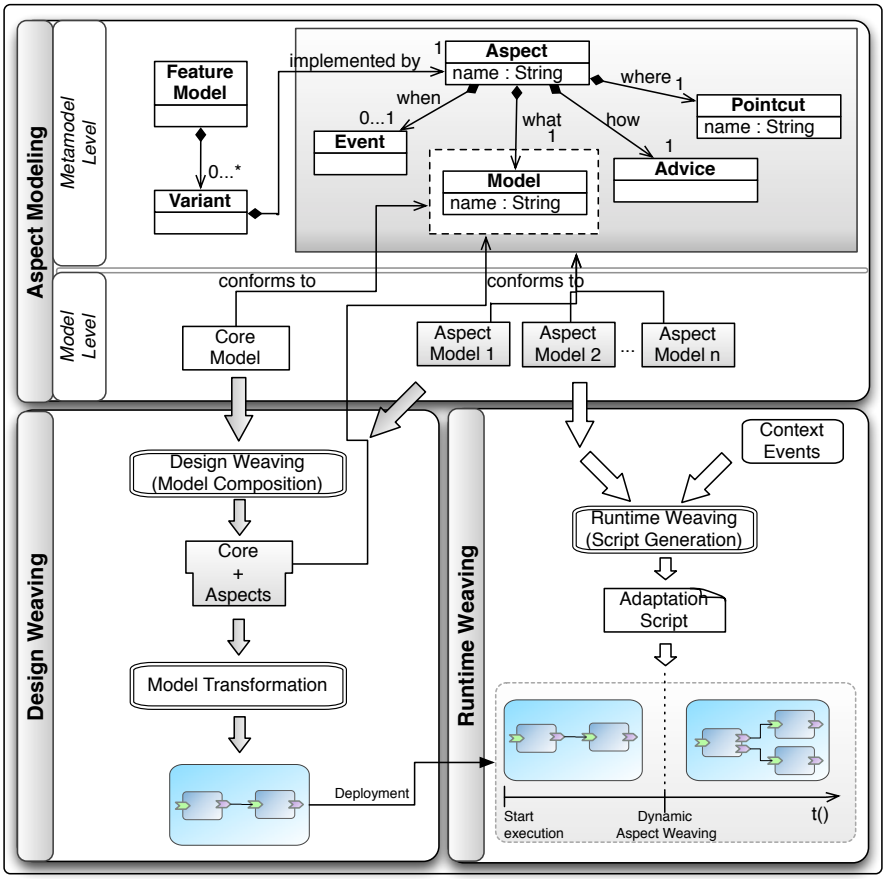

Figure 1: Different development phases using aspect models.

- **Aspect Modeling** relates to the definition, using a high level language, of different composable aspects to describe any product. Typically, a product is described by a *core* model, and a set of independent aspect models that are woven to the *core*. Our approach is symmetric i.e. both the core and the aspects are modeled using the same metamodel and are treated in the same way.
- **Design Weaving** represents the design adaptations. It covers the aspect model weaving of the core with several aspects. This process is triggered by the developer of the application who selects the aspects to be woven with the core model. The design weaving ends with a partially generated product, that after manual completion can be deployed and executed.
- **Runtime Weaving** represents the runtime adaptations. It covers the aspect weaving at runtime. Our approach is based on the execution environment (by means

of contextual events for example), to decide when an aspect can be woven at runtime. As we target component-based and service-oriented software, the changes that we propose to apply at runtime refer to architectural modifications (e.g. adding a new component, modifying existing bindings, removing services).

The following sections introduce the aspect metamodel, and describe in detail the process to transform aspect models into elements to perform design and runtime weaving.

3.1. Aspect Metamodeling

The aspect metamodel (see Figure 2) is formed by four parts: the elements to be woven (`Model`), the places where the weaving is realized (`Pointcut`), the modifications performed by the aspect (`Advice`), and optionally, the moment at runtime when the aspect gets to be woven (`Event`).

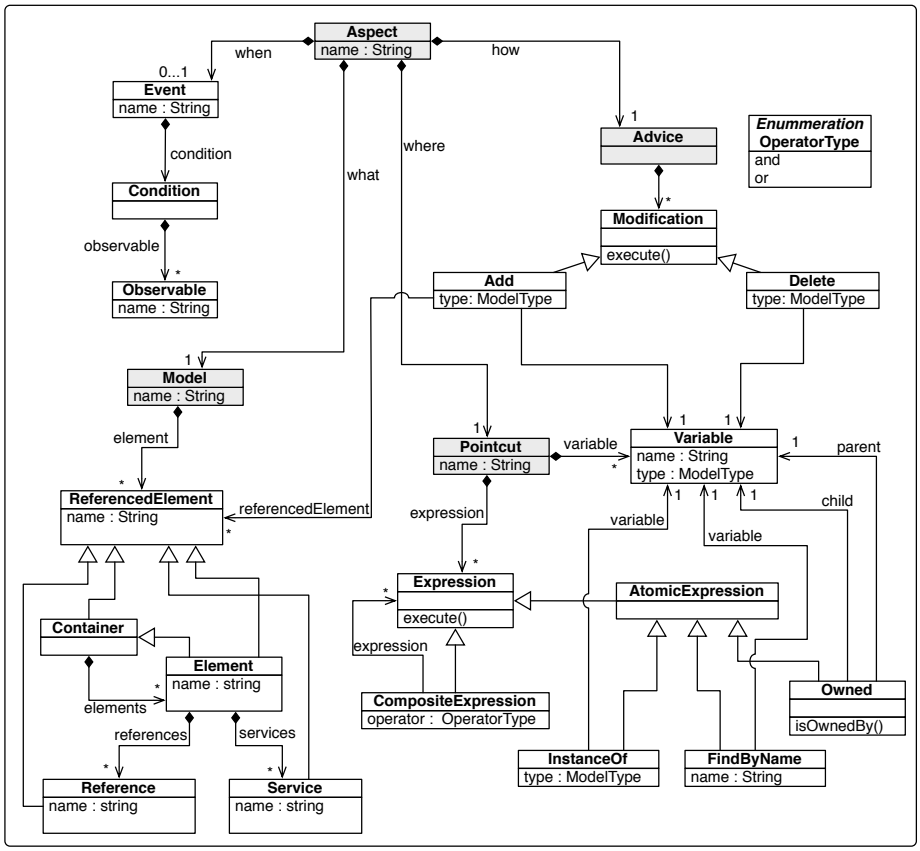

Figure 2: Aspect Metamodel.

- **Modeling the elements to be woven (*what*):** the `Model` part of the metamodel is used to define the *core* model. It describes the software structure as a set of elements (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`. To connect the `Model` with the other parts of the aspect, there is a meta-class called `ReferencedElement`. This meta-class is specialized by all the meta-classes in the `Model`, and works as an entry point for the `Pointcut` and `Advice`.

- **Modeling the place (*where*):** we consider the *Pointcut* as a query that returns all the model elements that have to be present in the model in order for an aspect to be woven. 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 may contain nested expressions. To aggregate the results it uses 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 is equal to 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 as a result of the execution of an expression are stored.

- **Modeling the modifications (*how*):** we consider the *Advice* to be a sequence of atomic modifications (meta-class `Modification`). The following atomic modifications are supported by our metamodel:

  - The addition of a new model element (meta-classes `Add`). To add an element, each `Add` statement 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.
  - The removal of an existing model element (meta-classes `Delete`). To remove an element, each `Remove` statement references a `Variable` which represents the elements that are going to be removed.

- **Modeling the time (*when*):** Modeling the time is only relevant for aspects that can be woven at runtime. This is why an aspect may or may not contain an event definition (multiplicity is 0..1). To model the time we use *context events*. By context we mean every piece of information that may affect an application. Examples of such information may vary from availability of resources or services to information like temperature, location, or even hardware restrictions like memory. Consequently, a context event is a change in context information. To model this event, we define the notion of *observable*. An observable is an abstraction of a single piece of information referring to context. It consists of a single value that

can be easily evaluated to decide whether to weave an aspect dynamically or not. Figure 2 illustrates this part of the aspect metamodel, an `Observable` appears as part of a `Condition` that belongs to the aspect's `Event`.The evaluation of the condition indicates wether the aspect to which it belongs has to be woven at runtime, and consequently triggering an adaptation of the running application.

3.2. Design and Runtime Weaving

Once the aspect metamodel has been defined, we need to define a platform that is able to use the aspect models. The platform is in charge of transforming models and generating code for design weaving and creating adaptation scripts for runtime weaving. Given the nature and the challenges of each type of weaving, the process and technologies used to transform the models are different in each case. Both are explained in detail in the following sections.

3.2.1. From aspect models to design weaving

In general terms, the design weaving 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 woven, and returns a single model representing the composition of the core and the aspect. The transformation itself relies on the metamodel of Figure 2. It consists in iterating over the set of modifications specified in the *Advice* of A in order to execute each one of them.

The places where each modification takes place is defined by the associated `Pointcut`. The execution of this pointcut on the core model iterates over its expressions, which can be either atomic (`FindByName`, `InstanceOf` and `Owned`) or composite. Each atomic expression returns the collection of core model elements that match the filter 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. Two possible modifications are allowed:

- *Add:* to add an element, the meta-class `Add` has a relationship with the `ReferencedElement` being added and the `Variable` where it will be added. Nevertheless, having this relationship allows for any referenced element to be woven at any place in the *core* model. To prevent incompatible combinations, like for example trying to add a `Container` inside an `Attribute`, we have defined a scope for these combinations. There is a set of allowed pairs of types, where the first type corresponds to the referenced element being woven and the second type corresponds to the variable where the element is going to be added. For each allowed pair, we perform the adequate operations to add the element. Allowed pairs vary from coarse grained operations (i.e. adding a new `Element` inside an existing `Container`) to finer grained ones (i.e. adding a new `Operation` inside a `Service Contract`). For the incompatible pairs, no weaving is performed.

- *Delete:* deletes the elements at the places described by the `Variable`. The deletion of an element triggers the destruction of it's inner elements (for `Element` or `Container` elements).

The transformation finishes when all modifications specified in the advice have been performed. The weaving process repeats until all the aspects whose variants are selected have been composed with the core model. The resulting model is no longer the core but a complete representation of an application that also includes the right constructs for the concerns defined in every aspect woven.

However, this model is still platform and technology independent. Following a classic MDE approach, the composed model is used again as an input to two different transformations to obtain platform and implementation models. A third transformation verifies the consistency between the two models obtained. Finally, the platform model is used to generate a composite file with the architecture of the product, and the implementation model is used to generate Java implementations for every component in the model. Figure 3 illustrates this process.

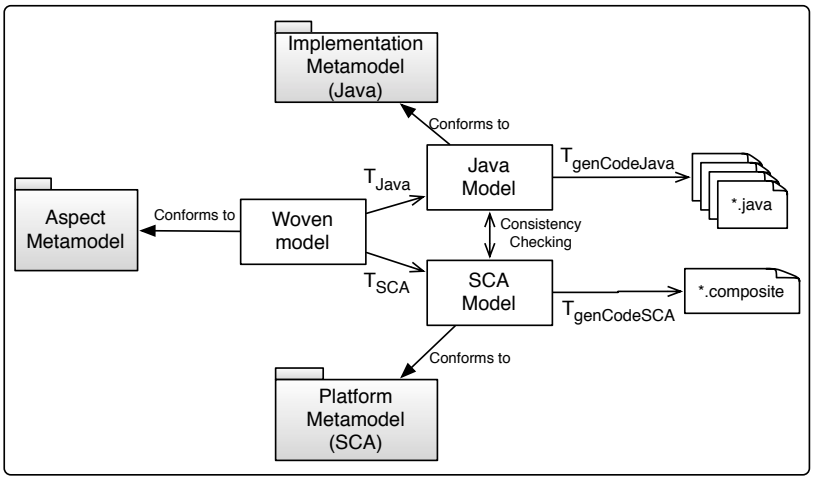

Figure 3: MDE transformation process.

3.2.2. From aspect models to runtime weaving

The second type of weaving is performed at runtime. The goal is to create the adaptation script required to perform an adaptation using the same aspect model that is used for design weaving, and execute the script at runtime. We use a Service Component Architecture (SCA) [21] implementation called `FraSCAti` [22] as target platform for our applications. In `FraSCAti`, Java-based SCA components are at the same time SCA-compliant and Fractal-compliant [23]. Thanks to this property, all the components in `FraSCAti` benefit from the loose coupling and platform agnostic properties from SCA, but also from the hierarchical and reflective characteristics of Fractal that

enable the product to be dynamically reconfigured. To achieve runtime weaving, we build a transformation that takes the aspect model and generates the needed reconfiguration scripts. Such transformation works as follows:

Model. The model is transformed into Java and SCA code, in the same way as the model that results from the design weaving. At the end of this process, the architecture and every specific service of the model are represented with SCA code as well as Java classes implementing each one of the services that conform the application.

Pointcuts. The pointcuts are transformed into FPath code [24]. FPath is a query language to navigate Fractal-based Architectures. It eases the navigation of component systems and enables developers to define queries that search for elements of the architecture that match some criterion. The goal of the pointcut transformation is to transform every `Expression` into the FPath script that finds the specific elements of the architecture of the application being executed. Hence, every atomic expression has an equivalence in terms of FPath. FPath also allows for multiple queries to be combined by using the '|' operator for a union and the '`&`' operator for an intersection. In this way, every `CompositeExpression` is translated into a union or an intersection of its sub-expressions. Table 1 summarizes the equivalences between the pointcut model and the FPath scripts.

Model Pointcut	**FPath Equivalent**	**Meaning**
FindByName(name)	*$root/descendant-or-self::*[name(.)== 'name']*	All the elements which name is equal to "name".
Type *- Service or Reference* *- Attribute* *- Element*	 *$root/descendant-or-self::*/interface::** *$root/descendant-or-self::*/attribute::** *$root/descendent-or-self::**	 All the interfaces. All the attributes. All the components.
Owned	*$root/descendent-or-self::*/child::**;	All the components owned by another component.
CompositeExpression *- Operator= OR* *- Operator= AND*	 *(exp1 | exp2)* or *union()* *(exp1 & exp2)* or *intersection()*	 Union of two expressions. Intersection of two expressions.

Table 1: Pointcut transformation

Advice. The advice is transformed into FScript code [24], which is a scripting language dedicated to architectural reconfigurations of Fractal-based systems. A reconfiguration in our case consists of two main steps: (1) find the place, and (2) perform the modifications. The former step corresponds to FPath code. Similarly to an advice that uses pointcuts to describe the places where it performs the modifications, FScript uses FPath to find the places in the architecture where the adaptation will be applied. For each variable required by the advice, there is an FPath script (generated from the pointcut). The latter step is a translation of the `Modify` instruction into FScript code.

Events. The context events are transformed into Java rules. For each observable, a method is generated. When a context event occurs, the corresponding method is invoked. This method has the conditions required to analyze the new value of the observable and to weave the corresponding aspect if needed. Such weaving corresponds to the execution of the FScript code generated from the advice.

So far, we have presented the language for defining aspect models, and the platform that uses those models at design and runtime. As it can be noticed, both design and runtime weaving have the same input i.e., an aspect model, but they rely on different processes and return different outputs. Design weaving can be seen as a refinement of the core using the variability and the choices of developers through feature selection and aspect weaving. Runtime weaving, on the other side, transforms the product to obtain a new version using the appropriate reconfiguration technologies. In both cases, the adaptation processes are transparent from the aspect definition. In the next section we present a case study where aspect models are used at both design and runtime in the implementation of a dynamic software product line.

4. Application to a Dynamic SPL

In this section, we illustrate with an example how the AOM detailed in Section 3 has been applied in *CAPucine*, a Dynamic SPL for context-aware applications [20]. Such a DSPL allows one to define a family of products using a feature diagram and then implements two phases: (1) an initial phase that deals with the design and automatic generation of applications, using models and model transformations, and (2) an iterative phase that supports runtime adaptations by relying on a context information manager to trigger reconfiguration scripts.

Here we show how AOM can be used to improve the DSPL by unifying design and runtime adaptations. We use aspect models to extend the DSPL by providing: (1) a unified language to describe a feature and the assets required to integrate it into a software product at any moment in the life cycle (including the execution), and (2) an automatized process to build the assets required in the iterative phase (in CAPucine, runtime feature derivation scripts were initially written by hand).

Let us consider the feature diagram of Figure 4. 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, (2) *optional* features (white circles), which can be chosen or not, and (3) *alternative* features (inverted arc), a special kind of optionality where the selection is realized among a limited set of exclusive (XOR) or non-exclusive (OR) alternatives.

The feature diagram presents the following features: (1) `Catalog`, corresponding to the functionality to browse and list a set of items, additionally, the catalog may use several filters (e.g. `ByDiscount, ByWeather, ByLocation`), (2) `Notification` which can be implemented using `Call` or `SMS`, (3) `Location` of the user that can be obtained via `GPS` and/or using `Wifi` triangulation, and finally (4)

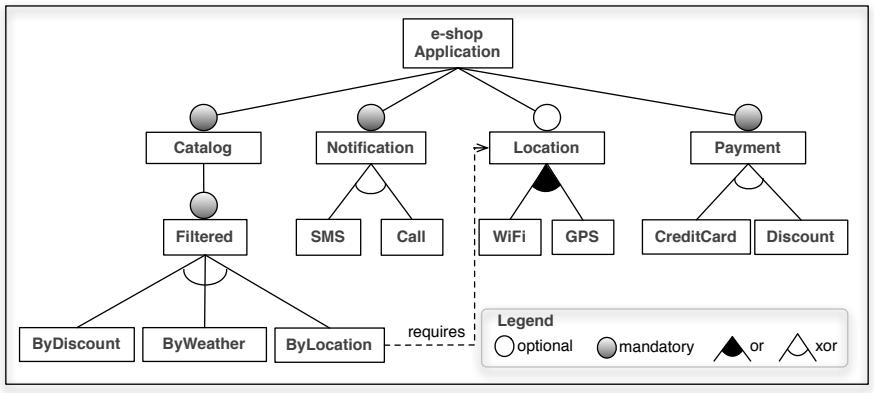

Figure 4: Feature Diagram.

`Payment` that can be performed through `CreditCard` or using a `Discount` ticket. Additionally, there is one constraint among features indicating that location-based filter requires that one type of location is selected. We applied our approach to create a high-level representation of the features. Each type of feature is modeled in a different way as explained below.

4.1. Mandatory Features

Since mandatory features are always present in every product, they are modeled all together as the core model. The core model of our example is illustrated in Figure 5. We notice that, in the core, only the model is relevant, hence it does not include advice, pointcuts, or events. The model holds a `Container` element which groups all the other elements in the architecture. There are four additional elements: `Catalog`, `FrontEnd`, `Notification`, and `Payment`. Besides the `Container` and the `FrontEnd`, which are created as part of the architecture, note that each element is intended to realize one of the mandatory features found in the feature model. For space reasons, we focus on the `Catalog` and `Notification` elements, we show the sub-elements they contain and how they are used in design and runtime weaving.

The `Catalog` element offers one service (`catalogQuery`) and requires one reference (`filterProducts`). As we will detail below, this reference is one join point where multiple filters get to be woven and used by the `Catalog`. Additionally, we specify the type of reference using the `CatalogFilter` contract which defines one operator. This simply indicates that in order for any external element to provide a compatible service for this reference, it must respect the same contract by implementing the same set of operations (in this case `getFilteredProducts`). In the same way, the `Notification` element offers one service `sendNotification` and requires one `smsNotification` typed with the `contractNotSMS` contract.

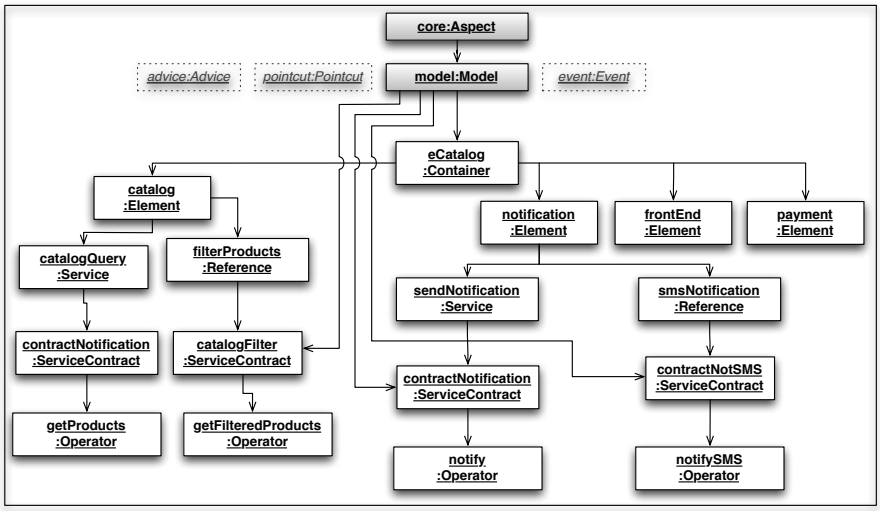

Figure 5: Example's core functionality.

4.2. Optional and Alternative Features

For each optional and alternative feature we define an independent aspect model. There are in total 9 different aspects (i.e. *discountFilter, weatherFilter, locationFilter, smsNotifier, callNotifier, creditCardPayment, discountPayment, triangulationLocation*, and *gpsLocation*).

In the remainder of this section, we describe two aspects (**smsNotifier** for design and **weatherFilter** for runtime), to illustrate how aspect models are defined and woven statically and dynamically with the core. Obviously, the same process is applied to all other aspects.

*4.2.1. Design weaving aspect: **SMS Notifier***

At design time, developers decide about the architecture of final products. Notification via SMS is a good example of design decision. It does not depends on context information. The decision whether to weave it or not come from a business requirements. Figure 6 presents the model for the SMS notifier aspect. It contains only three of the four parts of an aspect model (as an aspect that won't be woven at runtime, it does not contain an event description):

- `Advice`: the advice is the place when we describe the actions to be followed in order to weave the aspect. This aspect has one `Add` operation that puts together the `smsNotifier` element in the place defined by variable `reference`. Both `reference` and `smsNotifier` are defined in the pointcut and model parts of the aspect.

- `Pointcut`: the pointcut has one composite expression that accumulates two atomic expressions. The first atomic expression is a `FindByName` that searches all the referenced elements in the core called ”smsNotification”. The second atomic expression is an `InstanceOf` that finds all the referenced elements of the type `Reference`. Since the operator type of the composite expression is `AND`, the intersection of the results of these two atomic expressions is stored in variable `reference`.

- `Model`: the model contains one element called `smsNotifier`. This element provides a service that is compatible with the required reference in the core's `catalog` since it conforms to the `contractNotSMS` contract.

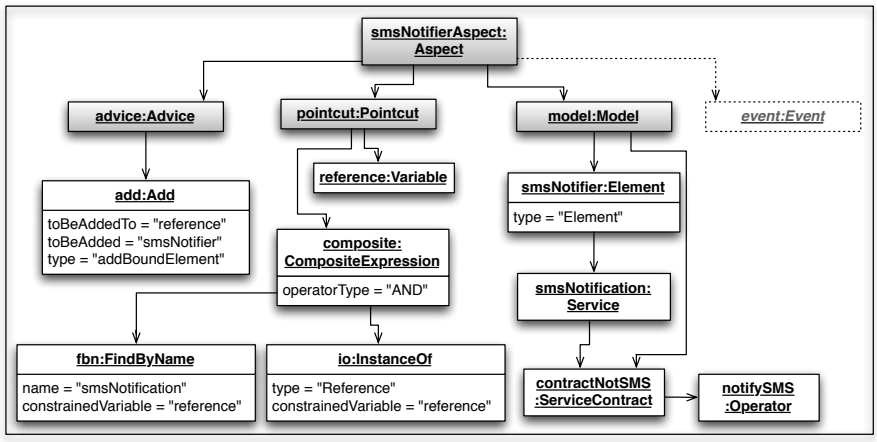

Figure 6: An aspect model for the SMS notification feature.

To weave the SMS notifier aspect, the advice modifications are executed one by one. In this case the weaver executes `Add`, which makes the element `smsNotifier` part of the whole architecture by adding it as one of the contained elements of the container `eCatalog`. Next, one connection is added to create a binding between the reference of the `catalog` and the service provided by the `smsNotifier`. Figure 7 depicts the core after the `weatherFilter` aspect has been woven. To simplify the diagram, we only show the services and references impacted by the weaving process. We use dashed lines to indicate which elements and relationships are added as a result of the weaving.

4.2.2. Runtime weaving aspect: ***Weather Filter***

Aspects that will be woven at runtime must have an event definition (see section 3.1). Regarding the **Weather Filter**, we have considered a context information related to the temperature. Figure 8 shows the contents of the weather filter aspect. Similar to the `smsNotifier`, the weather aspect defines: (1) a model with the elements

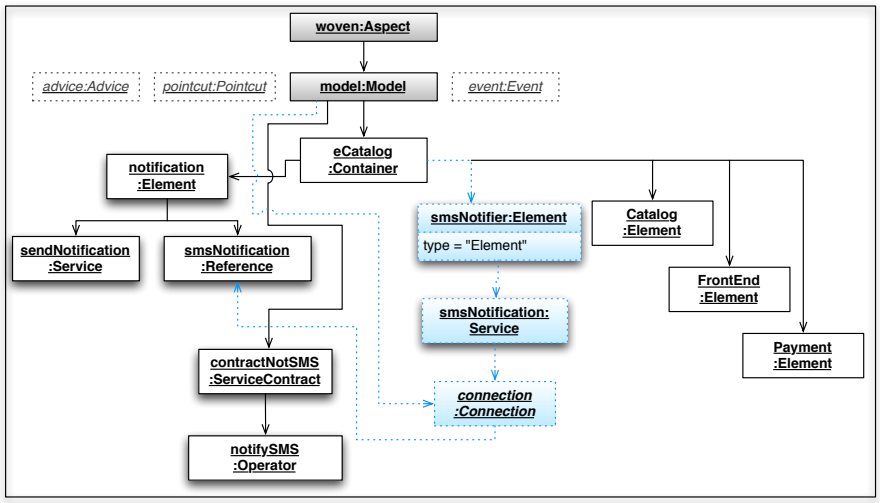

Figure 7: The result of weaving the core and the SMS notifier aspect.

to be added, (2) a pointcut that searches for a particular reference in the `Catalog`, and (3) an advice that consist in adding the `smsNotifier` as a bound element in the core architecture defined in the core model (Figure 5). What differentiates this aspect is its fourth part: the `Event`. It defines one condition over an `observable` called `coldWeather`. This means that dynamically, whenever the system receives a notification that the temperature is below a given threshold, the aspect must be woven.

The modifications performed by the weather filter aspect are equivalent to the ones of the design weaving. The difference resides in the technologies used to implement them. As explained in Section 3, starting from this model, an adaptation script and a Java rule that monitors the context information are generated.

Figure 9 illustrates a snippet of FScript generated for the advice example of Figure 8. In this case, the modification consist in adding a new Element `weatherFilter` and bind it to an existing component `catalog`. Finally, Figure 10 shows an example of a rule generated from the observable `coldWeather`

4.2.3. Dealing with more complex pointcuts

It is important to notice that the aspects presented in this section have a pointcut that matches a single element in the base architecture. However, there are some cases in which the variant needs to modify multiple places. For example, consider the aspect associated with the `ByLocation` variant. It needs to identify two different places in the architecture. First, it needs to find the filter mechanism found in the `Catalog` element and bound its model in the same way as the `ByWeather` implementing aspect. In addition to that, it also needs to find the service for location (Wifi or GPS) that it requires. For such cases, the structure presented in the aspect metamodel allows the aspects to define as many expressions as it needs to find the various places in the base

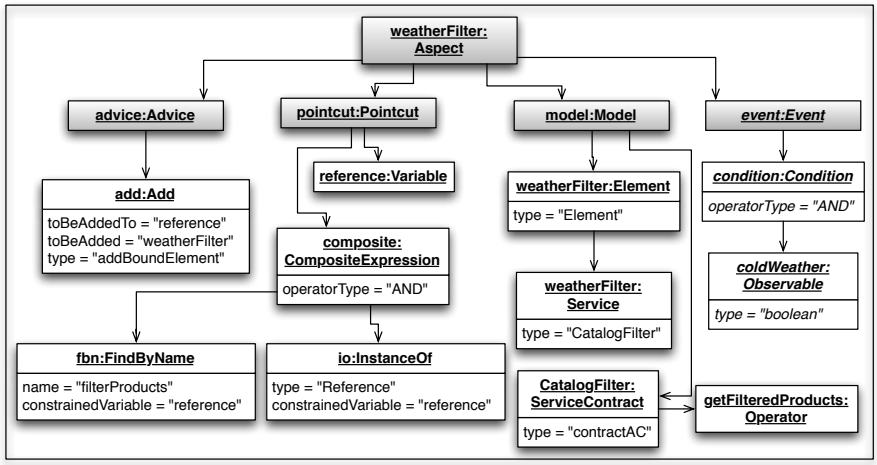

Figure 8: An aspect model for the weather filter feature.

architecture, and to create the modifications that represent the weaving of the aspect. The `ByLocation` implementing aspect can then define two different expressions for: (1) the reference in the catalog, and (2) the service provided by one of the location implementing aspects.

5. Discussion

In this section, we discuss the approach presented in this paper by justifying our choices and summarizing their advantages and their limitations.

5.1. Justification

On the difference between design and runtime adaptations. Applications can be adapted whether during the design phase or during the runtime phase. Design adaptations are motivated by design decisions, whereas runtime adaptations are motivated by changes in the executing environment. While a motivation of a design adaptation cannot be modeled (it is a choice), the motivation of a runtime adaptation can be modeled as a condition on the environment's state. Moreover, design and runtime adaptations are different also because they are realized by means of different mechanisms that use different technological platforms.

On the reason for unifying design and runtime adaptations. Although design and runtime adaptations have different *motivations*, and are *performed* using different mechanisms, they can be *specified* thanks to a unified language. Our aspect metamodel provides such an unified language. The main advantage is the fact that design and runtime adaptations are modeled thanks to aspects that share three principal descriptions:

```
1 action addElement()
2 {
3   --Step 1:Find the place(FPath)
4   current=$root/descendant-or-self::
5     *[name(.)== 'catalog'];
6   ref=$root/descendant-or-self::
7     */interface::*[name(.)== 'filterProducts']
8   new=$root/descendant-or-self::
9     *[name(.)== 'weatherFilter'];
10
11  --Step 2: Perform the adaptation (FScript)
12  stop($current);
13  stop($new);
14  bind($ref, $new/interface::weatherFilter);
15  start($current);
16  start($new);
17 }
```

Figure 9: An FScript equivalent of the **weatherFilter** advice.

```
1 public void changeColdWeather()
2 {
3         if(coldWeather.value())
4         {
5                 addElement(); //invoke reconfiguration
6         }
7         if(!coldWeather.value())
8         {
9                 removeElement(); //invoke reconfiguration
10        }
11 }
```

Figure 10: A java rule for a context event

the *what*, the *where* and the *how*. In fact, the only key difference comes from the fact that aspects that can be woven at runtime must have a description of the event: the *when*. As a consequence, aspects can be easily reused. Moreover, one can think about weaving aspects during the design phase, although they have been originally defined to be woven at runtime. Changing a runtime aspect into a design one is quite easy as it only consists in removing its event description. Vice versa, one can think about weaving aspects during the runtime phase, although they have been originally defined to be woven at design. Changing a design aspect into a runtime one is much more complex as it needs to add an event description.

On the necessity of weaving aspects at design time. One question that may naturally arise is: why do we need design weaving if we can perform any adaptations at runtime? Design weaving is important for three main reasons: automation, performance, and platform independence. Regarding automation, if there is no design weaving, the initial application have to be built manually. That is because runtime weaving just modifies an existing application. Regarding performance, it should be noted that all aspects that can be woven during the runtime phase need to define an event description. Regarding our example, it clearly appears that some aspects do not have any event

that motivates their weaving. Those aspects are definitively intrinsic design aspects. Without design weaving, we could weave them at the beginning of the execution, by defining a *fake* runtime event. This may potentially affect the performance of the application, while design weaving has no impact on performance. We firmly believe that intrinsic design aspects have to be woven as early as possible, and that runtime weaving only has to operate on aspects that depend on runtime events. Regarding platform independence, as we have presented in Section 3, obtaining a woven model through design weaving is only an intermediate step in the derivation of an executable product. The woven model obtained in our approach completely differs from the woven code produced by an AOP technology. In the AOP world, the woven code produced is language and platform-specific, and is not supposed to be further modified. In our approach, the woven model is a generic artefact that belongs to a high level of abstraction and constitutes a valuable input of a subsequent generation process, where it is still possible to make design decisions like execution platform and implementation languages.

On the use of aspect models for variability modeling. In our approach the use of aspect models provides a clear separation of concerns. It separates the core of the application from optional and possibly crosscutting functionalities. In our example, we have defined a set of variants expressed in a feature model. Each variant (or feature), being crosscutting or not, is represented through an aspect model. If a feature is not crosscutting, then the corresponding aspect pointcut is simple, as it only captures a single element of the base architecture. In contrast, if the feature is crosscutting, then the aspect pointcut and the aspect advice become more complex since they have to deal with multiple elements and different modifications. In summary, our approach allows aspects to be defined with multiple expressions and multiple modifications. Furthermore, the weavers at design time and at runtime are able to deal with these aspects. We consider that, regardless of the crosscutting nature of the variants, the proposed aspect metamodel and the two weaving processes provide the required flexibility for variants to be realized as aspect models and derivation to be defined as the weaving of such aspects.

5.2. Advantages

A unified and generic aspect metamodel. The use of a well-defined metamodel enhances the integration of aspects within complementary model-driven development strategies. This permits: (1) to define independent business models that are transformed into platform-specific models depending on the needs of a particular application, and (2) to have a unified approach in which software products and related adaptations can be modeled at the same time and derived from the same type of artifacts (e.g., aspect models). Furthermore, each aspect model is self-contained, and can be woven by a generic weaver that resolves the pointcuts, and then executes all modifications defined in the advice. The specified pointcut may be arbitrary complex to translate, for instance, a crosscutting feature; or very specific, like in the examples presented in section 4. The specified advice allows the weaving of both *monotonic* features (adding some func-

tionalities) and *non-monotonic* features (adding and removing some functionalities)[1]. Besides, since the specification of the reconfiguration actions is based on the aspect metamodel, we also prevent the execution platform to perform resource-consuming operations at runtime (like loading and computing a difference between two models). Furthermore, our approach enables dynamic reconfigurations, thanks to its integration with `FraSCAti`.

Graceful integration with (dynamic) software product lines. Finally, as we showed in Section 4, our approach can be used in the context of (D)SPLs, as a way to realize features and obtain assets for both the initial and iterative phases. In such a context, our approach provides support for product derivation at design time, based on the selection of optional features, in order to build an initial product. It also supports dynamic product reconfiguration at runtime, translating the iterative and automatic (de)selection of features in reaction to context changes.

5.3. Limitations

Dynamic adaptation of dynamic adaptations. Since we confine the definition of context-information to the design of an aspect model, new context observables, that were not initially specified by an aspect model, could not be taken into account. Our approach does not support (yet) the on-the-fly introduction of new observables and their corresponding adaptation rules, essentially due to technical limitations of the runtime platform.

Weaving order at runtime. Another limitation concerns the notion of weaving order. In the case of design weaving, we have recently proposed a constraint analysis approach aiming to derive a correct weaving order based on the explicit and implicit dependencies between the composed features [26]. At runtime, however, the weaving strategy may also depend on the context events triggering dynamic adaptations. More sophisticated techniques are needed to determine the set of aspects that must be (un)woven and the order according to which the (un)weaving operations must be performed.

Paradigmatic dependence. Finally, our current definition of advice is limited to the interaction between architecture elements that conform to our metamodel. Since we based our model in a component-based architecture, we only support coarse-grained adaptations of software systems following this architectural style, which facilitates model weaving in general and runtime weaving in particular. In order to support finer-grained adaptations like, for instance, changing the code inside every `Operator` of a given `ServiceContract` we would have to extend our model and verify the support for such operations in the platform level, which remains as future work.

[1] In the SPL domain, features are usually monotonic, as they are defined as *increments* in program functionality. However, composing non-monotonic features may also be required in certain cases [25].

6. Related Work

AOM proposes the use of MDE mechanism to describe and weave aspects. In [13], aspects are models and the weaving is realized thanks to model composition. Several other weaving mechanisms have been proposed such as Theme/UML [27, 28], composition directives for aspect-oriented class models [14], or multi-view composition [15]. Our approach does not contribute to this field and reuses existing model composition and code generation mechanisms.

Several approaches use AOM to express variability in SPL. In Perrouin et al.'s proposal [17], variants are specified by means of model fragments and the product derivation process consists in merging those fragments together. Voelter and Groher [29] combine AOM and MDE techniques to achieve an explicit separation of concerns in SPL context. In [30], variability is expressed by variation points and variants captured by means of feature models. The authors extend the FODA approach [31] with additional characteristics like cardinality and attributes. In [32], variability is defined by cardinality-based feature models. In [33, 34], the variability is both captured and expressed by cardinality-based feature models and the definition of metamodels.

In our approach, dynamic adaptations are supported thanks to the `FraSCAti` platform. Other approaches use different platforms such as FAC in [35], SAFRAN [19], Fractal [23] or OpenCOM [36, 37].

Work on adaptive systems and context awareness in SPL is also prolific. Bastida et al. [38] develop dynamic self-reconfiguring and context-aware compositions. They propose a multi-step methodology based on SPL notions of variability management. Their service composition infrastructure relies on Event-Condition-Action Rules. At runtime, a BPEL engine and a rule engine compose the middleware part, although the context information is not explicitly defined in the complete approach.

Bencomo et al. [39] propose software product lines for adaptive systems. In their approach, the execution context is related to corresponding changes by means of a state machine. Each state represents a particular variant of the system, while transitions define dynamic adaptations that are triggered by contextual events.

In [18], Morin et al. propose an approach that deals with dynamic variability in software product lines. Their approach relies on AOM for specifying variants and for realizing the binding of variation points. They also claim to propose a context-aware adaptation model that is in charge of selecting adequate variants depending on the context. Unfortunately, no detail is provided on how the context is specified and monitored. Our approach can therefore be considered as an extension of that work.

In [40], Morin et al. propose K@RT, a generic and extensible framework for managing DSPL. K@RT is an aspect-oriented and model-oriented framework for supervising component-based systems. It maintains a reference model at runtime allowing the navigation in the system architecture, and invokes services delegated to the running system. Each adaptation is supported by a safe reconfiguration script processed at runtime. In our approach, adaptations can be processed at the design or at the runtime, but they are prepared before the execution. In [41], Morin et al. go a step further with an improved AOM approach used to tame the combinatorial explosion of Dynamic Adaptive System modes. Using AOM, they compute a wide range of modes by weaving aspects into an explicit model reflecting the runtime system. Our approach is similar in

the sense that we use AOM and we attack dynamic adaptive systems, although unlike them, our approach does not use models at runtime as they do to calculate the next state of the application. All the aspects are defined at design, and runtime weaving takes place only if context information changes according to the rules defined in the aspects. In addition to that, we propose a solution for building the applications, our main goal is to define a unified representation of the adaptation with a single artifact (aspect model) for both design and runtime.

Zhang and Cheng [42] introduce an approach to create formal models for the behavior of adaptive programs. They separate adaptive from non-adaptive behavior of programs, making the models easier to specify. Our work focuses on architectural rather than behavioral models.

Finally, in [16], Lundesgaard et al. define how to construct and execute adaptable applications using an aspect-oriented and model-driven approach. They focus on managing multiple interacting cross-cutting Quality of Software (QoS) features. They propose Aspect-Oriented Modeling techniques and a QoS-aware middleware for execution of QoS sensitive applications. They specify alternative application variants and derive their runtime representation using model-driven engineering. The middleware chooses the best variant according to the current coperation context and available resources. Our approach is similar, although we make emphasis in unifying design and runtime modifications and we do not deal with QoS.

7. Conclusion

This article has presented an AOM-based approach to unifying design and runtime adaptations. The approach is made up of three phases: (1) aspect modeling to create aspects that conform to a generic aspect metamodel, (2) design weaving that links elements of the aspects at the model level, and (3) runtime weaving that links elements of the aspects dynamically by means of architectural reconfigurations. The design weaving phase is implemented by model transformations, while the adaptive platform `FraSCAti` supports the runtime weaving phase. To validate the approach we have used aspect models as a way to deal with variability in a dynamic SPL.

The main benefits of the proposed approach are (1) a clear separation of concerns, (2) a unified definition of design and runtime adaptations, (3) an explicit link between software adaptations and their motivations, and (4) a supporting platform that allows adaptations to be executed in different moments of the software life cycle, from the initial product configuration to its successive dynamic adaptations. As a result, our tool-supported approach allows software adaptation processes to reach high levels of reusability and flexibility.

We anticipate three main directions for future work. First, we would like to allow the system to process new pieces of context-information at runtime. So far, aspects are defined before the execution, which limits the adaptation possibilities to the set of foreseen observables. A second possible improvement concerns the aspect composition order, especially at runtime. We are currently exploring techniques to derive conflict-free weaving strategies based on (implicit) feature dependencies. Finally, we intend to consider finer-grained adaptations by allowing aspects to modify the code inside `Operator` elements.

Acknowledgements. This work is funded by the *Conseil Régional Nord-Pas-de-Calais*, *Oseo/ANVAR*, and the *Fonds Unique Interministériel*, under the `CAPPUCINO` project. This research was carried out during the tenure of an ERCIM "*Alain Bensoussan*" Fellowship.

References

[1] Nato software engineering conference, 1968.

[2] F. P. Brooks Jr., The mythical man-month: After 20 years, IEEE Software 12 (5) (1995) 57–60.

[3] S. Fraser, F. P. B. Jr., M. Fowler, R. Lopez, A. Namioka, L. M. Northrop, D. L. Parnas, D. A. Thomas, "no silver bullet" reloaded: retrospective on "essence and accidents of software engineering", in: Companion to the 22nd ACM SIGPLAN conference on Object-Oriented Programming Systems and Applications (OOPSLA'2007), 2007, pp. 1026–1030.

[4] M. Lehman, Laws of software evolution revisited, in: Proceedings of the 5th European Workshop on Software Process Technology, Vol. 1149 of Lecture Notes in Computer Science, Springer, 1996, pp. 108–124.

[5] P. Clements, L. Northrop, Software Product Lines : Practices and Patterns, Addison-Wesley Professional, 2001.

[6] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, Feature diagrams: A survey and a formal semantics, in: Proceedings of the 14th International Requirements Engineering Conference (RE'2006), IEEE Computer Society, 2006, pp. 136–145.

[7] M. Svahnberg, J. van Gurp, J. Bosch, A taxonomy of variability realization techniques, Software Practice & Experience 35 (8) (2005) 705–754.

[8] S. Deelstra, M. Sinnema, J. Bosch, Experiences in software product families: Problems and issues during product derivation, in: SPLC, Vol. 3154 of Lecture Notes in Computer Science, Springer, 2004, pp. 165–182.

[9] P. Tarr, H. Ossher, W. Harrison, S. M. Sutton, Jr., N degrees of separation: multi-dimensional separation of concerns, in: Proceedings of the 21st International Conference on Software Engineering (ICSE'1999), ACM, 1999, pp. 107–119.

[10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, J. Irwin, Aspect-oriented programming, in: Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'1997), 1997, pp. 220–242.

[11] D. C. Schmidt, Model-driven engineering, IEEE Computer 39 (2).

[12] Aspect-oriented modelling wokshops series, http://www.aspect-modeling.org/.

[13] J.-M. Jézéquel, Model driven design and aspect weaving, Software and System Modeling 7 (2) (2008) 209–218.

[14] Y. R. Reddy, S. Ghosh, R. B. France, G. Straw, J. M. Bieman, N. McEachen, E. Song, G. Georg, Directives for composing aspect-oriented design class models, T. Aspect-Oriented Software Development I 3880 (2006) 75–105.

[15] J. Kienzle, W. Al Abed, J. Klein, Aspect-oriented multi-view modeling, in: Proceedings of the 8th ACM international conference on Aspect-oriented software development (AOSD'2009), ACM, 2009, pp. 87–98.

[16] S. A. Lundesgaard, A. Solberg, J. Oldevik, R. B. France, J. Ø. Aagedal, F. Eliassen, Construction and execution of adaptable applications using an aspect-oriented and model driven approach, in: Proceedings of the 7th International Conference on Distributed Applications and Interoperable Systems (DAIS'2007), Vol. 4531 of Lecture Notes in Computer Science, Springer, 2007, pp. 76–89.

[17] G. Perrouin, J. Klein, N. Guelfi, J.-M. Jézéquel, Reconciling automation and flexibility in product derivation, in: 12th International Software Product Line Conference (SPLC 2008), IEEE Computer Society, 2008, pp. 339–348.

[18] B. Morin, F. Fleurey, N. Bencomo, J.-M. Jézéquel, A. Solberg, V. Dehlen, G. S. Blair, An aspect-oriented and model-driven approach for managing dynamic variability, in: Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS'2008), Vol. 5301 of Lecture Notes in Computer Science, Springer, 2008, pp. 782–796.

[19] P.-C. David, T. Ledoux, An aspect-oriented approach for developing self-adaptive fractal components, in: Software Composition, Vol. 4089 of Lecture Notes in Computer Science, Springer, 2006, pp. 82–97.

[20] C. Parra, X. Blanc, L. Duchien, Context awareness for dynamic service-oriented product lines, in: Proceedings of the 13th International Software Product Line Conference (SPLC 2009), IEEE Computer Society, 2009, pp. 131–140.

[21] Open SOA, Service component architecture specifications, http://www.osoa.org/display/Main/Service+Component+Architecture+Home.

[22] L. Seinturier, P. Merle, D. Fournier, N. Dolet, V. Schiavoni, J.-B. Stefani, Reconfigurable sca applications with the frascati platform, in: 6th International Conference on Service Computing (SCC'2009), 2009, pp. 268–275.

[23] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, J.-B. Stefani, The fractal component model and its support in java, Software Practice & Experience 36 (11-12) (2006) 1257–1284.

[24] OW2 consortium, Fscript and Fpath, http://fractal.objectweb.org/fscript/.

[25] M. Kuhlemann, D. Batory, C. Kästner, Safe composition of non-monotonic features, in: Proceedings of the 8th International Conference on Generative Programming and Component Engineering (GPCE'2009), ACM, 2009, pp. 177–186. doi:http://doi.acm.org/10.1145/1621607.1621634.

[26] C. Parra, A. Cleve, X. Blanc, L. Duchien, Feature-based composition of software architectures, in: Proceedings of the 4th European Conference on Software Architercture (ECSA'2010), Lecture Notes in Computer Science, Springer, 2010, pp. 230–245.

[27] S. Clarke, Extending standard uml with model composition semantics, Sci. Comput. Program. 44 (1) (2002) 71–100.

[28] E. L. A. Baniassad, S. Clarke, Theme: An approach for aspect-oriented analysis and design, in: ICSE, IEEE Computer Society, 2004, pp. 158–167.

[29] M. Voelter, I. Groher, Product line implementation using aspect-oriented and model-driven software development, in: Proceedings of the 11th International Software Product Line Conference (SPLC'2007), IEEE Computer Society, 2007, pp. 233–242.

[30] K. Czarnecki, M. Antkiewicz, Mapping features to models: A template approach based on superimposed variants, in: GPCE, Vol. 3676 of Lecture Notes in Computer Science, Springer, 2005, pp. 422–437.

[31] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, A. S. Peterson, Feature-oriented domain analysis (foda) feasibility study, Tech. rep., Carnegie-Mellon University Software Engineering Institute (November 1990).

[32] P. Sánchez, N. Loughran, L. Fuentes, A. Garcia, Engineering languages for specifying product-derivation processes in software product lines, in: Proceedings of the 1st International Conference on Software Language Engineering (SLE'2008), Springer, 2009, pp. 188–207.

[33] D. Wagelaar, Composition techniques for rule-based model transformation languages, in: Proceedings of the 1st International Conference on Theory and Practice of Model Transformations (ICMT'2008), Springer, 2008, pp. 152–167.

[34] H. Arboleda, A. Romero, R. Casallas, J.-C. Royer, Product derivation in a model-driven software product line using decision models, in: Proceedings of the Memorias de la XII Conferencia Iberoamericana de Software Engineering (CIbSE 2009), 2009, pp. 59–72.

[35] N. Pessemier, L. Seinturier, L. Duchien, T. Coupaye, A component-based and aspect-oriented model for software evolution, IJCAT 31 (1/2) (2008) 94–105.

[36] T. Batista, A. Joolia, G. Coulson, Managing dynamic reconfiguration in component-based systems, in: ECSA, Lecture Notes in Computer Science, 2005, pp. 1–17.

[37] G. Coulson, G. S. Blair, P. Grace, A. Joolia, K. Lee, J. Ueyama, A component model for building systems software, in: IASTED Conf. on Software Engineering and Applications, IASTED/ACTA Press, 2004, pp. 684–689.

[38] L. Bastida, F. J. Nieto, R. Tola, Context-aware service composition: a methodology and a case study, in: Proceedings of the 2nd international workshop on Systems development in SOA environments (SDSOA'2008), ACM, 2008, pp. 19–24.

[39] N. Bencomo, P. Sawyer, G. Blair, P. Grace, Dynamically adaptive systems are product lines too: Using model-driven techniques to capture dynamic variability of adaptive systems, in: Proceedings of the 2nd International Workshop on Dynamic Software Product Lines (DSPL'2008), 2008.

[40] B. Morin, O. Barais, J.-M. Jezequel, K@rt: An aspect-oriented and model-oriented framework for dynamic software product lines, in: Proceedings of the 3rd International Workshop on Models@Runtime, 2008.

[41] B. Morin, O. Barais, G. Nain, J.-M. Jezequel, Taming Dynamically Adaptive Systems with Models and Aspects, in: Proceedings of the 31st International Conference on Software Engineering (ICSE'2009), IEEE Computer Society, 2009, pp. 122–132.

[42] J. Zhang, B. H. C. Cheng, Model-based development of dynamically adaptive software, in: Proceedings of the 28th International Conference on Software Engineering (ICSE'2006), ACM, 2006, pp. 371–380.