Towards a Generic Aspect-Oriented Modeling Framework

Brice Morin, Olivier Barais, Jean-Marc Jézéquel, Rodrigo Ramos

To cite this version:


HAL Id: inria-00505222
https://inria.hal.science/inria-00505222
Submitted on 23 Jul 2010

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

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Towards a Generic Aspect-Oriented Modeling Framework

Brice Morin  
Olivier Barais  
Jean-Marc Jézéquel  
IRISA Rennes Projet Triskell  
Campus de Beaulieu  
F-35 042 Rennes Cedex  
{morin|barais|jezequel}@irisa.fr

Rodrigo Ramos  
Centre of Informatics  
Federal University of Pernambuco  
P.O. Box 7851, CEP 50732970, Recife, Brazil  
rte@cin.ufpe.br

ABSTRACT
Aspect-Oriented Modeling approaches propose to model reusable aspects, or cross-cutting concerns, that can be later on composed into various base systems. These approaches are often limited to a particular domain: UML class diagrams, UML sequence diagrams, ... and therefore they cannot easily be adapted to other domains. In this paper, we propose to extend the notion of aspect to encompass an open ended number of domains. We present our Generic Aspect-Oriented Modeling Framework and show how it can easily be specialized for any specific domain.

In this paper we propose a generic aspect-oriented modeling framework that can easily be adapted to different domains. We define the pointcut as a model snippet and the composition protocol as a set of adaptations or weaving directives. The remainder of this paper is organized as follows. The need for a generic aspect-oriented modeling framework is motivated in Section 2. Section 3 presents the generic template mechanism we use to point out join points while Section 4 presents our generic adaptation metamodel. Section 5 presents two applications of the framework and Section 6 concludes.

1. INTRODUCTION
Aspect-Oriented Modeling (AOM) approaches need to deal with two main activities: identifying in a base model points of interest or join points, where to compose aspects, and composing the aspects into the base model. Identifying join points in a base model require a mechanism to specify the match points: the pointcut. In most of the AOM approaches, the pointcut is a template model whose elements can match join points. Then, the aspect can be woven into the base model thanks to a composition protocol, or weaving directives. Both the pointcut expression and the composition protocol are domain-dependent.

In most of the AOM approaches, these two activities are often limited to a particular domain: UML Class Diagrams or Sequence Diagrams [2], State Machines [4], High-level Message State Charts [6]. We argue that the pointcut expression and the composition protocol could be generic. Identifying join points in a class diagram or in a Finite State Machine (FSM) relies on the same principle: matching model elements. Specifying a composition protocol for class diagram or a FSM needs domain-specific weaving directives, but the underlying process is similar in both cases and boils down to compose model elements.

In the class diagram, the aspect consists of making all the classes inheriting from a unique root class "RootClass". In other words, all the classes with no super class will inherit from "RootClass". In the FSM, the aspect consists in introducing a new state "Final" that can be reached from all the other states. In other words, all the states with no outgoing transition will be linked to "Final". Figure 2 shows the join points where the aspect can be composed, in each model.

The weaving of each aspect works as follows. In the class diagram, "ClassA" and "ClassB" are inheriting from "RootClass" while in the FSM model, "Final" can be reached from "B" and "C". Figure 3 illustrates the result of this composition.

We can see that these two aspects have strong similarities: in both cases a new element is introduced and then it is linked to existing model elements. We argue that an aspect-

2. A MOTIVATING EXAMPLE
This section illustrates the need for a generic aspect-oriented framework, adaptable to almost every domain. In this example, we will compose two "comparable" aspects from two different domains: Class Diagram and Finite State Machine (FSM). Figure 1 illustrates a simple class diagram and a FSM.

![Figure 1: Two models from two different domains](image_url)

In the class diagram, the aspect consists of making all the classes inheriting from a unique root class "RootClass". In other words, all the classes with no super class will inherit from "RootClass". In the FSM, the aspect consists in introducing a new state "Final" that can be reached from all the other states. In other words, all the states with no outgoing transition will be linked to "Final". Figure 2 shows the join points where the aspect can be composed, in each model.

The weaving of each aspect works as follows. In the class diagram, "ClassA" and "ClassB" are inheriting from "RootClass" while in the FSM model, "Final" can be reached from "B" and "C". Figure 3 illustrates the result of this composition.

We can see that these two aspects have strong similarities: in both cases a new element is introduced and then it is linked to existing model elements. We argue that an aspect-
oriented approach should be able to deal with different domains. Our generic aspect-oriented modeling framework is presented in the next two sections.

3. TEMPLATE MECHANISM

As suggested in the introduction, in most of the AOM approaches, the **pointcut** is a template model whose elements can match **join points**. This template model might be expressed as **model snippets**, such as the UML templates [3] in Figure 4. Having patterns expressed in this way, it is possible to allow a user to draw patterns using editors that he is used to, when drawing the models that he intends to match. This section presents the concept of model snippet, how we can build a template mechanism for any domain and how we perform the pattern matching [12].

3.1 Model snippet

Each model-snippet defines a set of information existing in the model that we wish to match. For example, in Figure 4, a class named *Trace* is declared with two methods (*traceEntry* and *traceExit*). Whenever, a class contains the same name and methods, it matches with *Trace*. Obviously, the matched class might have also more information, such as methods, attributes or associations.

The snippet in Figure 4 was constructed for UML models [3]. Many other domain specific languages could take advantage of a similar approach. In brief, we can define model snippets as:

A set of objects $S$ is a **model snippet** of a metamodel $MM$ if:

- every object in $S$ is an instance of a meta-class defined in $MM$;
- there exists a set $M$ where $M$ is a Valid Model w.r.t. $MM$ and $S$ is subset or equal to $M$.

where, we assume that a model and a metamodel are respectively sets of EMOF objects and classes.

Every valid model is also a snippet (w.r.t its own metamodel) and so is every model that can be obtained by removing objects from that model. But not every model that may be obtained by adding objects to such a model.

With this in mind, we show how we can express model-snippets in any domain. We present in Sect. 3.2 a pattern-framework with the minimal elements that form a pattern. In Sect. 3.3, we show how to create **model snippets** and to customise this framework according to a target metamodel.

3.2 Pattern-framework metamodel

Taking a closer look at the model-snippet in Figure 4, we can see the snippet as an instance of the UML metamodel. All elements in the snippets are instances of a UML classifier, and furthermore inherit from a superclass *NamedElement*. For instance, *Trace* is an instance of a UML *Class* identified with a feature *name* equal to 'Trace', and the method *traceEntry* is an instance of *Operator* with name equal to 'traceEntry'. The same happens in the model that we want to match. An important finding from this observation is that a snippet specifies a subset of instances, and of associations among them, in the model that we want to match. This set of instances is the information that we use to match models. However, a pattern seems to be a little more complex than just a set of instances of a metamodel.

It is clear from Figure 4 that a pattern is mainly formed by a **snippet** part (in each package of the figure) and a sequence of free-variables over this **snippet** part (in rectangle on the top-right side of each package). The purpose of variables is to define the selection criteria for a particular model element. A variant can also be conceptually seen as a placeholder for any element in the **intended model** that is matched to it. In most cases, variables represent elements that play a significant role in the pattern, and that we have a special interest in matching with. Contrary to variables, non-variables must be directly associated to a unique element in the **intended model**, containing all features which identify the element.

Nearly all metamodels define a special feature that uniquely identifies each element of their models. As we wish to be able to match variables with more than one element in the model, we do not take into account this identifier during pattern-matching of variables. For instance, *TracedClass* is
a variable in Figure 4. So it matches to any class, with any name, that has the same methods than TracedClass and an association to a class named 'Trace'. Trace is a non-variable, and, furthermore, we take into account its name during pattern-matching. As in most of the cases, UML uses a feature name as an identifier. However, the feature can change in other metamodels.

The more information is expressed in the structural part of the pattern, the more precise is the pattern-matching. However, an excessive and detailed snippet might also uncover all positive matches. For this reason, as any model, a pattern can have additional constraints, which help to better describe the pattern and to relate variables with other elements in the model.

Note that constraints between variables and non-variables in a pattern should still be valid after they have been matched with elements in the intended model. From this standpoint, constraints might also help to describe false positives in a pattern, improving the accuracy of the pattern-matching.

Based on the concepts presented above, we propose a generic metamodel for patterns illustrated in Figure 5. In this metamodel, Pattern represents the whole pattern, and Pattern Structure represents its structure. The structural part contains a PModel with a set of instances of classes (elements) from a given metamodel, related to the domain metamodel that describes models in which we look for matches (intended model). Pattern Structure has also a set of Roles, which express pattern variables in the structural part. Additionally, the pattern can also have some constraints, or invariants, that, here, might be expressed in OCL or Kerneta [10].

Figure 5 presents what we call a Pattern Framework. PM is the point (hot spots) where the framework can be adapted or specialized by the developer. The specialisation of our framework to the metamodel that describes the intended models is described in the next section.

![Pattern Framework](image)

**Figure 5: metamodel of the pattern framework**

### 3.3 Constructing model-snippets

Most of the time, the metamodel (MM) of the intended model is too restrictive to represent patterns. The reason for that is very simple: patterns have to be expressed in a higher level of abstraction, such as model-snippets (see definition in the beginning of Sect. 3.1). For example in the state machine metamodel, it is totally understandable that someone does not want to provide the mandatory event of a Transition, or even want to instantiate a Vertex, which is defined as abstract, in order to match over instances of State or PseudoState. We want a snippet to rely as much as possible on the same concepts of MM. In order to do that, we construct on demand a more flexible metamodel (MM’s) that allows us to represent abstract patterns with all concepts of the metamodel of the intended model. The flexible metamodel MM’s is equal to MM, except that:

- No invariant or pre-condition is defined in MM’s;
- All features of all classes in MM’ are optional;
- MM’ has no abstract element.

Then, we can notice that all concepts in MM are also represented in MM’. MM’ describe a wider range of models, including all models described by MM (see Figure 6). This is obtained by removing all the restrictions that exist in MM: invariants, mandatory features, nonexistence of instances of certain class. To allow a feature to be optional, we just set its lower bound as zero. All these restrictions can be expressed as invariants over a group of classes S, and any group of classes with a weaker invariant could be taken as a generalisation of S [14].

Note that any model that conforms to MM also conforms to MM’, and, furthermore, any model-snippet that conforms to MM also conforms to MM’. This is an important result, it shows that we can still use existing graphical editors to draw pattern snippets and use them for pattern-matching. It also means that any metamodel that generalizes MM can be used to specify more abstract patterns.

For example, we can generate a new and flexible metamodel (MM’) from a state machine metamodel (MM). MM’ might describe, for instance, a Region with zero InitialStates or a State with no Activity. It also describes Vertex as a concrete class, allowing instances of it.

Finally we need to merge our general framework for patterns (Figure 5) with this flexible metamodel (MM’), that generalizes the intended domain (MM). This composition is called a weaving because it integrates PObject from the Pattern Framework, as a superclass of all the meta-classes in MM’. This transformation can be compared to an interface introduction in AspectJ [1] that adds a new superclass to a type. Our weaving process is equivalent to this mechanism. The implicit pointcut used in our weaving applies the introduction of the PObject superclass into all the metaclasses with no superclass. The whole process to derive MM’ from MM and to permit the specification of valid pattern snippet (PAT_snippet) is presented in Figure 7.

For example, we can generate a new metamodel that weaves our pattern framework and the state machine metamodel, and
4. ADAPTATION METAMODEL

In this section, we first present our generic adaptation metamodel which is inspired from the SmartAdapters [8] approach for the composition of Java programs. Then, we explain how to specialize this framework for a specific domain metamodel.

4.1 Generic Adaptation Metamodel

The root element of the adaptation metamodel is the Adapter. An Adapter is composed of an Aspect and Adaptations. An Aspect is composed of a PatternModel (template) that is used to match base model elements, and a PModel (structure) that represents the aspect structure. Adaptation refers to PObjects (parameters) from the template or the aspect structure, and describe the composition protocol of the aspect. The generic adaptation metamodel is illustrated in Figure 8.

4.2 Specializing the Adaptation Metamodel

The Adaptation meta-class is abstract and therefore cannot be instantiated. This meta-class is an extension point of our framework. In order to specialize the framework for a specific domain, users can define in Kermeta [10] domain-specific adaptations that extends Adaptation. These specific adaptations must implement the execute method to specify how parameters are composed. For example, users can define an adaptation IntroduceTransition that inherits from Adaptation, and specifies which operations are needed to introduce a Transition between a source State and a target State.

Another complementary solution to specialize the framework for a specific domain is to automatically generate some adaptations e.g. creation or removal of model elements. We use the pattern matching framework to match metamodel snippets (M2 level), for a given meta-metamodel (M3 level: EMOF, ECore). Every pattern is associated with a template adaptation that should be concretized into some
domain-specific adaptations written in Kerneta, according to the computed bindings.

First, we need to generate an unconstrained ECore metamodel (or EMOF), as we explained in Section 3. Then we design some patterns relevant for code generation and write template adaptations. For example, in most of the domains, it would be very helpful to generate an adaptation that adds a new content to a container. We need a very simple pattern (Figure 9) composed of a class `Container` that is composed of classes `Containment` via a composition relationship `Composite`.

![Figure 9: Container pattern](image)

This pattern is associated with a template adaptation (Figure 10).

```java
class addContentInto[Container]Vis[Composite] inherits Adaptation {
    operation execute() : Void raises TypingException is do
        var container: [Container]
        var content: [Content]
        //Parameter typing
        if (container=void and content=void) then
            container.add(content)
        else //At least one parameter is not well typed
            raise TypingException, new
        end
    end
}
```

![Figure 10: Template adaptation](image)

We can search for this pattern in every metamodel, for example the FSM metamodel. There are two possible bindings because there are two containment relationship in this metamodel: a `FSM` contains states (`State`) and transitions (`Transition`). Then, one adaptation is generated for each binding: template parameters are substituted with corresponding elements matched in the FSM metamodel. Figure 11 shows the generated adaptation that adds a new state into a FSM. Generation of concrete adaptation is comparable to frame specification [9].

## 5. APPLICATION

When the framework is specialized, the user can design an adapter, specifying the template, the structure of the aspect, and several adaptations referring to the aspect template or structure model element, in order to specify how the aspect would be composed into the base model. Elements of the template are substituted with corresponding elements of the base model whereas elements of the structure are substituted with cloned elements.

We will describe the aspect presented in Section 2, for the FSM domain. The aspect is illustrated in Figure 12 and aims at introducing a final state for all the states without outgoing transitions. Then, the pattern matching is executed on the base model illustrated on the left of Figure 13.

![Figure 11: Generated concrete adaptation](image)

Figure 12: Designing the aspect for FSM

The composition protocol is very simple: an adaptation introduces the state “Final” into the base FSM, another adaptation introduces the transition between the state playing the role “Any” and the state “Final”. The state “Final” is unique, so it is introduced only once, while the transition is not unique, so a new clone is introduced for every binding. This protocol is applied for all the bindings as shown in Figure 13. Note that if “Final” were not unique, there would be two states “Final” after composition: one for “B” and one for “C”.

![Figure 13: Weaving the aspect into the base model](image)

Now, we can specialize the framework for class models and apply the aspect presented in Section 2. We define the aspect in Figure 14 that is syntactically very close to the previous one in the context of FSMs. This aspect aims at introducing a root class in a target class model. Then, we execute the pattern matching and compose the aspect for each binding, as shown in Figure 15.

The composition protocol is similar to the previous one in the context of FSMs: an adaptation introduces the class “RootClass” in the base class model, and a second adaptation makes the class bound to “Any” inherit from “RootClass”. The same process is applied for all the bindings, as illustrated in Figure 15.

Both applications are syntactically very close, but with a different semantic corresponding to their respective domain. Our framework can easily be specialized for different domains and can realize both applications.
6. CONCLUSION

In this paper we have presented our generic aspect-oriented modeling framework based on the Kernel Meta-Modeling language Kermeta\(^1\) [10]. This framework is inspired by the Smart/Adapters approach [8], but the template mechanism has been revised. Furthermore, the Smart/Adapters approach is domain-specific: it first focuses on Java program composition [8] and work is in progress in the domain of EMF models.

In future work, we will improve the automatic generation of domain-specific adaptations, so that the user has even less work to do. Currently we are working on the introduction of variability mechanisms in our approach to make our framework more flexible [7]. For example, it will possible to compose an aspect in different ways, declaring some parts of the protocol as alternatives with several possible variants of composition. We will also leverage the notion of model typing [14]. Currently, when the framework is customized for a given domain metamodel, aspects can only be composed into base models conforming to this metamodel. We want to generalize our approach, making it possible to apply an aspect on every model conforming to a subtype of the metamodel. Finally we will also study how to propose a complete aspect-oriented design process such as [3] based on our approach. In particular, we will have to focus on traceability: for example, we can assume that the class diagram and the FSM presented in this paper correspond to two different views of the same system, for two different phases of the lifecycle. We need to be able to trace the aspect from one phase to the next/previous.

7. REFERENCES


\(^1\)see www.kermeta.org