Weaving Aspect Configurations for Managing System Variability

Brice Morin, Olivier Barais, Jean-Marc Jézéquel

▶ To cite this version:

Brice Morin, Olivier Barais, Jean-Marc Jézéquel. Weaving Aspect Configurations for Managing System Variability. 2nd International Workshop on Variability Modelling of Software-intensive Systems, 2008, Essen, Germany, Germany. inria-00456484

HAL Id: inria-00456484
https://inria.hal.science/inria-00456484

Submitted on 15 Feb 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.

Weaving Aspect Configurations for Managing System Variability

Brice Morin, Olivier Barais[1] and Jean-Marc Jézéquel[1]
IRISA Rennes - Equipe Projet INRIA Triskell
[1]Université de Rennes 1
Campus de Beaulieu
F-35 042 Rennes Cedex

E-mail: {bmorin | barais | jezequel}@irisa.fr

Abstract

Variability management is a key concern in the software industry. It allows designers to rapidly propose applications that fit the environment and the user needs, with a certain Quality-of-Service level, by choosing adapted variants. While Aspect-Oriented Programming has been introduced for managing variability and complexity at the code level, the Software Product-Line community highlights the needs for variability in the earlier phases of the software lifecycle, where a system is generally described by means of models. In this paper, we propose a generic approach for weaving flexible and reusable aspects at a model level. By extending our generic Aspect-Oriented Modeling approach with variability, we can manage variability and complexity in the early phases of the software lifecycle.

1 Introduction

Variability management is a key concern in the software industry. It allows designers to rapidly propose a wide range of applications by choosing adapted variants and options. These customized systems will fit the environment and the user needs, with a certain Quality-of-Service level. In order to improve traceability, reliability and maintainability, variability should be explicitly modeled.

The Aspect-Oriented Software Development (AOSD) paradigm proposes to separate distinct concerns into different aspects *e.g.*, security, logging or persistency, and finally compose them into the base system. It first appeared at the code level [15] and has more recently gained attention in the earlier steps of the software life-cycle [4, 5, 8, 17, 28]: requirement, architecture, design, leading to the creation of numerous ad-hoc Aspect-Oriented Modeling (AOM) approaches, and a dispersion of effort in their tooling, documentation and adoption.

In order to manage variability, recent works [1, 2, 3, 13, 23] discuss the use of Aspect-Oriented Programming (AOP) for implementing Software Product Lines (SPL). At the code level, AOP offers mechanisms to encapsulate (optional) cross-cutting features. In contrast, with more traditional mechanisms like conditional compiling, these features would be tangled and scattered across the program.

Meanwhile, the SPL community points out the needs for managing the variability during the entire software lifecycle [22, 33], this in order to trace variability from requirements to implementation and even execution. The Model-Driven Engineering (MDE) paradigm proposes to consider models as first class entities during the entire life cycle. For example, requirement models [7] represent the user needs, class and component diagrams specify the structure of the system, scenarios and state machines specify its behavior, and runtime models [6] monitor the running system. MDE techniques allow to automate the transition between the different steps of the life cycle. All these models conform to different metamodels, and are generally described by Domain Specific Modeling Languages (DSML), or metamodels.

We argue that Aspect-Oriented Modeling (AOM) can help users to design optional and variant parts of a model, like AOP does at the code level. By weaving incrementally aspects into a base model it is possible to construct a final product step-by-step. But, to be able to weave aspect into different kinds of model, users have to adapt to numerous ad-hoc AOM approaches. Indeed, AOM approaches [4, 5, 8, 17, 28] often propose domain-specific mechanisms to represent aspects. We tackle this issue by automatically generating domain-specific AOM frameworks that all rely on the same concepts. Thus, designers do not need to adapt to a new AOM framework for all the domain metamodels they have to deal with. Weaving aspect

represents the first variability dimension of our approach.

Moreover, AOM approaches are often said flexible and reusable, but actually not enough. Using these approaches, it is often impossible to weave an aspect into a base model if it does not exactly propose what the aspect expects. Additionnaly, when it is possible to weave an aspect into the base model, it is always composed the same way. Based on previous work [18], we propose to integrate variability mechanisms into aspects themselves to tackle the issue of the limited reusability of aspects. These mechanisms turn standard aspects into configurable aspects, more reusable and flexible. Aspect configuration represents the second variability dimension of our approach.

The remainder of this paper is organized as follows. Section 2 presents our generic approach for aspect weaving. Section 3 details our 2-dimension approach for managing variability of software systems. Finally, Section 4 presents related works and Section 5 concludes and discusses future works.

2 Our Generic Model-Driven Approach for Aspect Weaving

This section presents our generic model-driven approach for aspect weaving. It briefly introduces the notion of metamodeling, with a simple running example, and introduces the notion of Aspect-Orientation. Then, it details our approach for automatically generating Aspect-Oriented Modeling frameworks, for any domain metamodel.

2.1 Metamodeling, AOP and AOM

Metamodeling A domain metamodel describes all the concepts of a particular domain of interest, and their relations. To illustrate our approach, we introduce a simple domain metamodel **MM** for state machines, illustrated in Figure 1. A region contains several vertices and transitions, that are the main elements of a state machine. Note that the `Vertex` meta-class is abstract and cannot be instantiated in a model, but is extended by two concrete meta-classes: `PseudoState` and `State`. A transition must declare a source and a target vertex.

This metamodel allows designers to represent state machines, with any number of transitions and vertices, in any configuration. Then, it is possible to simulate models or generate other artifacts, using Model-Driven Engineering techniques and dedicated tools like Kermeta [26], an open-source environment[1] for metamodel engineering. There exists metamodels for state machines, components, scenarios, class diagrams, etc.

[1] available at www.kermeta.org/download

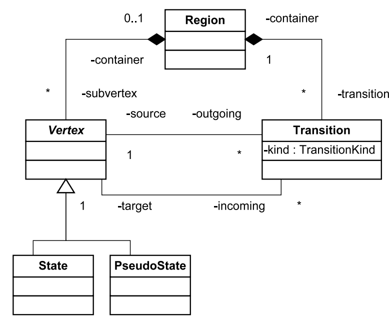

Figure 1. A Domain-Metamodel (MM)

Aspect-Oriented Programming (AOP) The Aspect-Oriented paradigm first appeared at the code level [15] and has been popularized with the AspectJ [14] programming language. AspectJ extends Java with the following concepts:

1. **Join Point**: point of interest in a program *e.g.*, method execution/call, attribute reading/writing.

2. **Pointcut**: it defines a set of join points where the aspect will intervene *e.g.*, all the calls to a given method.

3. **Advice**: it specifies the additional behavior that will modify the base program. It is executed in all the join points identified by a pointcut.

AOP allows users to encapsulate cross-cutting concerns into advice, and implicitly weave them into a base program, in all the join points identified by a pointcut. AOP significantly reduces the complexity of softwares a the code-level, by limiting the scattered and tangled code.

Aspect-Oriented Modeling (AOM) At a model level, AOM approaches [5, 10, 18, 29, 32] propose to encapsulate cross-cutting and reusable concerns. AOM concepts are comparable to AOP ones. But, as opposed to AOP, AOM mainly focus on the composition of structural and behavioral models, in the early phases of the software lifecycle, before implementation.

Template models represent what the aspect expects from the base model *i.e.*, the model elements needed to be able to weave the aspect into the base model, and their relations. Template do not need to be consistent models, for example, it can only be composed of a single operation, without representing its containing class, that is normally mandatory. Templates could be assimilated as pointcuts.

Then, aspect are woven into a base model. This is similar to advice weaving in AOP. On the one hand, symmetric AOM approaches [5, 10, 29, 32], that do not differentiate aspect and base, propose to systematically merge

all the corresponding concepts, and specify how to introduce non-shared ones. On the other hand, asymmetric approaches [18, 24, 30], that clearly differentiate aspect and base, propose to specify how to integrate the aspect. Generally, symmetric composition is a better way to compose homogeneous views of a given system, using a partially automated procedure, whereas asymmetric composition is a better way to introduce new concerns into models, and often offers better reusability, but the composition protocol must be explicited.

In the remainder of this paper, our running example focuses on state machines. However, our asymmetric approach is completely independent from any domain metamodel.

2.2 Generating the pointcut language

The previously introduced domain metamodel allows users to design consistent state machines, but it is too restrictive for designing aspects. For example, a template model might only be composed of a region with a vertex (whatever its type), a final state and a transition that links the vertex to the final state. We may also want the vertex to be an indirect source of the transition *i.e.*, it is possible to fire the (dashed) transition from vertex, directly or not. This model does not conform to **MM** because the `Vertex` meta-class cannot be instantiated and **MM** does not consider the semantic notion of indirect source. This template model is illustrated in Figure 2. Moreover, we want to be able to declare some elements as roles *i.e.*, elements that must be substituted by actual base model elements, whereas other elements can be seen as structural constraints that the pattern must respect. For example, if we want to modify all the *quitTransitions* from the region, we will declare the transition and the region as roles to be able to manipulate them. The other elements are just structural constrains: the transition must link a vertex to a final state.

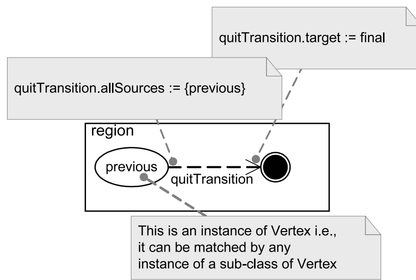

Figure 2. A Simple Template Model

In order to be able to describe more easily target models, we construct on demand a more flexible metamodel [27] **MM'**, using a model transformation written in Kermeta [26]. **MM'** is equivalent to **MM**, except that:

1. No invariant or pre-condition is defined in **MM'**;
2. All features of all meta-classes in **MM'** are optional;
3. **MM'** has no abstract element.

This model transformation is generic because instead of manipulating the domain elements (Vertex, Transition, ...), it manipulates higher-level concepts provided by ECore, MOF or EMOF for describing metamodels. Consequently, **MM'** can be generated for any input metamodel **MM**. Figure 3 illustrates the result of this transformation applied to the metamodel for state machines (Figure 1).

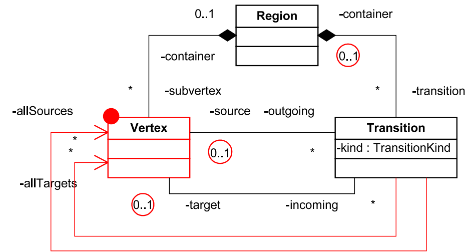

Figure 3. The Unconstrained Metamodel MM'

In **MM'** (Figure 3), we can see that a transition can declare no source/target vertex and can be instanciated without its containing region. Moreover, if a user wants to match a vertex, whatever its real type, he can now instantiate the `Vertex` meta-class. Additionally, we introduce two semantic associations (`allTargets` and `allSources`) to represent states (in)directly after or before a given transition. We also weave these associations as derived properties into **MM** to be able to compute all the state before/after a transition. Note that weaving derived properties does not change the metamodel, it only adds semantic.

In order to ease the detection of model elements that can match roles, we use a Prolog-based pattern matching engine [27], implemented in Kermeta [26]. The domain metamodel is automatically mapped onto a Prolog knowledge base. Then, patterns with roles are transformed into a Prolog queries over this knowledge base. Finally, the Prolog results are converted back into a Kermeta data-structure. This process is totally hidden from the user who only designs model snippets like the one presented in Figure 2.

2.3 Generating Adaptations

The second step of an Aspect-Oriented approach is the weaving process. It consists in composing aspects into the base model, at the places identified by the template model. The key concept is the adapter [18, 19], that describes the aspect structure (**what** will be woven), a template model with roles(**where** it will be woven) and a composition protocol (**how** it will be woven). The composition protocol is

described by adaptations, that are weaving operations manipulating the concepts of the domain. For example, in the context of class diagrams, an adaptation can add a super class to another class, introduce methods or attributes in a class, etc. These concepts are structured in the adaptation metamodel illustrated in the top-part of Figure 4.

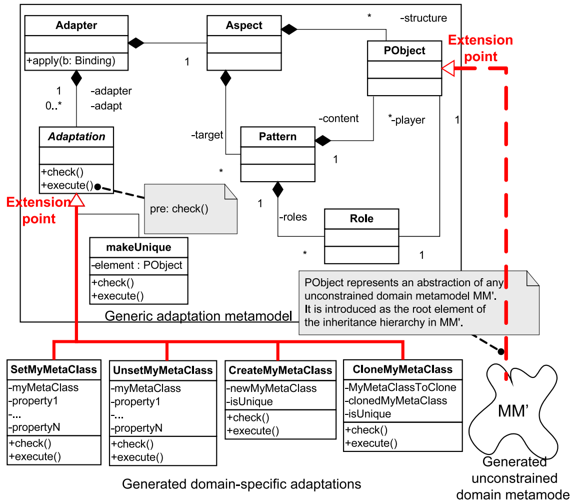

Figure 4. A Framework for Aspect Weaving

This metamodel is composed of three parts: *i)* a generic part describing the concept of adapter, adaptations and aspect (structure and target), *ii)* the unconstrained metamodel **MM'** that is linked to the generic part by introducing the meta-class `PObject` as the root element of **MM'**, and *iii)* domain specific adaptations extending the generic meta-class `Adaptation` (bottom-part of Figure 4).

We propose a systematic way to generate domain-specific adaptations. For each meta-classes `MyMetaClass` of a metamodel **MM**, we generate four adaptations:

1. **SetPropertiesOfMyMetaClass**: this adaptation allows user to set or update (addition) any property of `MyMetaClass`. For example, **SetPropertiesOfRegion** allows designers to add states and transitions in a region.
2. **UnsetPropertiesOfMyMetaClass**: this adaptation allows user to unset or update (removal) any property of `MyMetaClass`. Similarly, **UnsetPropertiesOfRegion** allows designers to remove states and transitions in a region.
3. **CreateMyMetaClass**: this adaptation allows user to create a new instance of `MyMetaClass`. It is generated only if `MyMetaClass` is concrete. For example, **CreateState** allows designers to create a new state, that can be manipulated in the remainder of the composition protocol.
4. **CloneMyMetaClass**: this adaptation allows user to clone an existing instance of `MyMetaClass`. It is generated only if `MyMetaClass` is concrete. Similarly, **CloneState** allows designers to clone an existing, and manipulate it.

The generation of these adaptations is also generic and can be done for any metamodel **MM**: we navigate the meta-classes and their properties and use Kermeta Emitter Template (KET) to generate all the above adaptations, specific to a domain metamodel. In our approach, we use KET to generate Kermeta files, but we can generate any kind of files such as Java code or textual documentation, by defining template. A template describes the structure of the output files (Kermeta, Java, text, etc), and the navigation is written in Kermeta, encapsulated in specific marks[2].

All these generated adaptations can manipulate elements from the template model or from the aspect structure *i.e.*, composition protocols written with these adaptations are totally independent from any base model, and can be reused in different contexts.

This section briefly exposed the principles of our generic model driven approach for aspect weaving. Our approach can be customized for any domain metamodel, to obtain a domain specific AO framework, through two extension points (Figure 4):

1. **PObject**: represents an abstraction of any (unconstrained) domain metamodel that allows us to describe the adaptation metamodel with no domain concepts. When we specialize the framework for a given domain, `PObject` is automatically introduced as the root meta-class of all the element of **MM'**, with a model transformation written in Kermeta [26].
2. **Adaptation**: represents an abstraction of any domain-specific weaving operation. All the domain-specific adaptations must extend this meta-class, declare some attributes, and implement the *execute* method that describes a composition between some model elements. We automatically generate some basic adaptations, but designers can create some additional adaptations that extends `Adaptation`, or modify existing ones.

3 Two-Dimension Variability Management

In the previous section we present our approach for generating Aspect-Oriented Modeling frameworks, for any domain with a well defined metamodel. In this section, we extend these AOM frameworks and describe our 2-dimension approach for managing the variability of software systems.

[2]similarly to Java code encapsulated in JSP or JET

The main idea is that each aspect is considered as a variability dimension *i.e.*, aspects integrate variability mechanisms to make them configurable and reusable in different contexts. Then, different configurations of an aspect can be woven (or not) in order to propose different variants of the system.

3.1 Variability Mechanisms for Aspects

The variability mechanisms we propose to integrate in the aspects are inspired by SPL approaches [31, 34][3]:

- **Alternatives/Variants**: specify that there exist several possible ways to compose the aspect (composition variability) and/or several different places where to compose it (targeting variability). All the variants are exclusive *i.e.*, we can only choose exactly one variant per alternative.
- **Options**: specify that some adaptations may be executed or not, and that some elements from the template model are not mandatory *i.e.*, they may be present or not in the base model where we want to weave the aspect.
- **Constraints**: control the variability mechanisms and limit the number of derived aspects to sensible ones. Without constraints, the number of possible combinations may become huge, and most of them would not be sensible. For example, we can easily imagine that some options or variants require (dependency) or exclude (mutual exclusion) some others.

We propose variability both for the composition protocol and for the targeting. For composition variability, we only need to apply the above concepts on adaptations, and integrate them in the adaptation metamodel. For the targeting variability, **MM'** does not allow designers to propose the full possible range of variability in their snippets because it is not possible to propose variants on certain features that have for example a [0..1] cardinality. For example, we can imagine that we want to instantiate a transition that targets either a pseudo-state or a state, and not simply a vertex, because the composition protocol uses adaptations that are specific to pseudo-state or state, in two distinct variants of an alternative. In order to propose variability in the target model, we propose to generate the maximum metamodel **MM"** that is equivalent to **MM'**, except that all the features can be multiple *i.e.*, all the upper bound are set to * (possibly infinite).

In order to allow composition and target variability, we extend the adaptation metamodel (Figure 4) presented in Section 2 with the following key concepts (see Figure 5):

[3] see http://www.sei.cmu.edu/productlines/ and http://www.splc.net

- **Derivable Adapter**: a derivable adapter is an adapter that contains variability *i.e.*, alternatives, options and constraints. It proposes both composition and targeting variability.
- **Adapter Element**: an adapter element is an element that can be optional or involved in an alternative: adaptation, target, alternative, conjunction (group of adapter elements). It is introduced as a super metaclass for all these elements.
- **Alternative**: an alternative describes several possible variants that are mutually exclusive. Each variant is an adapter element.
- **Constraints**: a constraint describes either a dependency or a mutual exclusion between some adapter elements. A dependency specifies that a source element requires some other elements, and an exclusion specifies that some elements are mutually exclusive *i.e.*, two elements cannot be present at the same time, after derivation.
- **Derivation**: a derivation allows designers to derive a derivable adapter *i.e.*, to fix variability. It allows designers to select options, and choose one variant, for each alternative.
- **Conjunction**: a conjunction is a block of dependent adapter elements. It allows to define optional blocks and variant blocks in an alternative.

To illustrate some of the variability mechanisms, the generated adaptations and the target model specific to state machines, we will describe an aspect that adds a *Log* state before reaching the final state *e.g.*, for logging errors. Optionally, we propose to come back to a previous vertex after logging an error, instead of reaching the final state. The target model and the aspect structure are the model snippets shown in Figure 6.

In the target model, the containing region, the final state, the transition that targets the final state are mandatory *i.e.*, they must be matched by actual base model elements before weaving the aspect into a base model. An option specify that we can target any vertex before the transition. All the elements of the target model are associated to roles, because we want these elements to be bound to base model elements, in order to modify the base model.

Now, we need to define the composition protocol that will describe how the structure will be woven into any base model. Note that the composition protocol is totally defined with elements from the target model and from the aspect structure *i.e.*, it does not reference elements from any base model. This protocol is illustrated in Figure 7.

The composition protocol describes the operations needed for integrating the aspect. In this example, all the

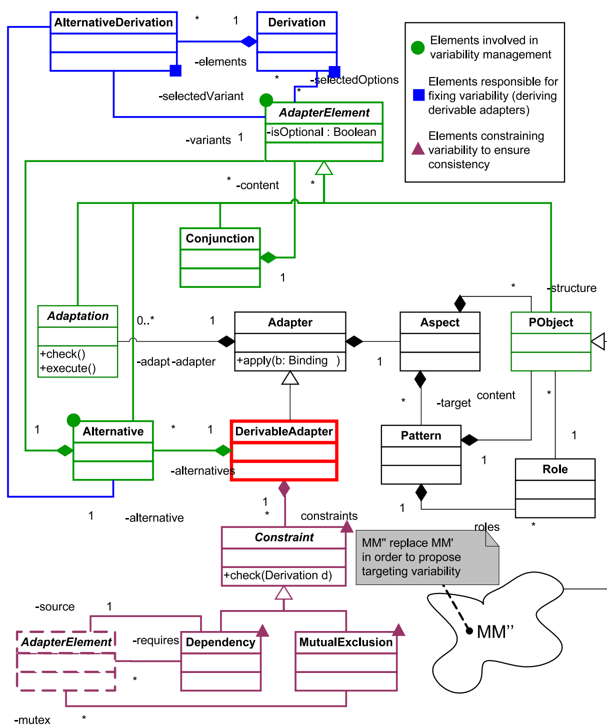

Figure 5. Extended Adaptation Metamodel

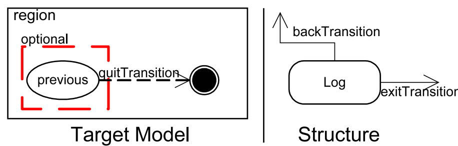

Figure 6. Target Model and Structure

adaptations are `Set*` adaptations because the aspect only adds model elements that exist in the aspect structure.

The concrete syntax we propose for adaptations is very basic. For example, the first adaptations (Line *01*) is called **introduceStruct** and its real type is `SetRegion`. Its first parameter is the region to set (Line *01-a*), and all the following parameters refer to the element we want to introduce in the targeted region: some subvertices (Line *01-b*) and transitions (Line *01-c*). The three following adaptations aims at connecting the transitions (Lines *2* and *3*) and renaming the *Log* state (Line *04*) to fit its context. Finally, we declare an optional conjunction (Lines *5*) that aims at introducing and connecting the *backTransition*.

Note that MDE tools like Sintaks[4] [25] can easily bridge abstract syntax (metamodel) and concrete syntax (text), by parsing texts into models, and transforming models into texts, according to rules defined in a Sintaks model.

[4]available at http://www.kermeta.org/sintaks

```
01 Adaptation introduceStruct SetPropertiesOfRegion
  a - region: target.region
  b - subvertex: {struct.Log}
  c - transition: {struct.exitTransition}
02 Adaptation setExitTrans SetPropertiesOfTransition
  a - transition: struct.exitTransition
  b - target: target.finalState
03 Adaptation setQuitTrans SetPropertiesOfTransition
  a - transition: target.quitTransition
  b - target: struct.Log
04 Adaptation setLogName SetPropertiesOfState
  a - state: struct.Log
  b - name: target.exitTransition.name + ``Log''
05 Conjunction backToPrev is optional {
 06 Adaptation introduceBack SetPropertiesOfRegion
   a  - region: target.region
   b  - transition: {struct.backTransition}
 07 Adaptation setBack SetPropertiesOfTransition
   a  - transition: struct.backTransition
   b  - target: target.previous
 08 TargetRef target.previous
}
```

Figure 7. Composition Protocol

3.2 Weaving Aspect Configurations

The previous sub-section details the first variability dimension *i.e.*, the integration of variability mechanisms into aspects. This sub-section details the second variability dimension: the configuration, or derivation of aspects and the weaving process.

The aspect presented in the previous sub-section (Figures 6 and 7) can be configured in two different ways, and consequently there are three possible variants:

- **Variant 1**: Do not weave the aspect
- **Variant 2**: Just add the *Log* state
- **Variant 3**: Variant 2, and we add a transition back to a previous state

If we consider several aspects, we can easily propose many different variants of the system by configuring aspects and weaving them, or not, into the base system. The derivation process can be summarized as follows:

1. Constraints: we check that the derivation *d* provided by the user respects all the constraints of the derivable adapter. We just call the *check(d)* method for all the constraints of the adapter, that is implemented directly in the adaptation metamodel (see `Constraint` in Figure 5), with Kermeta. If one constraint is not reached, the framework raises an exception telling the user that his derivation is not well-formed.
2. Adaptations: the composition protocol (adaptations) of the derived adapter is built in a positive way *i.e.*, selected options and variants are added into the derived adapter.
3. Target Model: the target model is (un)built in a negative way *i.e.*, the model elements that are not selected

(non-chosen options and variants) are deleted from the target model.

4. Post-condition: after derivation, the target model must conform to **MM'**, and not only to **MM''**. Otherwise it means that a cardinality is over the maximum bounds, and consequently the target model cannot be matched by any model snippet.

When an aspect is successfully configured, it can be woven into a base model, following this process:

1. Binding phase: the user provides a binding that links target model elements to actual base model elements. Note that bindings can automatically be found/checked using the pattern matching framework of Ramos *et al.* [27], to guide the user.

2. Weaving phase: for each binding selected by the user, we apply the composition protocol. In the adaptations, the target model elements are substituted with their corresponding actual base model elements, according to the binding. Between each binding, some elements of the aspect structure, or cloned/created elements (`Clone/Create*` adaptations), can be cloned, or remain unchanged. This choice depends on whether the user wants to use the same instances or introduce new instances, for each binding.

3. Post-condition : after composition, the modified base model must conform to **MM**, and not only to **MM'** or **MM''**. Otherwise it would mean that the composition protocol violates some constraints (*e.g.*, it removes mandatory features), or adds too many elements. In this case, we roll back to the initial base model.

The process can be applied several times and is potentially infinite and/or nondeterministic: if we consider that the process has been applied (n-1) times, we denote resp. $configuration^n$, $binding^n$, $weaving^n$, resp. the aspect configuration, the chosen binding and the result after weaving, for the n-th time. We have: $binding^n$=f($configuration^n$, $weaving^{n-1}$) and $weaving^n$=g($configuration^n$, $binding^n$)=h($configuration^n$, $weaving^{n-1}$). The configuration of an aspect may change the target model, and the previous weaving modify the base model, and potentially adds/removes possible targets, so the binding is dependent from the configuration and the previous weaving. The weaving depends on the aspect configuration (the selected adaptations) and on the selected binding, and consequently, it depends on the previous weaving. For this reason, the process is not fully automated: the user configures the aspect, then he chooses the binding and the composition protocol is applied. Next, he can reconfigure the aspect, choose another binding, etc.

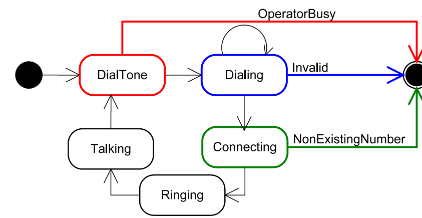

Figure 8. Basic behavior of a phone

Figure 8 illustrates a base model representing the behavior of a simple phone.

Figure 9 illustrates the composition of the aspect when no option is selected. In this case, we only introduce a *Log* state before reaching the final state.

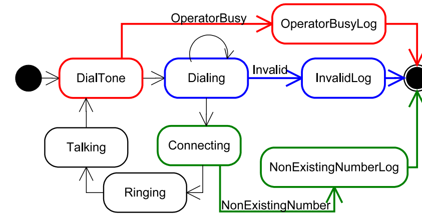

Figure 9. Behavior of a phone with error logging

Finally, Figure 10 illustrates the composition of the aspect when the option is selected. In this case, we also introduce a *Log* state before reaching the final state. Additionally, we introduce a roll-back transition that targets a previous state.

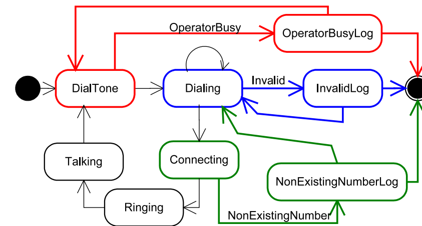

Figure 10. Behavior of a phone with error logging and roll-back

Note that is possible to combine different combination of the aspect to exactly fit the user needs.

4 Related Works

Our approach extends the SMARTADAPTERS approach [18, 19] by *i)* generalizing its concepts to any domain metamodel [24] (not only Java programs and UML

class diagrams), and *ii)* representing targets as model snippets [27], instead of declaring targets and constraints one by one. In [18], we introduce variability mechanisms in the base SMARTADAPTERS approach for class diagrams. In our generic approach, we also integrate these mechanisms (Section 3), in a slightly different way. Thus, we can propose configurable aspect, and weave them into models conforming to any domain metamodel.

Recent works discuss the use of Aspect-Oriented Programming (AOP) for managing variability at the code level, and implementing Software Product Lines (SPL). Some of these approaches advocate AOP for managing optional and variants cross-cutting features [2], or extracting and evolving SPL from a single application [1], and propose different variants, while some approaches like [13], insist on limitations and drawbacks of AspectJ for SPL implementation: code readability and maintainability, pointcut fragility making aspect weaving difficult. Moreover they point out that most of the mechanisms specific to AspectJ are not useful in most of the cases. Mezini *et al.* [23] point out the limitations of feature-oriented approach and AspectJ, especially its pointcut mechanism, and propose to use CaesarJ for resolving these problems. AOP is an interesting but still immature technology for managing variability. The combination of Aspect-Oriented Modeling (AOM) and Model-Driven Engineering (MDE) makes our approach more abstract and independent from problems inherent to the source code level. Unlike AspectJ pointcuts, our target models are totally independent from any base models and our aspect can be reused in different contexts, by binding target model elements to actual base model elements. There is no need for modifying the target model (pointcut), the aspect structure or the composition protocol (advice). Finally, AOP approaches for managing variability only propose one variability dimension and do not propose variability into the aspect itself, as we do.

In [21], Loughran *et al.* propose an approach that combines notions from AOP, frame technology and Feature-Oriented Domain Analysis (FODA). AOP aims at modularizing cross-cutting concerns and frame technology provide some means to configure aspects and make them context-independent and thus, more reusable. Using our approach, designers can also define context independent aspects using targets and adaptations that only reference elements from the aspect template or structure, and not directly base model elements. They use the variability mechanisms (alternative and options) of FODA models to represent the whole system *e.g.*, a generic cache. Then, they can delineate framed aspects and implement them in a reusable way using the frame technology. Frame is a fine mechanism to parameterize for example, the name and the type of attribute, method, parameters. In our approach, we use alternatives, options and constraints inside the aspect itself, for managing the different possible configurations. Frames are similar to our target model: both framed parameter and target model elements are substituted with actual elements from the base program/model, using bindings. Framed-aspect do not really propose internal variability, only configuration. Finally, both approach propose two variability dimensions, but they mainly focus on the system variability while we mainly focus on the aspect variability.

In [30], Schauerhuber *et al.* propose a common reference architecture for Aspect-Oriented Modeling. The concepts they identify are quite similar to the ones identified by Lahire *et al.* in the SMARTADAPTERS approach [18, 19], that we leverage to generalize the concepts of AOM to any domain metamodel. The approach of Schauerhuber *et al.* is also language-independent and may be applied for any domain metamodel. But, they do not propose means to generate the pointcut language nor domain-specific adaptations. Our generative approach, based on MDE techniques, allows designers to automatically specialize our framework, for any domain, by generating an unconstrained domain metamodel for designing target models (pointcuts), and generating domain-specific adaptations dealing with updating (addition/removal), creating and cloning elements. Moreover, they do not propose variability mechanisms, whereas we introduce mechanisms inspired by Software Product Line approaches.

In [16], Kim *et al.* combine this reference AOM architecture with a component-based SPL architecture. They propose to model variability using aspects, as we do in this paper. The variability mechanism is the *variability point* that is equivalent to our *alternatives* and *options*. In their architectures, they do not reify the notions of *constraints*, and do not really explicit how variants are selected, with their *variability point bindings*. In our metamodel, *constraints* and *derivation* clearly specify the dependencies between variants, and how to derive variants.

In [12], Whittle *et al.* propose the MATA (Modeling Aspects Using a Transformation Approach) tool for composing features in UML models (class diagrams, state charts and scenarios), based on graph rewriting. MATA allows user to describe the composition using stereotypes directly in feature models. The stereotypes they propose for composing features are similar to our Create/Set/Unset adaptations, but we also propose cloning adaptations. This can be useful, for example to implement a proxy, where all the operation needs to duplicate. Their notion of variable is equivalent to our notion of role *i.e.*, elements that can be substituted. With **MM'** and **MM"**, we can create more generic pattern by instantianting abstract elements and defining unconstrained models. Moreover, we propose variability mechanisms both for the matching and the composition whereas they only propose one variability dimension.

In [10], Fleurey *et al.* generalize the *Composition Di-*

rectives approach [29] and propose "a generic approach for automatic model composition", that can be adapted to any metamodel. This approach is based on signature matching and systematic merging of model elements. Their symmetric approach aims at merging different views of the same system *e.g.*, marketing and management views in order to obtain an integrated view of the system, using an automated weaving process that can be customized. Our asymmetric approach is different and aims at composing aspects, that can be considered as reusable patterns, into different base models, using parameterized composition protocols. Fleurey *et al.* do not propose variability mechanisms, but users can customize the matching by defining the signature of model elements, and customize the merging with context-specific composition directives. They do not propose alternatives, options and constraints for managing all the possible variants and consequently designers have to define as many aspects as possible configurations. Our approach allows designers to model an aspect per concern, with all the possible configurations. Then users select the most appropriate configurations to weave into their models.

In [11], Heidenreich *et al.* propose to extend the "Aspect Orientation for Your Language of Choice". Their generic approach is based on the Invasive Software Composition (ISC). Both base model and aspect model elements are annotated with *Slot*, *Hook* and *Anchor*. A slot indicates that a base element can be replaced by an aspect element with an anchor whereas a hook indicates a place in the base model where some anchored elements from the aspect can be added. They illustrate their approach on a UML class diagram and a Java program. Our approach is also generic but do not need to modify base models to make them aspect aware, letting base model oblivious of the aspect. We only use a binding mechanism before composition.

5 Conclusion

In this paper, we have presented our generic model-driven approach for aspect weaving: for any metamodel describing a given language or domain, we can generate both the targeting language and some weaving instructions that allow users to design reusable aspects. Then, we have extended this generic approach with variability mechanisms, and presented our 2-dimension approach for variability management. After deriving an aspect by choosing most appropriate variants and options, aspect configurations can be woven into base models, to integrate new features and propose different variants of the system.

In future work, we will extend our 2-dimension approach for variability management to runtime models [6], in the context of self-adaptive systems. The main idea is to use aspects at a model level, to adapt the running system, instead of hard-coding the adaptation logic at the platform level. Then, using a causal connection, modifications on the runtime model should be reflected on the running system. Moving models from design-time to runtime will reduce the complexity of runtime adaptations, by providing a higher level of abstraction. We are currently working on the implementation of the causal for the Fractal [20] component model. However, our causal link is not Fractal-specific and may be applied to other platforms like OpenCOM [9].

References

[1] V. Alves, P. M. Jr., L. Cole, P. Borba, and G. Ramalho. Extracting and Evolving Mobile Games Product Lines. In J. H. Obbink and K. Pohl, editors, *SPLC'05: 9th International Conference on Software Product Lines*, volume 3714, pages 70–81, Rennes, France, 2005.

[2] M. Anastasopoulos and D. Muthig. An evaluation of aspect-oriented programming as a product line implementation technology. In *ICSR'04: 8th International Conference on Software Reuse: Methods, Techniques and Tools*, pages 141–156, Madrid, Spain, 2004.

[3] S. Apel, T. Leich, and G. Saake. Aspectual Mixin Layers: Aspects and Features in Concert. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 122–131, New York, NY, USA, 2006. ACM Press.

[4] J. Araújo, J. Whittle, and D. K. Kim. Modeling and Composing Scenario-Based Requirements with Aspects. In *RE'04: Proceedings of the 12th IEEE International Conference on Requirements Engineering*, pages 58–67, Washington, DC, USA, 2004. IEEE Computer Society.

[5] E. Baniassad and S. Clarke. Theme: An Approach for Aspect-Oriented Analysis and Design. In *ICSE'04: Proceedings of the 26th International Conference on Software Engineering*, pages 158–167, Washington, DC, USA, 2004. IEEE Computer Society.

[6] N. Bencomo. Proceedings of the Models@run.time (at MoDELS) workshops. www.comp.lancs.ac.uk/ bencomo/MRT06/
www.comp.lancs.ac.uk/ bencomo/MRT07/.

[7] E. Brottier, B. Baudry, Y. L. Traon, D. Touzet, and B. Nicolas. Producing a Global Requirement Model from Multiple Requirement Specifications. In *EDOC'07: Proceedings of the 11th Enterprise Computing Conference*, Annapolis, Maryland, USA, 2007.

[8] T. Cottenier, A. van den Berg, and T. Elrad. Joinpoint Inference from Behavioral Specification to Implementation. *ECOOP'07: Proceedings of the 21st European Conference on Object-Oriented Programming*, 2007.

[9] G. Coulson, G. S. Blair, M. Clarke, and N. Parlavantzas. The Design of a Configurable and Reconfigurable Middleware Platform. *Distrib. Comput.*, 15(2):109–126, 2002.

[10] F. Fleurey, B. Baudry, R. France, and S. Ghosh. A Generic Approach For Automatic Model Composition. In *AOM@MoDELS'07: 11th International Workshop on Aspect-Oriented Modeling*, Nashville TN USA, Oct 2007.

[11] F. Heidenreich, J. Johannes, and S. Zschaler. Aspect-Orientation for Your Language of Choice. In *AOM@MoDELS'07: 11th International Workshop on Aspect-Oriented Modeling*, Nashville TN USA, Oct 2007.

[12] P. Jayaraman, J. Whittle, A. Elkhodary, and H. Gomaa. Model Composition in Product Lines and Feature Interaction Detection Using Critical Pair Analysis. In *MoDELS'07: Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems*, LNCS, pages 151–165, Nashville TN USA, Oct. 2007. Vanderbilt University, Springer-Verlag.

[13] C. Kästner, S. Apel, and D. Batory. A Case Study Implementing Features using AspectJ. In *SPLC'07: 11th International Software Product Line Conference*, September 2007.

[14] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An Overview of AspectJ. In *ECOOP'01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.

[15] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *ECOOP'97: Proceedings of the 11th European Conference on Object-Oriented Programming*, volume 1241, pages 220–242, Berlin, Heidelberg, and New York, 1997. Springer-Verlag.

[16] Y. Kim, M. Moon, and K. Yeom. An Aspect-Oriented Approach for Reprensenting Variability in Product Line Architecture. In *VaMoS'07: 1st International Workshop on Variability Modelling of Software-intensive Systems*, 2007.

[17] J. Klein, F. Fleurey, and J. Jézéquel. Weaving Multiple Aspects in Sequence Diagrams. *To appear in Transactions on Aspect-Oriented Software Development (TAOSD)*, 2007.

[18] P. Lahire, B. Morin, G. Vanwormhoudt, A. Gaignard, O. Barais, and J. M. Jézéquel. Introducing Variability into Aspect-Oriented Modeling Approaches. In *MoDELS'07: Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems*, LNCS, pages 498–513, Nashville TN USA, Oct. 2007. Vanderbilt University, Springer-Verlag.

[19] P. Lahire and L. Quintian. New Perspective To Improve Reusability in Object-Oriented Languages. *Journal Of Object Technology (JOT)*, 5(1):117–138, 2006.

[20] M. Leclercq, A. E. Ozcan, V. Quema, and J.-B. Stefani. Supporting Heterogeneous Architecture Descriptions in an Extensible Toolset. In *ICSE'07: Proceedings of the 29th International Conference on Software Engineering*, pages 209–219, Washington, DC, USA, 2007. IEEE Computer Society.

[21] N. Loughran and A. Rashid. Framed Aspects: Supporting Variability and Configurability for AOP. In *ICSR'04: 8th International Conference on Software Reuse: Methods, Techniques and Tools*, volume 3107 of *Lecture Notes in Computer Science*, pages 127–140, Madrid, Spain, 2004. Springer.

[22] N. Loughran, A. Sampaio, and A. Rashid. From Requirements Documents to Feature Models for Aspect Oriented Product Line Implementation. In *MoDELS Satellite Events*, pages 262–271, 2005.

[23] M. Mezini and K. Ostermann. Variability Management with Feature-Oriented Programming and Aspects. *SIGSOFT Software Engineering Notes*, 29(6):127–136, 2004.

[24] B. Morin, O. Barais, J. M. Jézéquel, and R. Ramos. Towards a Generic Aspect-Oriented Modeling Framework. In *3rd International ECOOP'07 Workshop on Models and Aspects - Handling Crosscutting Concerns in MDSD*, Berlin, Germany, August 2007.

[25] P. Muller, F. Fleurey, F. Fondement, M. Hassenforder, R. Schneckenburger, S. Gérard, and J. Jézéquel. Model-Driven Analysis and Synthesis of Concrete Syntax. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *MoDELS'06 : 9th International Conference on Model Driven Engineering Languages and Systems*, volume 4199 of *Lecture Notes in Computer Science*, pages 98–110, Genova, Italy, 2006. Springer.

[26] P. Muller, F. Fleurey, and J. M. Jézéquel. Weaving Executability into Object-Oriented Meta-languages. In *MoDELS'05: Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems*, volume 3713 of *Lecture Notes in Computer Science*, pages 264–278, Montego Bay, Jamaica, Oct 2005. Springer.

[27] R. Ramos, O. Barais, and J. M. Jézéquel. Matching Model Snippets. In *MoDELS'07: Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems*, LNCS, page 15, Nashville TN USA, Oct. 2007. Vanderbilt University, Springer-Verlag.

[28] A. Rashid, A. Moreira, and J. Araújo. Modularisation and Composition of Aspectual Requirements. In *AOSD'03: Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, pages 11–20, New York, NY, USA, 2003. ACM Press.

[29] Y. R. Reddy, S. Ghosh, R. B. France, G. Straw, J. M. Bieman, N. McEachen, E. Song, and G. Georg. Directives for Composing Aspect-Oriented Design Class Models. *Transactions on Aspect-Oriented Software Development I*, LNCS 3880:75–105, 2006.

[30] A. Schauerhuber, W. Schwinger, E. Kapsammer, W. Retschitzegger, and M. Wimmer. Towards a Common Reference Architecture for Aspect-Oriented Modeling. In *AOM'06@AOSD: 8th International Workshop on Aspect-Oriented Modeling at AOSD*, 2006.

[31] M. Sinnema, S. Deelstra, J. Nijhuis, and J. Bosch. COVAMOF: A Framework for Modeling Variability in Software Product Families. In R. L. Nord, editor, *SPLC'04: 3rd International Conference on Software Product Lines*, volume 3154 of *Lecture Notes in Computer Science*, pages 197–213, Boston, MA, USA, 2004. Springer.

[32] G. Straw, G. Georg, E. Song, S. Ghosh, R. B. France, and J. M. Bieman. Model Composition Directives. In T. Baar, A. Strohmeier, A. Moreira, and S. Mellor, editors, *UML'04: Proceedings of the 7th Conference on the Unified Modeling Language*, volume 3273 of *LNCS*, pages 84–97. Springer, Oct 2004.

[33] J. Van Gurp, J. Bosch, and M. Svahnberg. On the Notion of Variability in Software Product Lines. In *WICSA '01: Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, page 45, Washington, DC, USA, 2001. IEEE Computer Society.

[34] T. Ziadi and J. Jézéquel. *Families Research Book*, chapter Product Line Engineering with the UML: Products Derivation, pages 557–588. LNCS. Springer Verlag, 2006.