Managing Variability Complexity in Aspect Oriented Modelling

Brice Morin[1], Gilles Vanwormhoudt[2], Philippe Lahire[3], Alban Gaignard[3], Olivier Barais[1], and Jean-Marc Jézéquel[1]

[1] IRISA/INRIA/Rennes1, Equipe Triskell, F-35042 Rennes Cedex
[2] Institut Telecom / LIFL, Université de Lille 1, F-59655 Villeneuve d'Ascq Cedex
[3] I3S Nice-Sophia Antipolis, Equipe Rainbow, F-06903 Sophia-Antipolis Cedex

Abstract. Aspect-Oriented Modeling (AOM) approaches propose to model reusable aspects that can be composed in different systems at a model level. To improve the reusability, several contributions have pointed out the needs of variability in the AOM approaches. Nevertheless, the support of variability makes more complex the aspect design and the introduction of several dimensions of variability (advice, pointcut and weaving) creates a combinatorial explosion of variants and a risk of inconsistency in the aspect model. As the integration of an aspect model may be complex, it is essential that the AOM framework ensures the consistency of the resulting model. This paper presents an approach describing how to ensure that an aspect model with variability can be safely integrated into an existing model. The verifications include static checking of aspect models consistency and dynamic checking through testing with a focus on the parts of the model that are impacted by the aspect.

1 Introduction

Model-driven engineering (MDE) involves the development and evolution of complex models. To manage this complexity, models are usually decomposed in several smaller models. Different criteria can be considered for decomposition: a concern-driven decomposition (e.g., aspect-oriented modeling AOM), decomposition according to views (e.g., UML proposes several views to build a model), an object-oriented decomposition (where packages can provide a manageable unit for modeling), etc. Once a large model is decomposed in smaller models that are easier to manage, it is possible to work on these models. They can be discussed, refined, checked or simulated. To improve the reusability, several contributions, as our previous work called SmartAdapters [7], have pointed out the needs of variability support in the modeling approaches and introduce seamless variability mechanisms.

Nevertheless, if the support of variability improves the reusability, it makes in the same time the model design more complex. For example, in the context of aspect oriented modeling in which each model represents one view of the system, the introduction of several dimensions of variability into the advice, the pointcut and the weaving creates a combinatorial explosion of variants and a risk of inconsistency into the aspect model. As the interaction between an aspect model and the existing model may be itself complex, it is essential that the AOM provider ensures the consistency of the AOM model.

In this paper, we present how to ensure that an aspect model with variability can be safely integrated into an existing model. The verifications include static

verifications that check coherence properties on the aspect model and dynamic verifications through testing that focus on the parts of the model that are affected by the aspect. As a result of these verifications, aspect models can be provided as a commodity, such that a software architect can confidently apply an aspect model obtained from a third-party designer.

The remainder of this paper is organized as follows. In Section 2, we present a short overview of SmartAdapters, the AOM approach chosen as a basis of this work and we motivate our work through the presentation of samples of design errors linked to the variability support in an AOM approach. Section 3 presents an overview of the checking process performed on the aspect model in order to improve the designer confidence. Section 4 and 5 detail two main steps of this process: the static analysis and the testing phase. Section 6 illustrates how the design errors presented in section 2 are detected thanks to this process. Finally, Section 7 presents related works and Section 8 concludes and points out future works.

2 Background and motivating examples

SmartAdapters is an AOM approach that has formerly been applied to the domain of Java programs [8] and UML class diagrams. More recently, it has been generalized to any domain metamodel [11]. In the remainder of this paper, we will focus on class diagrams but the ideas described in this paper may be generalized to any domain.

In the SmartAdapters approach, an aspect is composed of three parts: *i)* a graft model, representing *what* we want to weave, *ii)* an interface model, representing *where* we want to weave the aspect and *iii)* a composition protocol specifying *how* to weave the graft model into the interface model. The graft model is a model fragment representing a given concern. The interface model is a model fragment parameterized by roles allowing the interface model to be matched in different base models [15]. Finally, the composition protocol is described by model transformation primitives that manipulate elements from the graft and the interface models.

In [7] we extended SmartAdapters with the addition of variability ingredients to both composition protocol (*how*) and interface model (*where*). We have adopted concepts introduced by Software Product Lines (SPL) paradigms [22] such as alternatives and variants, options, and constraints. An *alternative* in the composition protocol indicates that there exist several possible ways to compose a part of the graft model into the interface model. An *option*, either in the composition protocol or in the interface model, indicates non-mandatory elements. In the protocol, an optional adaptation might be realized or not. In the interface model optional elements may be bound or not at composition time. The resulting aspect family (*i.e.*, all the aspects we can obtain by derivation), may not be consistent. We propose to add *constraints* in the protocol in order to maintain its integrity.

To illustrate the SmartAdapters approach with variability, we introduce an aspect that aims at allocating tasks to workers, according to their skills and working load. Figure 1 depicts this aspect. Its graft model contains a *Scheduler* class that manages workers described by the *Worker* class and allocates tasks

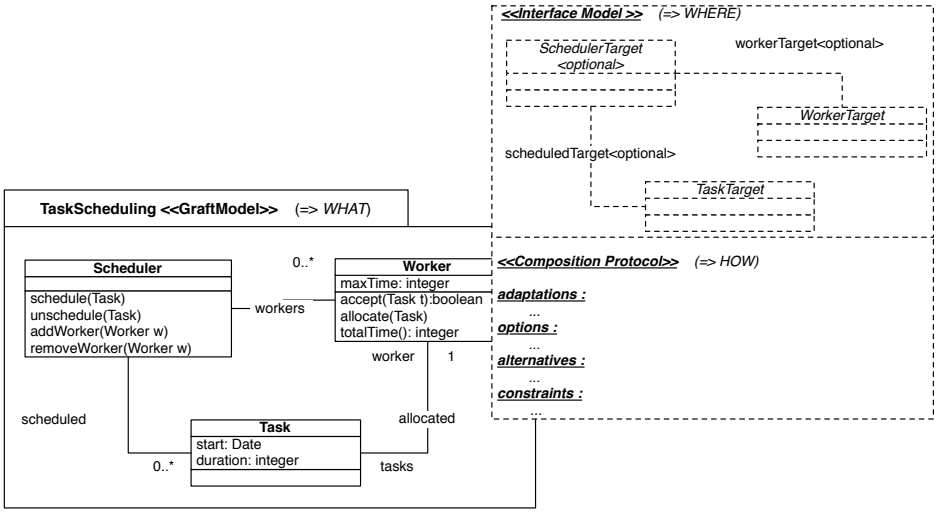

Fig. 1. The task allocation aspect

described by the *Task* class. Its interface model, illustrated in top right part of Figure 1, declares that three classes related by associations, some being optional, must be present into a base model to apply this aspect. The composition protocol that describes the different ways to compose this aspect is illustrated in figure 2.

This protocol proposes to integrate elements (classes, associations, ...) of the task allocation aspect into a base model either by merging, by inheritance or by insertion. To this end, it introduces several alternatives. The first alternative called *TaskWorkInsertion* deals with the integration of features provided by classes *Worker* and *Task*. This alternative contains two variants corresponding respectively to the choice between inheriting from these classes or merging their features into *WorkerTarget* and *TaskTarget* classes. A variant may contain several adaptations and options. Implicitly this means that these elements are dependent from each others. In the example, *Worker* and *Task* classes as well as their associations are integrated together in a consistent way.

The second alternative called *SchedulerInsertion* is related to the integration of the *Scheduler* class. This alternative consists in choosing between merging its features to an existing class of the base model or inserting the class as new. For this second variant, remind that the class is declared optional in the interface model and therefore does not require a binding.

Next, the composition protocol addresses the two associations between the *Scheduler*, the *Worker* and *Task* classes. To manage these associations, the composition protocol declares an alternative with three variants called *VIntroduction-TWMerge*, *VIntroductionTWInheritance* and *VMappingTWMerge*. The first two alternatives deal with the case where the associations do not exist in the base model and must be inserted. In this case, two variants are proposed to consider the ways to insert both *Worker* and *Task* classes : the first variant corresponds to the inheritance version whereas the second variant deals with the merging one. As we can see through this example, an alternative may depend on the choice made

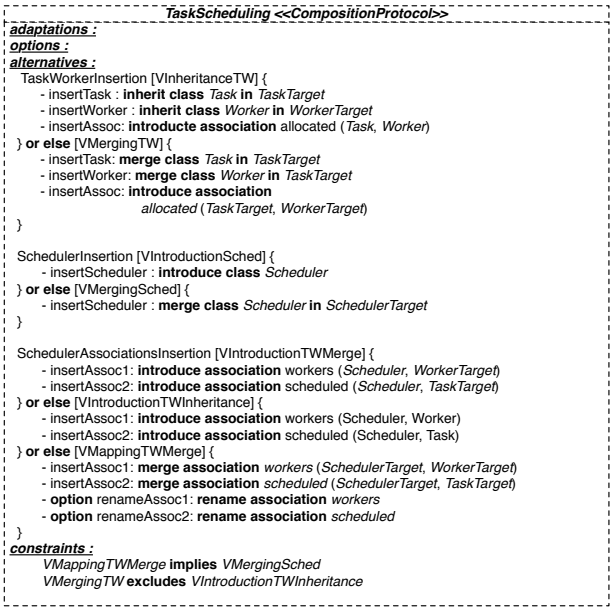

Fig. 2. Composition protocol of the Task Allocation aspect

for another one. The third variant assumes that both associations exist in the base model and provide the adaptation to map them. This variant also gives the capacity to rename each association. Each renaming adaptation (*renameAssoc1*, *renameAssoc2*) is an example of optional adaptation included in a variant.

The choice of the variants and options is realized during the derivation step that produces an aspect with no variability, depending on the choices of the user. For the previous example, we may decide to select variants in order to insert the Scheduler class in combination with the inheritance of Task and Worker and the introduction of association between Scheduler, Task and Worker. After derivation, the resulting aspect can be composed with a base model by specifying a binding that associates each interface model elements to a model element of the base model. This binding implicitly contextualizes the primitives planned in the composition protocol with concrete elements.

The table given in figure 3 summarizes all possible combinations of variants for this previous aspect and indicates if the corresponding combination is consistent or not. An inconsistent combination means that the adaptations resulting from the selected variants are incompatible. There may be several reasons why variants are incompatible: *i)* a variant may depend of elements introduced by other variants not selected in the derivation, *ii)* a variant may introduce elements that prevent other selected variant to be applied and, *iii)* combined variants may break the conformance with the metamodel or invalidate user-level constraints. In case of the previous aspect, we may identify two main cases of incompatibility:

Incompatibility between VIntroScheduler and VMappingTWMerge:
The insertion of the *Scheduler* class into the base model excludes the mapping of the associations between *Scheduler*, *Task* and *Worker* with any ex-

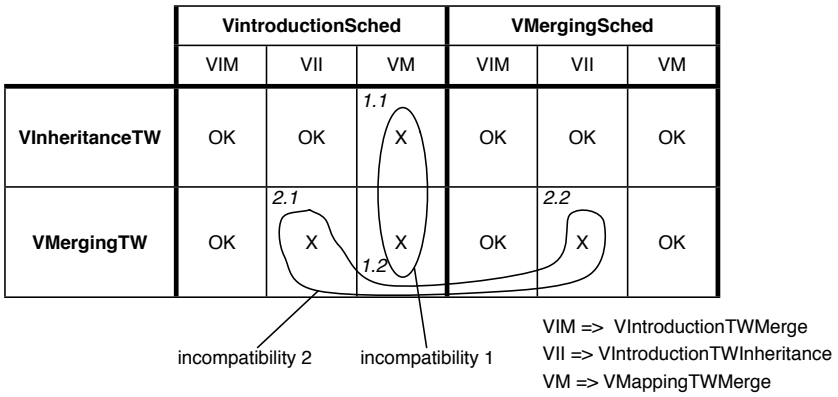

Fig. 3. All variants combination of the TaskScheduling aspect

isting ones. This incompatibility exists whatever if these last two classes are introduced by inheritance or by merging.

Incompatibility betwneen VMergingTW and VIntroTWInheritance: The merging of *Task* and *Worker* with existing classes of the base model is incompatible with the introduction of the associations dedicated to the inheritance version.

In order to deal with these incompatibilities, the previous protocol defines one dependency constraint and one mutual exclusion constraint. These constraints restrict the number of possible combinations of variant to consistent ones. The dependency constraint between *VMappingTWMerge* and *VMergingSched* ensures that the variant to map existing associations between *Scheduler*, *Task* and *Worker* classes can only be selected if the first class is integrated by merging, e.g the *VMergingSched* variant. The mutual exclusion constraint between *VMergingTW* and *VIntroductionTWInheritance* implies that the merging of *Task* and *Worker* is incompatible with the introduction of the association dedicated to the inheritance version, i.e the *VIntroductionTWInheritance* alternative.

Constraint is an essential construct to ensure the consistency of a family and increase its reliability. However, defining these constraints manually might be time-consuming and not optimal as it requires to manually derive the whole family. The constraints defined by a user might be over (resp. under)-specified leading to a reduced (resp. inconsistent) aspect family.

This need to test all possible configurations (as illustrated by the table of our example) hinders scalability and puts great limitation on the use of variability into AOM approaches. We think that the aspect designer should be alleviated as much as possible from these repetitive operations and should benefit from an automatic support that generates and tests all configurations without designer intervention. Such support should provide feedback at design time about the configurations of variants that cause conflicts so that the aspect designer is able to ensure the consistency of its protocol by introducing all necessary constraints. In the following, we present our solution to provide this automatic support in the context of our AOM approach.

3 Checking aspect families consistency

We propose a systematic approach for analysing and assessing the consistency of aspect models with variability [7, 11]. Our approach generates, analyzes and tests all the possible variants of the aspect family. The complexity of managing all the members of a product line may be time and resource consuming in the case of real SPL. However, an aspect model can be seen as a micro software product line focusing on a given concern and does not aim at representing a whole application. Thus, the number of possible variants for each variable aspect model is reasonable.

Our proposed process, shown in figure 4, starts with a variable aspect. It is composed of three sequential steps which are performed for all possible variants of the variable aspect, that is to say the aspect family:

Step 1: Producing a variant of the aspect family: This step produces an aspect model without variability corresponding to a test case for the analysis. The produced aspect model is computed by selecting a configuration of options and variants among the ones defined by the variable aspect: its composition protocol (resp. interface model) only contains elements defined (resp. used) by the selected variants and options. For instance, applying this step in the context of our example consists in constructing an aspect model corresponding to one cell of the table shown in figure 3.

Step 2: Static analysis of produced aspect: This step takes the produced aspect model as input and performs a static analysis that checks various properties of consistency on each part of the aspect (graft model, interface Model, composition protocol). If the aspect is detected to be inconsistent, the process stops and the following step is not performed.

Step 3: Testing aspect composition: In this step, the produced aspect is tested for validation by applying its composition protocol to a generated base model. This base model is automatically built from the content of the interface model to ensure that they respect all its structural constraints. The binding of the aspect model is computed to match each interface model element to the corresponding base model element. If the composition raised an error, we are in the case of a variant that is inconsistent.

At the end of the process, a report is generated that specifies all variant configurations that have been tested. For the ones that have failed, explanations about which checking step (step 2 or 3) fails and a detailed diagnostic is provided. This report allows the aspect designer to identify and understand precisely the issues and fix them by changing or extending parts of the aspect model. After the changes, the effects on the aspect model can be tested again by re-running the previous process. The analysis process can be repeated until the variable aspect model meets its reliability requirements.

In the following, we focus on the last two steps of the above process.

4 Aspect model static analysis

We address *i)* the consistency of the models involved in the description of the aspect model (Section 4.1), *ii)* the consistency of the composition protocol (Sec-

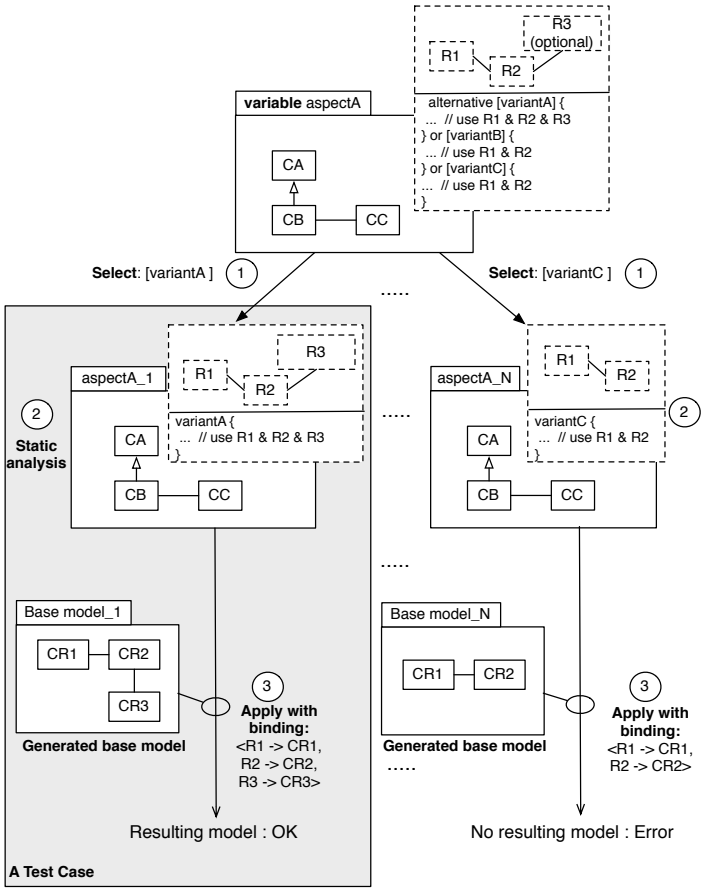

Fig. 4. Overall analysis process

tion 4.3). To achieve the latter we propose in Section 4.2 an approach allowing an accurate description of the composition protocol semantics.

4.1 Well Formedness Rules of an aspect model

Given an aspect model, SmartAdapters first checks that its various elements are syntactically correct and respect the Well-Formedness Rules.

Graft model and Interface model. Both the graft model and the interface model are designed using the concepts defined in our domain metamodel which is very similar to MOF (in fact we use EMF/ECore[4] which is dedicated to the design of class diagrams). Unlike base models, graft and interface only represent model fragments. However, we statically check that model elements are well-typed *e.g.*, all association ends are correctly typed and an object can not be contained by two model elements.

[4] See *http://www.eclipse.org/modeling/emf/*.

Adaptations. Several features of the composition language have been designed to prevent the designer from expressing unsafe composition. For example, the association in SmartAdapters is based on the linking of two references (*Ereference* in the ECore metamodel) in two classes. When the designer specifies that he wants to set one of the references, SmartAdapters modifies the related association end automatically, relieving the designer of the burden of setting the opposite reference of an association, and then reducing the size of the transformation specification. The SmartAdapters model also requires that each model element is linked thru a composition relationship (i.e. the containment property is set in EMF) to at most one model element. The composition protocol enforces this constraint by automatically removing the element of the containment property in the container when the designer changes the container of a model element.

4.2 Adapter Specification Language semantics

To perform this static analysis, we propose to express our aspect models with a lower level composition language that has a clear operational semantics. We distinguish two types of model composition: basic primitives and merging. We propose to define the semantic of these composition operators in the remainder of this section. This composition language allows us to reason on an aspect before actually weaving it into a particular base model.

An action language for manipulating models Basic composition primitives are simple operations on model elements and can easily be implemented with an action language such as Kermeta [12]. The language proposes basic operators on model elements (create, update, delete) and properties (get, set, add, remove). It supports associations and provides loop and condition statements. A full description of the semantics of this language is provided in [19].

The graft model, the interface model and the composition protocol are described with the primitives of this action language. Figure 5 shows an excerpt of the graft model of the task scheduling aspect defined with this action language features. *EClass* and *Ereference* are two meta-classes of ECore allowing to define a class and an object reference. *eReferences* is a property of the meta-class *EClass* which contains all the object references of one class. Lines *01* and *02* show the creation of the classes *Task* and *Worker* and lines *03* and *04* describe the association-end *allocated* connecting these two classes.

```
....
01  CREATE {Eclass,Worker}: creation of class Worker
02 CREATE {Eclass,Task}: creation of class Task
03 CREATE{Ereference,allocated}: creation of reference "allocated" between classes Worker and Task
04 ADD_REF{eReferences,Worker,allocated}: association of reference "allocated" to property eReferences of class Worker
...
```

Fig. 5. extract of the graft model description with our language [19]

All the basic adaptations provided by SmartAdapters (*e.g*, adding a super class, adding a parameter in an operation, etc) can be expressed using these low level primitives. For example, the adaptation for adding a super class A to a class B[5] is expressed with our action language by: ADD_REF$_{eSuperTypes,A,B}$ where A is the class to be added as parent and B the class to which A should be added as parent. *eSuperTypes* is the property contained in the metaclass "Eclass" that allows to record the super types of a class.

High level model composition operator The composition of model elements can be also performed through a merging operator. Merge operator is a symmetric 1 to 1 composition operator (it involves exactly two input models). It is an operator that combines two model elements representing different views on the same concept (in both input models) into a new (or into an updated version of a) model element that represents the merge of those views.

SmartAdapters Models composition involves the unification of common elements in the SmartAdapters Models (SAM) being merged. The composition of two SAMs is based on a merge operator with a parametric semantics proposed in previous work [1]. The characteristic of this merge operator is the fact that it provides a default merge behavior when "no conflicts" exist between SAMs to be merged. The default behavior defines directives which include: "the members of two classes with the same identifier should be unified into one composite class with this identifier". The conflict detection is based on the following mechanism. First, two elements that have the same identifier are detected. When two model elements have the same identifier, they become possible candidates for being merged. Next, the signatures of these elements are compared in order to check if there is a strict equivalence between them. When the signatures of two matching elements are not equal, a conflict is detected. In [1], we define the signatures for each model elements and all possible conflicts. The designer can extend the merge semantics in registering plugin (conflict fixer) in order to customize the semantics of *merge*.

4.3 Composition protocol analysis

The language described in previous section allows us to benefit from *i)* a description of the model element creations (See figure 5) and *ii)* a clear semantics of the adaptations. We first propose to analyze each composition protocol *a priori*, before actually weaving the aspect into any model. Each composition protocol is an ordered sequence of adaptations. An example of adaptation described with our action language is presented in section 4.2 for one type of adaptation. We propose to raise some errors when the composition protocol may produce inconsistencies in the woven model. In order to ease the analysis of composition protocols, we consider *canonical* composition protocols.

Definition 1 (Canonical composition protocol). *A canonical composition protocol is composed of adaptations that all actually have an effect. It does not*

[5] A and B are instances of the metaclass *Eclass* and had been already created as it has been shown in Figure 5.

contain antagonist adaptations (e.g., adding/deleting the same element) nor neutral adaptations (e.g., adding several times the same element in a set).

A non canonical protocol will be considered as an error. A report specifying which adaptations are antagonists is generated in order to help the designer in refactoring his aspect.

Definition 2 (Impact). *The impact of a canonical composition protocol on a property p of an object obj is a value* $\Delta_{p,obj} \in \mathbb{Z}$ *computed as follows:*
$\Delta_{p,obj} = \#ADD\_^*_{p,anyInstance,obj}$ - $\#DEL\_^*_{p,anyInstance,obj}$
Less formally, it quantifies the increase or decrease of the size of the property.

Based on the above definitions, we can perform a static analysis of composition protocol, independently from any base model where it may be applied. We define and formalize a set of rules:

- **1)** $\exists$ CREATE$_{anyMeta,instance} \wedge \exists$ ADD_REF$_{anyPrty,instance,anyTarget}$
  $\Rightarrow \exists$ ADD_CONTAIN$_{anyPrty,instance,anyTarget}$[6]
  This rule detect the dangling references that refer to elements that are not present into the resulting model.
- **2)** $\forall$ obj : Object, p : Property $\in$ obj
  $\Delta_{p,obj} \leq$ p.upper - p.lower
  This rule detects if the composition of the aspect respects the cardinality defined in the metamodel.
- **3)** $\exists$ DEL_CONTAIN$_{p,obj,any} \Rightarrow \neg\exists$ obj',p' | obj $\in$ obj'.p'
  This rule raises a warning specifying that a model element is removed from the resulting model and may lead to some dangling references.

Rules **1** and **2** allows detecting some inconsistencies before actually weaving the aspect into a particular base model. We now propose to use the knowledge provided by the base model to describe some model-specific rules that are checked before weaving. In the following rules, we denote *base(obj)* the base model element associated to the interface model element *obj* and *base(obj).p.size* the actual size of the property *p* of the base model element bound to *obj*.

- **4)** $\forall$ obj : Object, p : Property $\in$ obj
  base(obj).p.size - p.lower $\leq \Delta_{p,obj} \leq$ p.upper - base(obj).p.size
  This rule refines rule **2** with the knowledge provided by the base model and verifies that the cardinality defined in the metamodel will be respected by the resulting model.
- **5)** $\exists$ DEL_CONTAIN$_{p,obj,any} \Rightarrow \neg\exists$ obj',p' | base(obj) $\in$ obj'.p'
  This rule refines rule **2** with the knowledge provided by the base model and verifies that no dangling references will be produced after weaving.

5 Validating an aspect model through testing

5.1 Overview

The static analysis step can detect some errors but others are not detected through the static analysis due to the lack of information in the interface model. The last

[6] ADD/DEL_CONTAIN [19] allow to set/unset containment property mentionned in section 4.

step of the process is a testing approach. In this case, we consider the aspect model with variability as a system to check. Each derived aspect model (see section 3) can be seen as a test case for the system. The oracle is composed of two kinds of assertion:

1. Could we perform the composition? If the composition raised an error before the end, we are in the case of an inconsistent aspect model. It generally means that some constraints are missing to correctly handle the variation points of the aspect model.
2. Does the resulting model conform to its meta-model and respect its well-formedness rules ?

The last point in the testing process consists in obtaining the test data. From the interface model, we generate a set of base models. The derived aspect model will be woven into each of the generated base models.

For each kind of error, the SmartAdapters framework gives a diagnostic that explains why the test case fails. In the first case, it gives the diagnostic to explain which adaptation cannot be executed and why ? In the second case, it gives which global invariant is not respected. The designer should then analyze precisely the generated variant and the generated base model to understand the issue.

5.2 Generating a set of basis model

We automatically generate a set of base models in order to validate composition protocols. Generating base models from scratch may be resource and time-consuming. But, we are only interested in base models where the derived aspect model may be applied, *i.e.* where the interface model matches. In other words, models that do not integrate the interface model are not interesting for validating our composition protocols. We propose to use the approach of Sen *et al.* [18, 17] to produce a set of base models that completes the interface model in order to respect all the explicit and implicit constraints defined in the metamodel.

The base models that are used as test data for model composition are mainly constrained by the interface model that defines a set of constraints on the attending base model. Thus, in order to automatically generate base models for testing it is necessary to interpret both structural information and invariants defined in the interface model (see [15]) into a consistent set of information that can be used for model synthesis. In our approach, we use the Cartier tools [17] based on the first-order relational logic language Alloy to transform interface models into constraints. Once solved, these constraints lead to a selection of qualified base models from the input domain of the model composition.

For each composition protocol, we generate a report that specifies the percentage of woven models meeting the oracle criteria defined in Section 5.1. This report can be used by the designer to validate and invalidate some variants of the composition protocol. Note that this report does not prove that the composition is totally safe.

6 Assessments

In section 2 we introduce a task scheduling aspect that contains variability. In Figure 3, we identified four cases where the derived aspects does not make sense. In this section, we give a quick look on how these cases could be automatically identified by our verification process. Due to a lack of place, we detail only two of the four problematic cases identified in Figure 3: Entry 2.1 and Entry 2.2.

6.1 VMergingTW x VIntroductionSched x VIntroductionTWInheritance

The derived aspect corresponding to this case is illustrated in Figure 3 (Entry 2.1). The *Task* and *Worker* classes are respectively introduced by merging into the *TaskTarget* and *WorkerTarget* classes. An association is introduced between those two merged classes. Note that our merge modifies the existing classes[7], so the association is properly introduced. The *Scheduler* class is then introduced. Whether the *SchedulerBase* class is present or not in the interface model, the *Scheduler* class can be introduced. The introduction of the associations *worker* and *scheduled* is problematic. Indeed, only one extremity class of each association has been introduced. In other words, we refer to elements outside of the resulting model. This problem is detected by two violations of the rule **1**:

- $\exists$ CREATE$_{class,Worker} \wedge \exists$ ADD_REF$_{referenceType,worker,Worker}$
  $\Rightarrow \exists$ ADD_CONTAIN$_{_,_,Worker}$
  The *Worker* class will not be introduced into the base model but the *worker* association will refer to it.
- $\exists$ CREATE$_{class,Task} \wedge \exists$ ADD_REF$_{referenceType,scheduled,Task}$
  $\Rightarrow \exists$ ADD_CONTAIN$_{_,_,Task}$
  The *Task* class will not be introduced into the base model but the *worker* association will refer to it.

6.2 VMergingTW x VMergingSched x VIntroductionTWInheritance

The derived aspect associated to this combination of alternative (Entry 2.2 in Figure 3) leads to the violation of the same rules. Moreover, the two extremities of both associations (*worker* and *scheduled*) are not present in the resulting model because the *Scheduler* class is introduced by merging it into the *SchedulerTarget* class. Thus, it cannot be referred to.

6.3 Discussion

We have shown in this section how the static analysis of the derived aspects can determine before weaving two problematic cases. The detection of these problems early in the design of an aspect model allows the designer to identify that it misses constraints to forbid these configurations during the derivation. The three kinds of checking: WFR (section 4.1), static analysis through partial evaluation (sections 4.2-4.3), and test (section 5) create a modular verification process allowing the designer to incrementally validate his aspect model. For example, if an error is detecting by static analysis, it is not necessary to perform the testing phase.

[7] "merge a into b" modifies b with features from a

7 Related work and discussion

Ensuring software correctness is an important issue and this is amplified in the case of software product lines [14, 20]. One of the main concern for correctness in product lines is about the methods to be used in order to limit the number of tests to be performed for a family of products. Two issues are more especially considered: the increase of work for the programmer and the time spent to perform them [3, 9]. These contributions mainly introduce formal methods in order to exploit the commonalities of software family in order to achieve these issues. They rely for example on SAT solver [4] or more generally on model-checking [6] techniques in order to verify those tests.

Because the domain engineering is handled in the early phases of the software development process, approaches rely for test modeling - for example - on use-cases [13], activity diagrams [16], feature models [4, 10, 16, 21], OCL [4] or on a combination of these formalisms. It is interesting to mention that application engineering deals more with source code testing and that tests related to domain engineering can be addressed also later on during product line evolution.

When the product line is built from legacy systems, the main idea is to generalize tests made for existing applications [5]. On the contrary, when the product line is built first approaches favor the design of generic tests [6, 16] allowing to produce tests for the applications which are derived from the product line.

It is worth to note that the "product" which is foreseen in our approach is the composition protocol associated to a graft model. The complexity which is expected for such a product line has nothing to see with the one addressed by [3, 9] so that, trying to reduce the number of tests to be performed is not an issue. Moreover the operations supported by our product line correspond to a small set of adaptations whose semantics is precisely defined (see Section 4.2) so that most of the tests could be generated. Because a composition protocol relies on a business model, our approach deals with model but not source-code testing.

When dealing with tests relating to the domain engineering phase, some of the interesting issues are dealing with the kind of tests to be performed. Possible ones are : *i)* to check that the derived products do not break any rules of the metamodel that is used to describe the product family [4], *ii)* to ensure that the product derivation ensures the variability constraints [2, 16], *iii)* to control that there is no more functionality than those used by the derived applications [16] and, *iv)* to check the consistency of the product line that is designed.

Most approaches are dealing with testing the conformance of applications with a product line but very few consider the issue of building a consistent product line (point *iv*). Our approach intends to provide an homogeneous process to guide and control the derivation process even if our contribution mainly focuses on the point *iv*. According to this last issue, we should mention contributions such as [4] which relies on OCL constraint specification or [10] which provides advanced tests. The tests proposed by [10] deal with *i)* realizability (are there non realizable product-line members?); *ii)* internal competition (are there identical feature combination?), *iii)* commonality between realizable product line members and *vi)* dead variants or features in the derived product. Like our approach, these approaches are not dedicated to a specific product line.

8 Conclusion and future work

Our approach SmartAdapters promotes the design of reusable business models that can be woven according to the needs of application architects. In our approach these reusable models are composed of: a graft model (the business description), an interface model (the required entities from base models) and a composition protocol (how to compose the graft model into the interface model) independent from the base model. Because there exists a wide range of base models for a given domain, the way we compose our reusable models should really be flexible in order to fit to the various contexts of reuse. To achieve this issue we designed our composition protocol as a product line [7] but its underlying variability mechanisms create a combinatorial explosion of variants and a risk of inconsistency in the aspect model.

Consequently, it is necessary to provide not only the aspect description but also the verification process that comes with it: like software engineering, model engineering requires validation. To ensure the consistency of the models obtained after weaving an aspect, it is mandatory to ensure the consistency of the aspect model, that is to say to perform a deep analysis on *i)* the set of derived composition protocols and *ii)* the involved models (graft and interface). This work has proposed a checking process that achieves accurate analysis and consistency tests of aspect models with variability. It relies on a clear semantic of both the domain and the composition protocol thanks to an action language and generative features. This checking process aims at ensuring that the aspect designer did not forget any of the constraints driving its variability.

In future works, we plan to extend our approach in order to provide an integrated environment dedicated to the control and the guidance of the user during the derivation process. We also plan to add the capability to attach domain constraints to an aspect model that will act as post-conditions of the model compositions. Our approach is not currently supporting the detection of faults in the pointcut definition but we intend to show that domain constraints contribute to address this issue. Lastly, we plan to study more sophisticated strategies for the generation of base models in order to get tests which validate an aspect model on contexts closer to what may be found in industrial case-studies.

References

1. O. Barais, J. Klein, B. Baudry, A. Jackson, and S. Clarke. Composing Multi-View Aspect Models. In *7th IEEE International Conference on Composition-Based Software Systems (ICCBSS)*, Madrid, Spain, feb 2008.
2. A. Bertolino, A. Fantechi, S. Gnesi, and G. Lami. Product line use cases: Scenario-based specification and testing of requirements. In *Software Product Lines*, pages 425–445. Springer, 2006.
3. M.B. Cohen, M.B. Dwyer, and J. Shi. Coverage and adequacy in software product line testing. In *ROSATEA '06: Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis*, pages 53–63, New York, NY, USA, 2006. ACM.
4. K. Czarnecki and K. Pietroszek. verifying feature-based model templates against well-fformedness ocl constraints. In *GPCE '06: Proceedings of the 6th international conference on Generative programming and component engineering*, pages 211–220, New York, NY, USA, 2006. ACM.
5. B. Geppert, J.J. Li, F. Rößler, and D.M. Weiss. Towards Generating Acceptance Tests for Product Lines. In *Software Reuse: Methods, Techniques and Tools: 8th*

International Conference, ICSR 2004, volume 3107 of *Lecture Notes in Computer Science*, pages 35–48. Springer, 2004.
6. T. Kishi, N. Noda, and T. Katayama. Design verification for product line development. In J.H. Obbink and K. Pohl, editors, *SPLC*, volume 3714 of *Lecture Notes in Computer Science*, pages 150–161. Springer, 2005.
7. Ph. Lahire, B. Morin, G. Vanwormhoudt, A. Gaignard, O. Barais, and J. M. Jézéquel. Introducing Variability into Aspect-Oriented Modeling Approaches. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *MoDELS'07: 10th Int. Conf. on Model Driven Engineering Languages and Systems*, Lecture Notes in Computer Science, Nashville TN USA, Oct 2007. Springer.
8. Ph. Lahire and L. Quintian. New Perspective To Improve Reusability in Object-Oriented Languages. *Journal Of Object Technology (JOT)*, 5(1):117–138, 2006.
9. M. Mannion and J. Cmara. Theorem proving for product line model verification. In Frank van der Linden, editor, *PFE*, volume 3014 of *Lecture Notes in Computer Science*, pages 211–224. Springer, 2003.
10. A. Metzger, P. Heymans, K. Pohl, P.-Y. Schobbens, and G. Saval. Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. In *RE'07: 15th Int. Conf. on Requirements Engineering*, pages 243–253, Delhi, october 2007.
11. B. Morin, O. Barais, and J. M. Jézéquel. Weaving Aspect Configurations for Managing System Variability. In *VaMoS'08: 2nd International Workshop on Variability Modelling of Software-intensive Systems*, Essen, Germany, 2008.
12. P.A. Muller, F. Fleurey, and J.M. Jézéquel. Weaving Executability into Object-Oriented Meta-languages. In *MODELS/UML'05:*, volume LNCS 3713, Springer-Verlag, October 2005. Kermeta is available at: http://www.kermeta.org/.
13. C. Nebut, F. Fleurey, Y. Le Traon, and J.M. Jézéquel. A Requirement-Based Approach to Test Product Families. In *PFE*, pages 198–210, 2003.
14. K. Pohl and A. Metzger. Software product line testing. *Commun. ACM*, 49(12):78–81, 2006.
15. R. Ramos, O. Barais, and J. M. Jézéquel. Matching Model Snippets. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *MoDELS'07: 10th Int. Conf. on Model Driven Engineering Languages and Systems*, Lecture Notes in Computer Science, Nashville TN USA, Oct 2007. Springer.
16. A. Reuys, S. Reis, E. Kamsties, and K. Pohl. The scented method for testing software product lines. In *Software Product Lines*, pages 479–520. Springer, 2006.
17. S. Sen, B. Baudry, and J.-M. Mottu. On Combining Multi-formalism Knowledge to Select Models for Model Transformation Testing. In *ICST'08: 1st Int. Conf. on Software Testing Verification and Validation*, Lillehamer, Norway, Apr 2008.
18. S. Sen, B. Baudry, and D. Precup. Partial Model Completion in Model Driven Engineering using Constraint Logic Programming. In *Int. Conf. on Applications of Declarative Programming and Knowledge Management*, Wurzburg, Germany, Oct 2007.
19. Smartadapters action language semantics, 2008. Available at http://www.irisa.fr/triskell/perso_pro/obarais/pmwiki.php?n=Research.SmartAdapters.
20. M. Svahnberg and J. Bosch. Issues concerning variability in software product lines. *Lecture Notes in Computer Science*, 1951:146–157, 2001.
21. S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe composition of product lines. In *GPCE'07: 6th Int. Conf. on Generative programming and component engineering*, pages 95–104, New York, NY, USA, 2007. ACM.
22. T. Ziadi and J.M. Jézéquel. *Families Research Book*, chapter Product Line Engineering with the UML: Products Derivation, pages 557–588. LNCS. Springer Verlag, 2006.