Preliminary Proceedings

of the Second International Workshop on

Aspect-Based and Model-Based Separation of Concerns in Software Systems (ABMB 2006)

http://www.open.ou.nl/ABMB/

the t0th of July 2006, Bilbao, Spain

In collaboration with
the European Conference on Model Driven Architecture - Foundations and Applications

The post workshop proceedings will be published as a volume of the ENTCS

Aspect-Based and Model-Based Separation of Concerns in Software Systems (ABMB 2006)

Organizers:
Mehmet Akşit, University of Twente, the Netherlands M.Aksit@ewi.utwente.nl
Tzilla Elrad , Illinois Institute of Technology, USA Elrad@iit.edu
Ella Roubtsova, Open University of the Netherlands Ella.Roubtsova@ou.nl

Programme committe:

Mehmet Akşit, University of Twente, the Netherlands M.Aksit@ewi.utwente.nl
Pierre Cointe , Ecole des Mines, France Pierre.Cointe@emn.fr
Siobhán Clarke, Trinity College, Dublin, Ireland siobhan.clarke@cs.tcd.ie
Tzilla Elrad , Illinois Institute of Technology, USA Elrad@iit.edu
Jeff Gray, University of Alabama at Birmingham, USA gray@cis.uab.edu
Reiko Heckel, University of Leicester, UK reiko@mcs.le.ac.uk
Ruurd Kuiper, TU Eindhoven, The Netherlands wsinruur@win.tue.nl
Tommi Mikkonen, Tampere University of Technology, Finland tjm@cs.tut.fi
Awais Rashid, Lancaster University Lancaster UK marash@comp.lancs.ac.uk
Ella Roubtsova, Open University, the Netherlands Ella.Roubtsova@ou.nl
Dominik Stein, University of Essen, Germany dstein@cs.uni-essen.de
Gerd Wagner, Brandenburg University of Technology at Cottbus, Germany. G.Wagner@tu-cottbus.de
Takuo Watanabe, Tokyo Institute of Technology, Tokyo, Japan. takuo@acm.org

Aspect-Based and Model-Based Separation of Concerns in Software Systems (ABMB 2006)

Table of contents

1. ***Using aspect-orientation techniques to improve the reuse of metamodels***
A. M. Reina Quintero,J. Torres Valderrama,
Department of Languages and Computer Systems; University of Seville,Seville, Spain

2. ***Concern-Specific Languages in a Visual Web Service Creation Environment***
Mathieu Braem, Niels Joncheere,Wim Vanderperren,Ragnhild Van Der Straeten Viviane Jonckers,
System and Software Engineering Lab; Vrije Universiteit Brussel.

3. ***Model Driven Development of Security Aspects.***
Julia Reznik, Tom Ritter,
Froundhofer Institute FOKUS, Berlin, Germany,
Rudolf Schreiner, Ulrich Lan,
ObjectSecurity Ltd., Cambridge, UK

4. ***On the Dominance of Decompositions in Models and their Aspect-Oriented Implementations***.
Tommi Mikkonen,
Institute of Software Systems; Tampere University of Technology; Tampere, Finland

5**. *Impact of evolution of concerns in the Model Driven Architecture Design Approach.***
Bedir Tekinerdogan, Mehmet Aksit, Francis Henninger.
Department of Computer Science; University of Twente.

Using aspect-orientation techniques to improve the reuse of metamodels ⋆

A. M. Reina Quintero [1] J. Torres Valderrama [2]

Department of Languages and Computer Systems
University of Seville
Seville, Spain

Abstract

Metamodelling is an activity that is getting much more attention by the research community. One of the main causes of this recent interest is being an important piece for the different approaches for Model-Driven Development (MDD). In this context, the definition and implementation of metamodels not only is important for improving the MDD, but also for their later reuse.

Obliviousness is an important property to maintain when metamodels are reused. That is, a metamodel should be unaware of being extended by another metamodel. This paper shows that current techniques for implementing metamodels don't maintain this obliviousness when some elements of the extended metamodel are related by means of associations with the elements of the original metamodel. Thus, in order to improve the reuse, three different approaches (one using traditional object-oriented techniques, and the others using aspect-oriented techniques) are analyzed. As a result, we think that the third approach, which considers relationships as first-class citizens at the implementation level by using relationship aspects is the best one.

Key words: Metamodelling, Aspect-Oriented Programming, Model-Driven Architecture.

1 Introduction

Nowadays there is an increasing interest in Model Driven Development (MDD). This interest is being encouraged due to the apparition of new and more powerful tools that let us face up to the whole software development process.

⋆ This work has been partially supported by the Spanish Ministry of Science and Technology and FEDER funds: TIC-2003-369.

[1] Email:mailto:reinaqu@lsi.us.es
[2] Email:mailto:jtorres@lsi.us.es

Thus, in the last few years have come up two main approaches for Model Driven Development: Software Factories (SF) [11] promoted by Microsoft and Model Driven Architecture (MDA) [19] promoted by the Object Management Group (OMG).

Although at first sight it seems that these two approaches clash, they both can be complementary [16,5]. On the one hand, the Software Factories approach is concerned with software product lines, in such a way that it proposes the use of extensible and configurable tools to automate the development and maintenance of the different families of a software product. The automation is obtained by means of the composition and configuration of different components based on frameworks. Thus, the Software Factories approach integrates different activities and techniques. One of these activities consist in the development of different modelling languages and specific tools for the specific domain.

On the other hand, the Model Driven Architecture is an approach that it is based on the use of the modelling standards proposed by the OMG, specially UML[22] and MOF [20]. MDA has promoted the use of transformations because it proposes a framework composed of different levels of modelling (Computation Independent Model, Platform Independent Model and Platform Specific Model). In this framework transformations are the way of obtaining one model in one level from another model or set of models from other level. Thus, models and transformations have become first-class citizens. Figure 1 shows a MOF diagram of the main elements that intervene in the MDA approach.

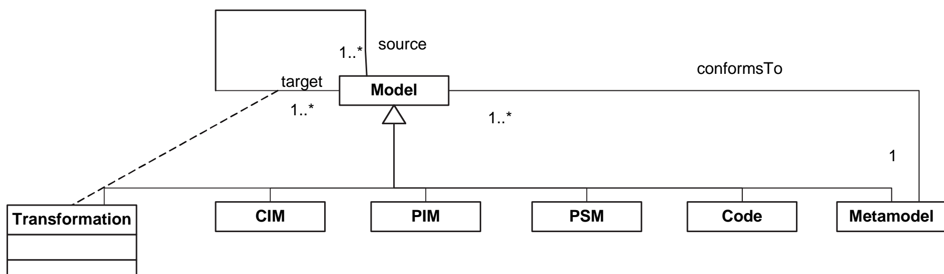

Fig. 1. Metamodel of the MDA approach

In addition to depicting CIMs, PIMs, PSMs and transformations, Figure 1 also shows another important piece in this puzzle: metamodels. A metamodel is a special kind of model which describes the abstract syntax of other models, that is, it describes the pieces (the constructors and the relationships between them) that are important for creating a model.

At this point, it can be seen that metamodels has become key elements for both approaches. Firstly, although MDA promotes the use of the UML, new modelling languages can be described using MOF. And, secondly, Software Factories are based on the definition of domain specific modelling languages, that is, languages that have been designed to facilitate modelling in a concrete

domain, and these languages can be described by means of metamodels.

In this context it is very interesting to define and implement metamodels that can be reused. In fact, the need for defining clean and powerful metamodel extension mechanisms is highlighted in [2].

As metamodels are a special kind of models (Figure 1), the two mechanisms for reuse introduced as part of the Catalysis method [7] can be used: package extension and package template mechanism. In metamodelling these mechanisms are used to assist with the definition of families of languages [4].

In this paper we are going to focus on the package extension mechanism. One metamodel extends another metamodel if it specializes or uses the other metamodel. Thus, it can be said that a `MMA` metamodel extends another `MMB` metamodel if any of the elements of the `MMA` metamodel has a relationship with any of the elements of the `MMB` metamodel. This mechanism allows the definition of the `MMA` and `MMB` metamodels separately, and then merged them to form a complete language. In order to improve the reuse of `MMB` it is important to maintain the obliviousness, that is, the condition of being unaware of being extended by some other metamodel.

This paper shows that with current technologies for implementing metamodels, sometimes is not possible to maintain this obliviousness, especially when the elements of the new metamodel are related to the previous metamodel by means of association and composition relationships.

In order to maintain the original metamodel unaware of being extended, three different approaches have been analyzed. The first approach consists of using inheritance to make the original metamodel oblivious, the problem is solved with traditional object-oriented techniques. The second and third approaches are based on aspect-oriented techniques.

Aspect-Oriented Software Development (AOSD) [9] is a new software development paradigm which intends to improve software evolution through a better localization of concerns. According to [10], there are two properties that are necessary for AOP: *quantification* and *obliviousness*.

Thus, the second approach is based on the introduction of inter-type declarations, while the third one is inspired by [23] and consists of the treatment of relationships in metamodels as aspects. The problem here arises when relationships, that are first-class citizens at modelling, are not managed in the same way at the implementation level.

As a result of the analysis of the three approaches, we think that the third one is the better option to improve the reuse of metamodels, because relationships are better localized, defined and we can take advantage of the AspectJ compiler.

This paper is structured as follows: section 2 introduces the main metamodelling concepts. After that, the Eclipse Modelling Framework is presented, a framework for implementing models, and as a consequence metamodels. Section 4 presents the problems that arise when metamodels are being extended, and analyzes two different situations: one with just inheritance, and the other

one, an extension by means of other kinds of relationships. Afterwards, in section 5 the three different approaches for facing up the improving of the reuse of metamodels are analyzed. Finally the paper is concluded and some future work is pointed out.

2 Metamodelling

According to [15] a metamodel is a precise definition of the constructs and rules needed for creating semantic models. Furthermore, currently, metamodelling can be considered an activity that it is getting much more attention by the research community. This interest is due mainly to three decisive factors: the increase of metamodel-driven technologies and standards; the need to raise the level of abstraction; and, above all, the increasing interest in Model-Driven Software Development.

Metamodelling also can be seen as a way of organizing models, in such a way that the model in one level is described by another model placed on top of it. Thus, the OMG define four different levels of modelling [14]. Figure 2 shows an scheme of the relationships among the levels M0, M1, M2 and M3 defined by the OMG. MOF is placed on the top of the hierarchy, and it is used to define itself; therefore, the level above MOF (M3) can be seen as MOF itself.

UML is at the M2 level. The abstract syntax of UML has been described using MOF. An instance of the UML metamodel can be seen as a class diagram (level M1). Finally, at the M0 level is place the instantiation of the class diagram, that is, the objects in a concrete system.

Currently several metamodel repositories can be found, and they can be classified in two main groups:

- *The MOF-based ones.* MOF[20] is the metamodelling framework proposed by the Object Management Group (OMG) to define other modelling frameworks. The MOF specification is vendor and language independent. With the MOF specification, the OMG has standardized a set of mappings that specify how meta-data is represented and managed in a specific technology. XMI [21] is a XML representation for model interchange between tools, while JMI [25] is an abstract syntax definition for meta-data in Java applications. Some repositories that implement the JMI interface are: MDR from NetBeans [17] or NSMDF [18] from NovoSoft.

  The Coral Metamodelling Framework [24] has the MOF hard coded, although other different metamodels can be installed. Coral is not based in Java, but in Python.

- *The Ecore-based ones.* Ecore is the metamodel included in the Eclipse Modelling Framework (EMF)[8] and it is different from MOF. EMF is a low-cost tool to obtain the benefits of formal modelling and Java code generation and it is language-dependent. The functionality of EMF is similar to MDR.

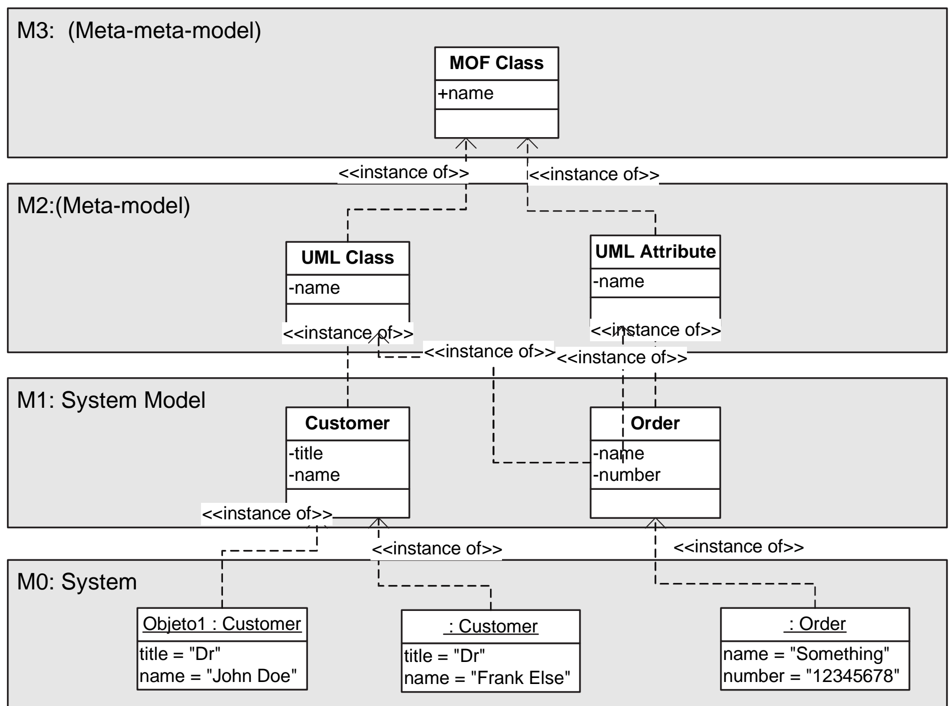

Fig. 2. The four layer metamodel architecture proposed by the OMG

The Java-based approaches (EMF and those based on JMI) use the class and interface concepts of Java to implement the metaclasses.

3 Eclipse Modelling Framework (EMF)

The Eclipse Modelling Framework (EMF)[3] is a modelling framework for Eclipse. EMF is, on the one hand, a framework, and, on the other hand, a facility for defining a model in one of the following forms: Java interfaces, UML diagrams or XML Schemas. `Ecore` is the metamodel that uses EMF to represent models. Ecore is itself an Ecore model, and it is at the same OMG level (Figure 2) as MOF. Figure 3 depicts the different sources of a core model.

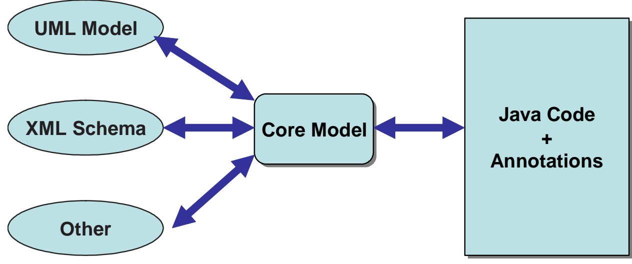

Fig. 3. Sources of a Core Model

There are four basic metaclasses to represent an Ecore model: `EClass`, `EAttribute`, `EReference` and `EDataType`.

- EClass models a class. It has an attribute called name to store the name of the modelled class. And it also has composition relationships with EAttribute and EReference. The cardinality of the composition means that an EClass can have zero or more attributes and zero or more references.
- EAttribute models an attribute. It has an attribute (name) and an association with EDataType. The association represents that an attribute must have a type.
- EReference models and end of an association between classes. It has two attributes: name and containment. containment is true if the association end represents a composition relation. Finally, EReference has an association with EClass. This relation models the target type, that is, the class which is at the other end of the association.
- EDataType models the type of an attribute. It can be a primitive type (int, float, ...) or an object type.

One of the ways of defining the core model is by means of Java interfaces (the other ways of defining core models are not interesting for the purpose of this paper). For each class of the model an interface should be defined. For each attribute and for each reference contained in the class, a standard get() method should be declared in the interface. With this information the EMF generator will deduce the model attributes and references. The Java interfaces and the get() methods should be annotated in order to help the EMF generator to deduce the model properties.

With the interfaces and the annotations, EMF produces two files: a .ecore file and a .genmodel. The .ecore is an XML file that contains the core model. The .genmodel is a kind of wrapper of the core model with extra information. This information is needed by EMF for generating the implementation of the model.

Once the implementation is generated, each Ecore class (that is, each EClass) corresponds to two things in Java: an interface and its corresponding implementation class. For example, let's suppose that we have a class named Book in our Ecore model. This class will be modelled as an EClass in EMF, and it will be mapped to a Java interface (public interface Book) and an implementation class (public class BookImpl ... implements Book) (Figure 4 depicts these relations).

If we define the Ecore model using annotated Java, we will just be in charge of writing the interfaces and the get() methods (if needed). Afterwards, the EMF generator will complete these interfaces with more annotations and set¬() methods. Furthermore, it will generate the implementation classes and all the extra code needed.

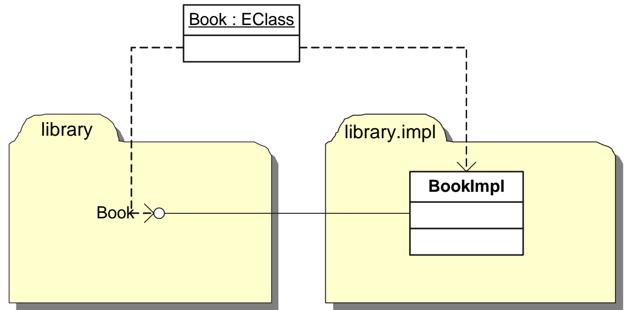

Fig. 4. Relationship between an Ecore class and the generated stuff

4 Problem Statement: Metamodel Extension

Many times it is useful to define a metamodel from other metamodel. In these cases, some kind of extension mechanism is used, that is, the new metamodel extends the original one. This means that, in a certain way, there are some elements in the original metamodel that have some kind of relationship with some elements of the new metamodel. In order to promote the reuse, the original metamodel should be defined in such a way that it is completely unaware of being extended by another metamodel.

This unawareness is beyond of all doubt if we are dealing with MOF metamodels expressed in a graphical way or with XMI. But it isn't so clear we want to express these metamodels in a Java notation, because some problems arise.

As it has been shown in section 3, if we use EMF to implement a metamodel, we have three different ways of expressing the metamodels: using an UML tool such as Rational, by means of XML files or by means of annotated Java files. As problems arise with the Java notation, in this section we are going to focus just on this kind of representation.

To introduce the problem, we have classified the possible relations between the elements of two models (the original and the extended one) into two main groups: inheritance relationships and the rest of relationships. The following subsections introduce two different examples to analyze in depth the consequences of these two kinds of relations in the definition and implementation of metamodels.

4.1 Example 1: Extensions by means of inheritance relationships

This example has been obtained from [8]. It has been chosen because it is a very simple, introductory example. Figure 5 shows a package named `Library` which holds a MOF metamodel for a library. This package contains three metaclasses (`Library`, `Book` and `Writer`) and one enumeration (`BookCategory`). For writing this metamodel with EMF and using a Java notation, three interfaces (one for each metaclass) have to be defined. In each interface a number of `get()` methods should be written (one `get()` method for each

attribute and another one for each reference). Furthermore, a new class should be defined for the enumeration.

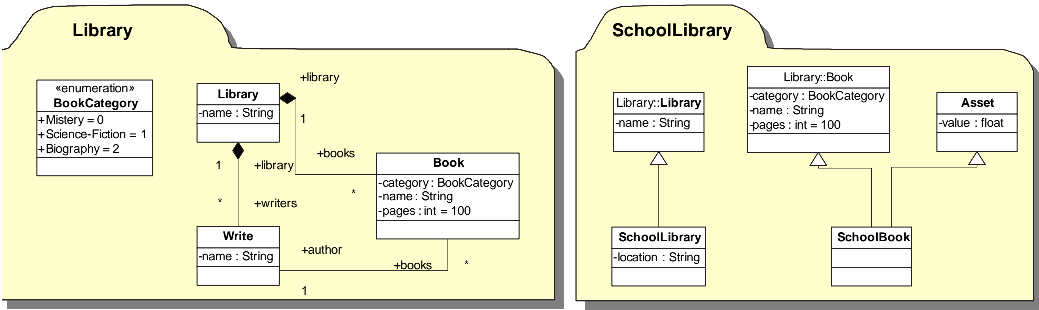

Fig. 5. Exteding a metamodel by means of inheritance relationship

Figure 6 shows part of the implementation of the Library metamodel with EMF. Concretely, the `Library` metaclass and the `BookCategory` enumeration.

```
package library;
import java.util.List;
/**
 * @model
 */
public interface Library
{
  /**
   * @model
   */
  String getName();

  /**
   * @model type="Writer" containment="true"
   */
  List getWriters();

  /**
   * @model type="Book" containment="true"
   */
  List getBooks();
}
```

```
package library;
/**
 * @model
 */
public class BookCategory
{
  /**
   * @model name="Mystery"
   */
  public static final int MYSTERY = 0;

  /**
   * @model name="ScienceFiction"
   */
  public static final int SCIENCE_FICTION = 1;

  /**
   * @model name="Biography"
   */
  public static final int BIOGRAPHY = 2;
}
```

Fig. 6. Part of the Library EMF metamodel with Java notation

Figure 5 also shows the `SchoolLibrary` package which extends the `Library` package. From the MOF model point of view, this extension implies that the `SchoolLibrary` package includes two classes (`Book` and `Library`) with the stereotype `<<from Library>>`. This stereotype means that those metaclasses belong to the `Library` package. In Figure 6, this is depicted with the name of the package placed just before the name of the class (`Library::Book`). If we look at the annotated Java implementation, we will realize that, in this case, the extension is obtained by writing an extends clause in the `SchoolLibrary` and `SchoolBook` interfaces. These sentences express the inheritance relationship between the pair of classes `SchoolLibrary-Library` and `SchoolBook-Book`. The classes `Book` and `Library` don't have to be modified.

```
package schoollibrary;
import library.Library;
/**
 * @model
 */
public interface SchoolLibrary extends Library
{
  /**
   * @model
   */
  String getLocation();
}
```

```
package schoollibrary;
import library.Book;

/**
 * @model
 */
public interface SchoolBook extends Book, Asset
{
}
```

Fig. 7. Part of the extended SchoolLibrary package

Thus, as a conclusion, if we extend a metamodel by means of inheritance the obliviousness of the original metamodel is maintained.

4.2 Example 2: Extensions by means of associations

The example in this section is a bit more complex that the previous one. In this case, a metamodel is going to be extended, but not only with inheritance relationships, but also with other kind of relations. Figure 8 shows the relation between the two packages that intervene in this example. The original package is called `com.metamodels.java2`, it has been obtained from [6], and it is a leaner version of the UML metamodel which tailors the UML metaclasses to the Java 2 Specification. The `com.metamodels.aspectj` package contains an AspectJ metamodel which extends the Java metamodel according to the AspectJ specification [1]. AspectJ is an extension to Java to develop aspect-oriented applications. The AspectJ metamodel has been obtained from [12].

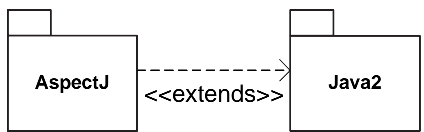

Fig. 8. Package relationship

Figure 9 shows the MOF Java 2 metamodel. Most of the metaclasses use the same name as their corresponding UML metaclasses. We only are going to focus on the metaclasses that are relevant to our example. They are: `Element`, `Generalization`, `Feature` and `Parameter`.

The `Generalization` class models the `extend` and `implements` relationships between classifiers (classes and/or interfaces). An instance of this class represents an inheritance relationship between a subtype and a supertype. It can also represent the relationship between a class and an interface.

A `Feature` represents something that can be declared in a class or an interface (a field, a constructor or an ordinary method).

The `Parameter` class abstracts the parameters in a method or constructor. This relationship is represented by a composition relation between `Parameter`

and `BehavioralFeature`. A parameter is also related to a `Type`.

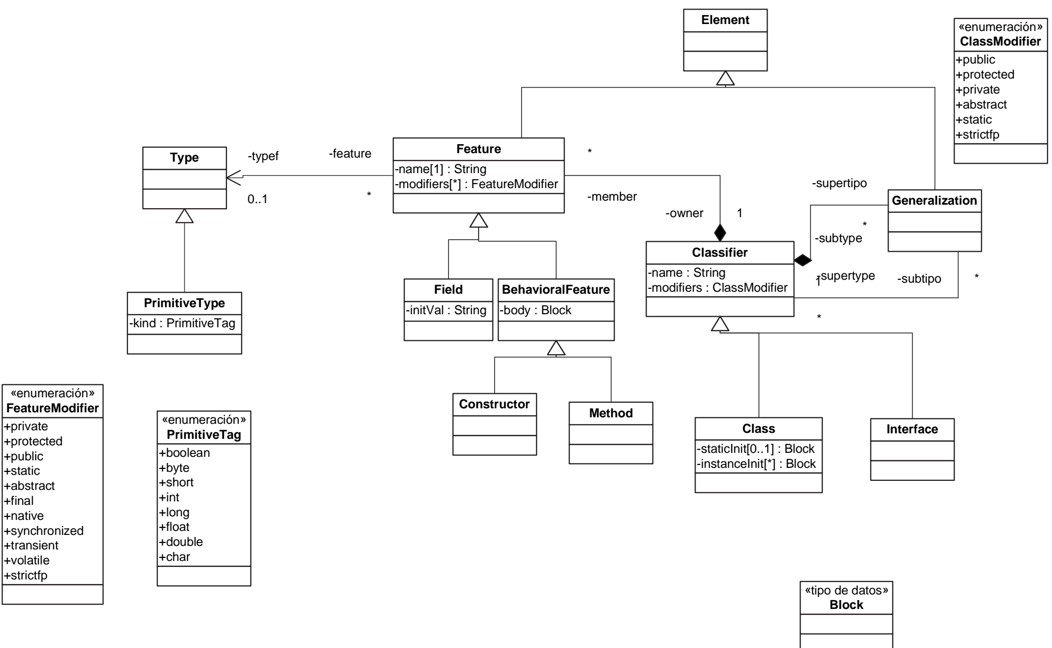

Fig. 9. Java 2 Metamodel

In order to specify this metamodel using annotated Java one interface should be created for each metaclass and one final static class for each enumeration. Figure 10 shows the code needed for generating the metaclasses `Feature` and `Class`. `Feature` has two attributes `name` and `modifiers` which in the implementation are transformed into two `get()` methods, `getName` and `getModifiers`.

`Feature` is also related to `Type` and `Classifier`. The first relation is an association while the second one is a composition relationship. From the point of view of `Feature` there is no difference between these two relationships, there are two `get()` methods (`getOwner` and `getTypefeature`) with an opposite annotation. This annotation is referred to the name of the other end of the association. Here the difference between these two kinds of relationships is specified in the classes `Type` and `Classifier`, respectively. The `Type` interface will have a `get()` method named `getFeatures` while the `Classifier` interface will have a `get()` method (`getMember`) with an extra annotation (`containment=true`). This containment annotation means that the relationship has a composition semantics.

The Java2 metamodel is extended by the metamodel depicted in Figure 11. As occurred in the example introduced in the previous section, at the modelling level, there is no problem to reuse some metaclasses belonging to the `com.metamodel.java2` package. We only need to include the classes with the stereotype `<<Java2>>` (in the metamodel represented in Figure 11 the stereotyped classes are `Class`, `Generalization`, `Feature`, `Element` and `Parameter`.

```
package com.metamodel.java2;

import org.eclipse.emf.common.util.EList;

/**
 * @model
 */
public interface Feature extends Element{
   /**
    * @model
    */
   String getName();

   /**
    * @model
    * type="com.metamodel.java2.FeatureModifier"
    */
   EList getModifiers();

   /**
    * @model opposite="member"
    */
   Classifier getOwner();

   /**
    * @model opposite="features"
    */
   Type getTypefeature();

}
```

```
package com.metamodel.java2;

import org.eclipse.emf.common.util.EList;

/**
 * @model
 */
public interface Class extends Classifier{
   /**
    * @model
    * dataType="com.metamodel.java2.Block"
    */
   Block getStaticInit();

   /**
    * @model
    * dataType="com.metamodel.java2.Block"
    */
   Block getInstanceInit();

   /**
    * @model
    * type="com.metamodel.java2.BehavioralFeature"
    * opposite="thrownExceptions"
    */
   EList getBehavioralFeatures();

}
```

Fig. 10. Java 2 Implementation

But when we move to the implementation level, things don't remain the same.

If we look at the `Element` metaclass, we can realize, as it happened in the example shown in the subsection 4.1, that there is no problem, because the only relation of `Element` with a class belonging to the `com.metamodel.aspectj` package is an inheritance relationship. But the other metaclasses in the `com.metamodel.java2` package maintain other kinds of relations with the metaclasses in the `com.metamodel.aspectj` package. In fact, there are composition relationships between `Class` and `Pointcut`; between `Parameter` and `Pointcut`; and between `Parameter` and `Advice`. Furthermore, there are also associations between `Generalization` and `Aspect` and between `Feature` and `Aspect`.

If we want to implement these relationships we need to declare `get()` methods in the interfaces. Figure 12 shows two of these interfaces, `Pointcut` and `Aspect`. If we look at the `Pointcut` interface we will realize that there is an annotated method called `getDeclarer` that models one edge of the relationship between `Pointcut` and `Class`. But in order to model the other extreme of the association, a new annotated `get()` method with an annotation `containment=true` should be included in the `Class` interface. Therefore, at this point we need to modify the `Class` interface and, as a consequence, the `com.metamodel.java2` package has to be aware of being extended with the `com.metamodel.aspectj` package.

The same problem arises when we try to implement the association between `Aspect` and `Feature` the `getIntroducedFeatures` method implements one edge of the relationship, but in order to implement the other edge we have to

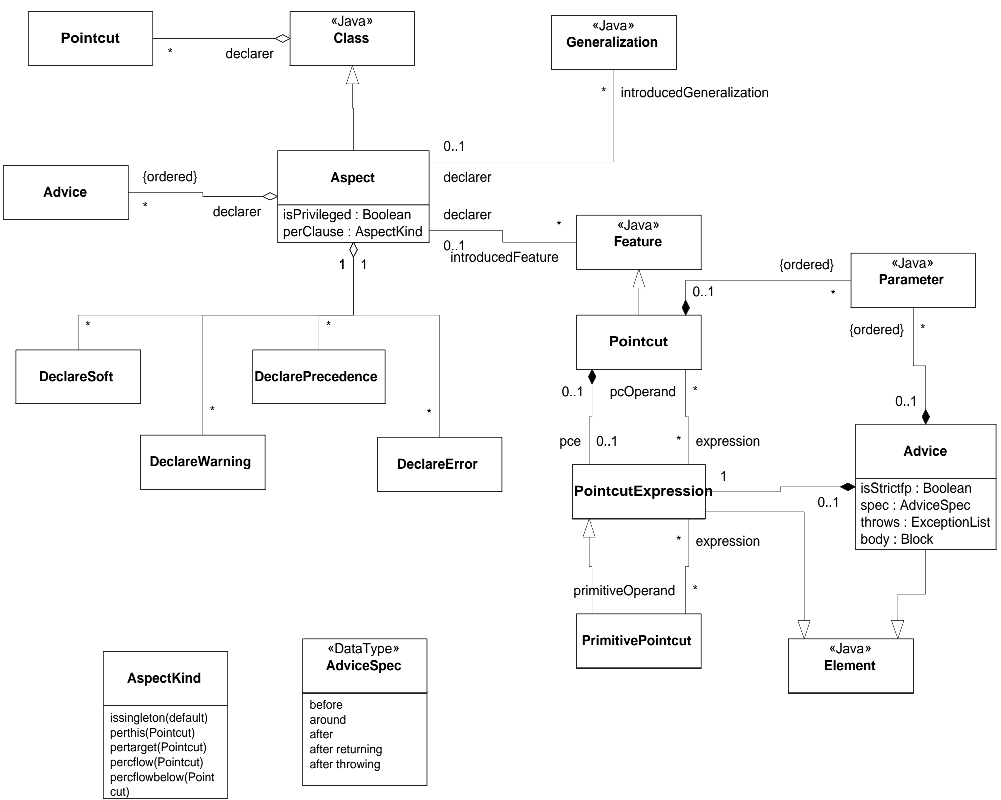

Fig. 11. AspectJ Metamodel

declare a new get method in the `Feature` interface.

As a conclusion, if we extend a metamodel using relationships different from inheritance the obliviousness of the original metamodel is lost.

5 Using aspect technology to improve the reuse of metamodels

This section describes three different approaches for making the `com.metamodel.java2` unaware of being extended by another metamodel. The first one is based on the traditional object-oriented inheritance mechanism, while the second and the third ones are based on aspect-oriented solutions. In [13] a comparison between inheritance and aspect-oriented mechanisms for changing the behavior of objects is made.

5.1 First Approach: Inheritance

This first approach is based on a traditional object-oriented solution. The idea behind this approach is to leave the inheritance as the only possible extension mechanism between elements of different packages. This assumption implies that we have to create a new fictitious metaclass in the new metamodel which will extend the class in the original metamodel. Thus, the associations and

```
package com.metamodel.aspectj;

import com.metamodel.java2.Feature;
import org.eclipse.emf.common.util.EList;
import com.metamodel.java2.Class;
import com.metamodel.java2.Parameter;

/**
 * @model
 */
public interface Pointcut extends Feature {

   /**
    * @model
    * type="com.metamodel.java2.Class"
    * lowerBound="1" upperBound="1"
    */
    Class getDeclarer();

   /**
    * @model
    * type="com.metamodel.java2.Parameter"
    * containment="true"
    */
    EList getParameters();

   /**
    * @model type="PointcutExpression"
    * containment="true" lowerBound="0"
    * upperBound="1"
    */
    PointcutExpression getPce();

   /**
    * @model type="PointcutExpression"
    * opposite="pcOperand"
    */
    EList getExpression();
}
```

```
package com.metamodel.aspectj;

import org.eclipse.emf.common.util.EList;
import com.metamodel.java2.Class;
import com.metamodel.java2.Feature;

/**
 * @model
 */
public interface Aspect extends Class{
   /**
    * @model
    */
    boolean isIsPrivileged();

   /**
    * @model
    */
    AspectKind getPerClause();

   /**
    * @model type="Advice" containment="true"
    */
    EList getAdvices();

   /**
    * @model
    * type="com.metamodel.java2.Generalization"
    */
    EList getGeneralizations();

   /**
    * @model type="com.metamodel.java2.Feature"
    */
    EList getIntroducedFeatures();

   /**
    * @model type="DeclareSoft"
    * containment="true"
    */
    EList getDeclareSoft();

   /**
    * @model type="DeclareWarning"
    * containment="true"
    */
    EList getDeclareWarning();

   /**
    * @model type="DeclarePrecedence"
    * containment="true"
    */
    EList getDeclarePrecedence();

   /**
    * @model type="DeclareError"
    * containment="true"
    */
    EList getDeclareError();
}
```

Fig. 12. AspectJ Implementation

the composition relationships will relate this fictitious class with the class in the new metamodel. This idea is better explained if we look at Figure 13. This figure shows an excerpt of the `com.metamodel.aspectj` after applying this approach.

If we compare the excerpt shown in Figure 13 and the original AspecJ metamodel depicted in Figure 11, we will realize that a new metaclass named `Class` has been added. This metaclass is the introduced fictitious class and it inherits from the metaclass `Class` declared in the package `com.metamodel`¬

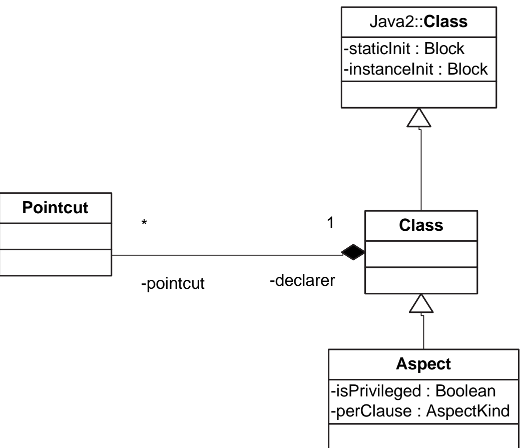

Fig. 13. Excerpt of the AspectJ implementation package applying the inheritance approach

`.java2`. The figure also shows that the composition relationship which appeared between `Java2::Class` and `Pointcut` in the original metamodel (Figure 11) has changed, and in Figure 13 it has turned into a composition between `Pointcut` and `Class` (the introduced fictitious class). Therefore, if we apply this same approach to all the relations between the classes included in both packages, we will have that four new fictitious classes are needed in the `com.metamodel.java2` package.

Although this approach is simple, it has an important inconvenient: Our `aspectj` metamodel implementation is tangled with a set of classes that are needed for implementation but that are mining the clarity of our metamodel.

5.2 Second Approach: Inter-type declarations

The second approach has been inspired by AspectJ. The idea is to introduce a new constructor that can be understood by the EMF generator in order to deal with aspects. The constructor will be `aspect` and, as in AspectJ, the only thing we have to do is to introduce a inter-type declaration. Thus, to implement the relationship between `Class` and `Pointcut` we will write something similar to the code that appears in Figure 14.

But, if the AspectJ syntaxis and compiler are going to be used, the solution shown in Figure 14 presents a problem: in AspectJ only classes are allowed to be introduced with inter-type declarations. And we have to remember that `Class` is an interface. In this case, the EMF generator should ignore this error and generate an aspect for the class `ClassImpl`. Here the EMF generator should read and use the annotations that have been included in the aspect.

5.3 Third Approach: Relationship Aspect

The third approach is related to the treatment of relatioships as first-class citizens. The problem here is that although relationships are treated as first-class

```
import com.metamodel.java2.Class;
import org.eclipse.emf.common.util.EList;

/**
  * @model
  */
public aspect ClassAsp {

   /**
    * @model type="com.metamodel.aspecj.Pointcut"
    * containment=true
    **/
    public EList Class.getPointcuts();

}
```

Fig. 14. Definition of an aspect with an inter-type declaration

citizens at the modelling level, this same treatment is not maintained at the implementation level. At this level, the implementation of the relationships is hand-crafted and spread across the objects which participate in those relationships. In [23], these problems are faced up to by modelling relationships as separable, crosscutting concerns. Thus, the clarity and cohesion of each participating class is improved.

In order to clarify this approach, we will firstly introduce an example of the relationship aspect obtained from [23], and afterwards, we will apply this relationship aspect to the reuse of metamodels. Figure 15 shows an UML representation of a relationship between the students that attend courses.

Fig. 15. Simple UML Diagram that shows a relationship between the students that attend courses

If we want to implement the UML diagram of Figure 15, we have to hard code the relationship in the classes `Student` and `Course`. Thus, Figure 16 shows a possible implementation of this diagram. There it can be seen that there is an attribute **`attends`** in the `Student` class to implement one end of the relationship, and another attribute **`attendees`** in the `Course` class to implement the other end of the relationship.

Using the *Relationship Aspect Library (RAL)* proposed in [23] the relationship `Attend` is implemented as an aspect which extends `SimpleStaticRelationship`, a generic AspectJ aspect from the RAL library. This implementation can be seen in Figure 17.

If we apply this approach to our metamodel, we will obtain the code shown in Figure 18. That is, instead of hand code `get()` methods in the interfaces representing the ends of the relationships, we define an aspect for each relationship in the metamodel. This aspect will extend an abstract aspect that will be defined in an Aspect Relationship Library. This library can be the one

```
import java.util.HashSet;

public class Student {

        String name;
        Integer number;

        HashSet<Course> attends;
}
```

```
import java.util.HashSet;

public class Course {
        String code;
        String title;
        Integer workload;

        HashSet<Student> attendees;
}
```

Fig. 16. An implementation of the UML Diagram of Fig. 15

```
import java.util.HashSet;

public class Student {

        String name;
        Integer number;

}
```

```
import java.util.HashSet;

public class Course {
        String code;
        String title;
        Integer workload;

}
```

```
public aspect Attends extends StaticRel<Student, Course>{
}
```

Fig. 17. An implementation of the UML Diagram of Fig. 15

defined in [23] or a new one of our creation. For example, the kind of relation `CompositionRel` used in Figure 18 is not defined in the *RAL* library. There are also other limitations of the *RAL* library that should be taken into account. The current implementation of the *RAL* library does not let us define two different relationships between the same classes due to name problems. For example, in the `com.metamodel.aspectj` package we will have problems to define the association and the composition relationships between `Pointcut` and `PointcutExpression`.

```
import com.metamodel.java2.Class;

/**
 * @model
 **/
public aspect PointcutDeclaration extends CompositionRel <Class, Pointcut>{

}
```

Fig. 18. An aspect for implementing the relationship between Pointcut and Class

Using this approach, the clarity and cohesion of the implementation of each metaclass is increased. If a comparison between the second approach and this one is made, we have an important advantage, the sintaxis is totally compatible with the AspectJ compiler.

6 Related work

In [26] a method for model evolution using model transformations based on aspect orientation is proposed. They propose a mechanism for modularizing crosscutting concerns, but they are focused on the modelling level, while our proposal deals with the reuse of the implementation of metamodels.

On the other hand, in [4] a metamodel definition of the package extension mechanism is given. But they also have been focus on the definition of the different elements that intervene in a package extension, and it is at the modelling level. There is no assumption at the implementation level.

7 Conclusions and further work

This paper has presented a problem that arises when trying to extend a metamodel with relationships different from inheritance. In this case, the original metamodel is not oblivious because it should be modified in order to implement correctly the extension.

After that, three different approaches to solve the problem have been introduced. The first approach is based on the inheritance mechanism of traditional object-oriented languages. Although it is a simple approach, the new metamodel is spread with different virtual classes that are only needed for implementation.

The second approach is based on the introduction of inter-type declarations. An inter-type declaration is a concept that has appeared thanks to aspect-oriented languages. Although this approach is the first we can think of when we work with aspect-orientation, it has a great inconvenient: the AspectJ compiler does not allow the use of inter-type declarations for interfaces.

The third approach consists of treating relationships as first-class citizens at the implementation level. This aim is gotten with the definition of relationships as aspects. The Relationship Aspect Library (RAL) [23] has been taken as an example, but this library should be extended to deal with more kinds of relationships. There is also another problem: the RAL library only allows the definition of one relationship between to classes, and that is a great handicap for metamodelling.

As a result of the analysis of the three approaches, we think that the third one is the better option to improve the reuse of metamodels, firstly, because relationships are better localized, defined and the coupling of the metamodel implementation is reduced. Secondly, because we can take advantage of the AspectJ compiler and all the written code is validated by the AspectJ compiler. And, finally, because relationships themselves can be reused.

As a future line of work we will like to adapt the RAL library to our needs. This implies the solution of some problems that has the RAL library (as the name problem produced when two different relationships are defined between two classes), and the definition of other kinds of relationships. We

also would like to add this aspect oriented features to current modelling and metamodelling frameworks.

References

[1] The AspectJ Team. *AspectJ Programming Guide (v.1.2)*. Avalaible at: http://www.eclipse.org/aspectj. 2003.

[2] J. Bzivin, F. Jouault, P. Valduriez. *First Experiments with a ModelWeaver*. Proceedings of the Workshop on Best Practices for Model Driven Software Development held in conjunction with the OOSPLA conference. 2004.

[3] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, T. J. Grose. *Eclipse Modelling Framework: A Developer's Guide*. Addison-Wesley, 2003.

[4] T. Clark, A. Evans, S. Kent. *A metamodel for Package Extension with Renaming*. Proceedings of the UML Conference (UML'02). LCNCS 2460, pp. 305-320, 2002.

[5] S. Cook, *Domain-Specific Modeling and Model Driven Architecture*, MDA Journal. Jan, 2004.

[6] S. Dedic and M. Matula *Metamodel for the Java Language* Avalaible at: http://java.netbeans.org/models/java/java-model.html.

[7] D. D'Souza, A. Willis. *Objects, Components and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1998.

[8] Eclipse, *Tutorial: Generating Extended EMF 2.1 Model*.Avalaible at: http://www.eclipse.org/emf. July, 2005.

[9] T. Elrad, R. E. Filman, A. Bader. *Aspect Oriented Programming*. Communications of the ACM. Vol. 44, n. 10., Oct. 2001.

[10] R. E.Filman, D. P. Friedman. *Aspect-Oriented Programming is Quantification and Obliviousness*. Proceedings of the Workshop on Advanced Separation of Concerns, OOPSLA 2000. Oct, 2000.

[11] J. Greenfield, K. Short, S. Cook, S. Kent, *Software Factories. Assembling Applications with Patterns, Models, Frameworks and Tools*.Wiley Publishing, Inc., 2004.

[12] . Y. Han, G. Kniesel, A. Cremers. *Towards Visual AspectJ by a MetaModel and Modeling Notation*. Proceedings of the 6th International Workshop on Aspect-Oriented Modeling held in conjunction with the 4th International Conference on Aspect-Oriented Software Development (AOSD'05). Chicago, Illinois, USA. Mar, 2005.

[13] S. Hanenberg, R. Unland. *Concerning AOP and Inheritance*In: Mehner, K., Mezini, M., Pulvermller, E., Speck, A. (Eds.): Aspect-Orientation - Workshop. Paderborn, Mai 2001, University of Paderborn, Technical Report, tr-ri-01-223, 2001.

[14] A. Kleppe, J. Warner, W. Best. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2004.

[15] metamodel.com "What is metamodelling?". Avalaible at: http://www.metamodel.com/staticpages/index.php?page=20021010231056977.

[16] J. Muñoz, V. Pelechano. *MDA vs. Factorías Software*. Proceedings of the Second Workshop on Model-Driven Development, MDA and Applications (DSDM'05).pp. 1-10.Granada, Spain. Sept, 2005.

[17] NetBeans. *Metadata Repository (MDR)*. Avalaible at: http://mdr.netbeans.org/.

[18] Novosoft. *NovoSoft Metadata Framework*. Avalaible at: http://nsuml.sourceforge.net/.

[19] OMG, *MDA Guide Version 1.0*, Eds. J. Miller and J. Mukerji. May, 2003.

[20] OMG, *Meta Object Facility Specification Version 2.0*,January, 2006.

[21] OMG. *MOF 2.0 / XMI Mapping Specification, v2.1*. Jan, 2006. Avalaible at: http://www.omg.org/technology/documents/formal/xmi.htm.

[22] OMG, *UML 2.0 Superstructure Specification*, 2004.

[23] D. J. Pearce, J. Noble. *Relatioships Aspects*. Proceedings of the 5th Aspect Oriented Sofware Development Conference (AOSD'06). Bonn, Germany. Mar, 2006.

[24] I. Porres. *A Toolkit for Model Manipulation* Springer International Journal on Software and Systems Modeling, vol: 2, num: 4, 2003.

[25] Sun Corporation: "The JavaTM Metadata Interface (JMI) Specification". Jun, 2002. Avalaible at: http://jcp.org/aboutJava/communityprocess/final/jsr040/index.html.

[26] N. Ubayashi, T. Tamai, S. Sano, Y. Maeno, S. Murakami. *Model Evolution with Aspect-Oriented Mechanisms*. Proceedings of the 2005 Eighth International Workshop on Principles of Software Evolution (IWPSE'05). IEEE Publishing.

Concern-Specific Languages in a Visual Web Service Creation Environment

Mathieu Braem [1] Niels Joncheere [2] Wim Vanderperren [3]
Ragnhild Van Der Straeten [4] Viviane Jonckers [5]

System and Software Engineering Lab
Vrije Universiteit Brussel
Pleinlaan 2
B-1050 Brussel, Belgium

Abstract

This paper presents a high-level, visual Service Creation Environment (SCE) for web services. The SCE introduces two main concepts: services and composition templates. Composition templates are abstract descriptions of reusable compositions containing several placeholders for services. Services are verified to be compatible with the composition template when a service is mapped onto a composition template. The SCE supports the modularization of crosscutting concerns using both the general-purpose AOP language Padus and several concern-specific languages. Aspects can be visually deployed on a target composition template or service, which automatically triggers the weaving process.

Key words: Service-Oriented Architecture, Concern-Specific Languages, Aspect-Oriented Software Development, Web Services

1 Introduction

Over the last years, *web services* [2] have been gaining a lot of popularity as a means of integrating existing software in new environments. Basic web services can be created by exposing existing applications to the internet using XML front-ends. By composing a number of basic web services, new web services can be created that provide more advanced functionality. These compound web

[1] Email: `mbraem@vub.ac.be`
[2] Email: `njonchee@vub.ac.be`
[3] Email: `wvdperre@vub.ac.be`
[4] Email: `rvdstrae@vub.ac.be`
[5] Email: `vejoncke@ssel.vub.ac.be`

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

services can then be used by other web services, further improving software reusability.

Originally, the only way to compose web services was by manually writing the necessary glue-code in programming languages such as C and Java. It quickly became clear, however, that a composition of web services is more naturally captured by dedicated *workflow languages* [14] than by general-purpose programming languages. Today, the most popular workflow language with regard to the composition of web services is the *Business Process Execution Language* (WS-BPEL) [3]. WS-BPEL processes are platform- and transport-independent, and are expressed using XML. Recently, a higher-level visual notation for WS-BPEL, called the Business Process Modeling Notation (BPMN) [36], has been proposed.

Meanwhile, *aspect-oriented software development* (AOSD) has been proposed as a means of improving *separation of concerns* [26] in software. AOSD is based on the observation that a number of concerns in software (such as logging [18] and billing [13]) cannot be modularized using object-oriented software development: a program can only be decomposed in one way (i.e., according to the class hierarchy), and concerns that do not align with this decomposition end up scattered across the program and tangled with one another. This problem is dubbed "the tyranny of the dominant decomposition" [25]. AOSD allows expressing such *crosscutting concerns* in well-modularized *aspects*, so that adding, modifying or removing such concerns does not require changes to the main program.

Initial research on AOSD has concentrated on applying its principles to the object-oriented programming paradigm. Arsanjani *et al.* [4] and others [10,12,35] have shown that AOSD has a lot of potential in a web services context, too.

Although workflow languages are better suited for web service composition than general-purpose programming languages, they still require a large amount of in-depth technical knowledge. In order to facilitate service composition without requiring such in-depth technical knowledge, a higher level of abstraction is required. We therefore propose a visual *service creation environment* (SCE), which allows user-friendly configuration of web service compositions using reusable *composition templates*, and which supports encapsulating crosscutting concerns using both general-purpose and concern-specific aspect languages. This environment is implemented as a plug-in for the Eclipse platform [15].

The outline of the paper is as follows: Section 2 explains the motivation for the SCE, and Section 3 provides an overview of the SCE architecture. Next, the support for concern-specific languages in the SCE is presented. Section 5 describes related work, and Section 6 states our conclusions and future work.

2 Motivation for the Service Creation Environment

The research presented in this paper is conducted in the context of the WIT-CASE project, which is partly funded by Alcatel Belgium, a telecom company, and by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen).

In the telecom community, the *service delivery platform* (SDP) is a central ICT infrastructure that is targeted at the development, deployment and execution of value-added telecom services by network operators as well as by third-party service providers. Our approach focuses on the development and configuration of service compositions in a visual *service creation environment* (SCE). A number of key requirements for such an environment stated by the telecom partner are:

- A *faster introduction of new services* is needed *through innovative service creation mechanisms* ranging from using open ICT programming environments to service composition tools, reuse of common components and integration of service logic with business applications.
- *Fast and easy modification of service and business logic of baseline services* is required.
- The ability to *offer service bundles to customers as a strategic move against competition* is also required. This requirement implies that common capabilities must be offered on which these services of the service bundles can rely, e.g., common billing and the capability to offer flexible tariff packages for the grouped services.
- A last requirement is the *reduction of operational expenses by service providers*. Therefore, different end-user services should as much as possible be based on generic reusable building blocks. Furthermore, service providers want to build end-user services and applications on top of an integration platform instead of deploying a collection of out-of-the box end-user services and applications. An integration platform offers the additional benefit of integration with legacy network infrastructure.

In order to meet the above stated requirements, the SCE needs to allow the configuration of service compositions on a high level of abstraction in order to facilitate application development without in-depth technical knowledge of the involved services. In order to achieve this, the following objectives are pursued:

- A visual SCE that enables easy plug-and-play composition of both internal and external services.
- The SCE has to guide the service composition process by providing feedback about the correctness of the resulting service composition.
- The SCE has to allow describing management concerns (e.g., billing) of the resulting service composition in a concise and declarative manner.

The current state-of-the-art is insufficient for supporting the envisioned SCE. Typical workflow languages (such as WS-BPEL) provide visual GUIs in order to facilitate the creation of workflows. However, these GUIs are nothing more than a visual interface on top of the language. Examples of such GUIs are BPWS4J [6] and Oracle BPEL Designer [24]. There is no support for guiding the service composition process to a correct service composition. Furthermore, management concerns still have to be encoded in the workflow itself, which results in a workflow that is tangled with several secondary concerns, and as such makes the resulting workflow more complex. In the next section, we introduce our SCE and show how the above stated objectives are met.

3 The Service Creation Environment

In this section, we first introduce the architecture of our visual SCE. Secondly, we explain how the SCE supports AOSD. Next, the GUI of the SCE is presented, followed by an explanation on how services can be composed, on how service compositions can be verified, and finally, on how services can be deployed.

3.1 Architecture of the SCE

Figure 1 gives an overview of the architecture of the SCE. The SCE contains three repositories:

- A first repository contains a set of *documented services*. The services contained in this repository are the basic building blocks of the SCE. Each service is described by a WSDL file. In addition, each service is documented by a WS-BPEL process that specifies the external protocol information and by a description of basic quality of service requirements.
- Another repository contains a set of documented *composition templates*. These templates are specified in WS-BPEL. The templates are abstract descriptions of web service compositions and may contain one or more placeholders for services. The composition templates can be instantiated by filling in the placeholders with different services. When services are added to service composition templates, the SCE checks whether the protocols of the services are compatible with the protocol of the service composition template.
- A third repository contains different *crosscutting concerns* corresponding to management concerns such as billing schemes. A crosscutting concern can be connected to services and composition templates visually or through a pointcut language.

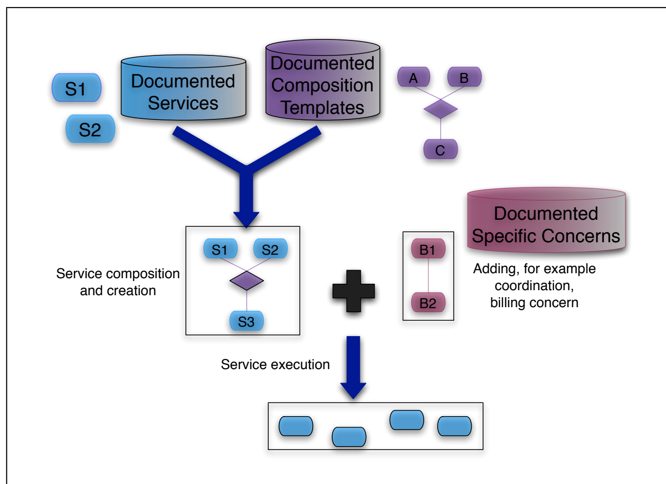

Fig. 1. SCE architecture

3.2 Aspects and Padus

The SCE supports the modularization of crosscutting concerns through the Padus [7] language. Padus is our aspect-oriented process language based on WS-BPEL. A detailed explanation of Padus is outside the scope of this paper. We only introduce and explain the features of Padus relevant for the SCE and for defining concern-specific languages.

Padus is an XML-based language, and introduces two main concepts: *aspects* and *aspect deployments*. An aspect is a reusable description of a crosscutting concern, and contains one or more pointcuts and advice. A pointcut selects interesting points in the execution of the target WS-BPEL process (called joinpoints), and exposes target objects to the advice. The pointcut language of Padus is a logic language based on Prolog, and is thus very expressive [17]. The complete target WS-BPEL process is reified as a collection of facts that can be queried by the pointcut. The advice language is WS-BPEL, extended with some AOSD-specific constructs. The Padus technology is based on a traditional static weaver that processes the target WS-BPEL processes and generates new WS-BPEL processes containing the advice code as specified in our visual environment. The main advantage of this approach is the compatibility with existing infrastructure, as the output can be deployed on any WS-BPEL-compatible engine.

3.3 SCE GUI

Figure 2 provides a screenshot of the SCE's interface. The editor view (in the middle of the screen) is used to edit compositions, and consists of two main parts: a large drawing canvas, and a smaller palette. The palette contains some selection and connection tools, and shows the available services, composition templates and aspects as they are loaded from the library. By double-clicking on an entity, the configured editor for that entity is launched. For instance, a graphical BPMN-based editor is launched for a composition

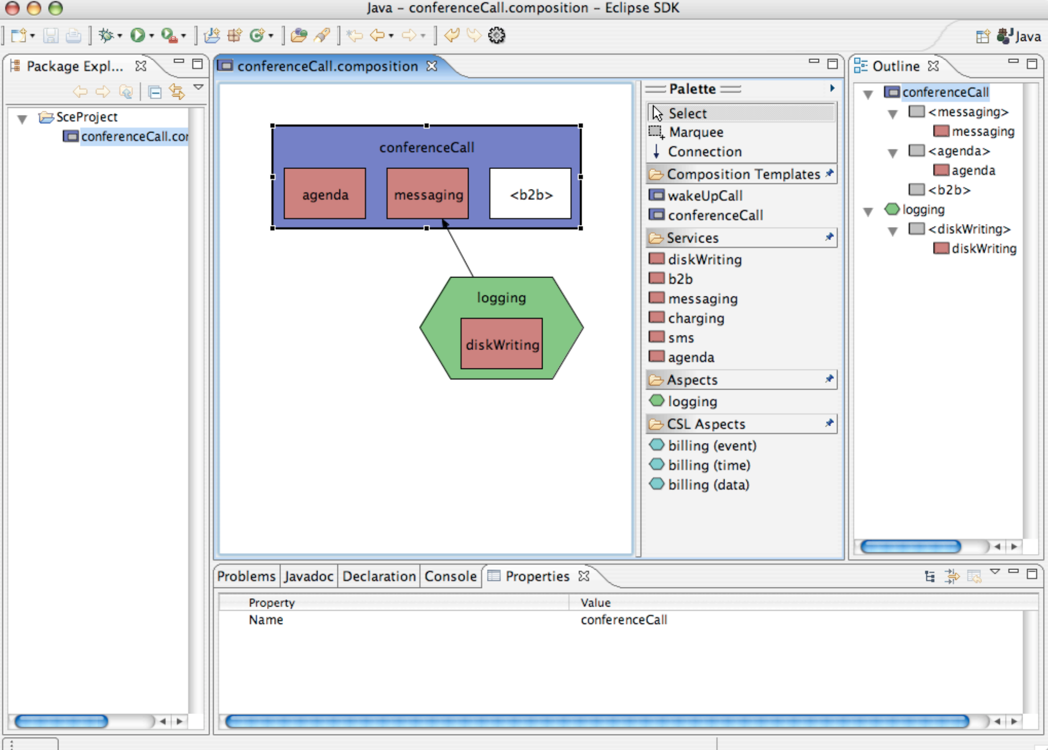

Fig. 2. Screenshot of the SCE's interface

template. The changes made through the visual editor are taken into account for the composition at hand. As such, a composition template can be adapted on the fly and at a higher level of abstraction than WS-BPEL code.

The outline view (at the right of the screen) shows a tree-based overview of the state of the composition, and the properties view (at the bottom of the screen) shows the properties of the element that is currently selected in the editor view or in the outline view.

3.4 Composition

In order to create a composition in the SCE, it suffices to drag a composition template on the composition canvas and fill all the placeholders with concrete services. Aspects can be connected to services, meaning that they will only be applied to these concrete services, or to a complete composition template, in order to apply them to all the services that take part in this composition.

The composition shown in Figure 2 contains a composition template called "conferenceCall" with three placeholders. Two services called "agenda" and "messaging" have been added to the composition template's placeholders, while one placeholder is still empty. This placeholder should be filled in before the composition can work, for instance using the "b2b" service available in the library. The composition also contains an aspect called "logging", which is connected to the "messaging" service. A service called "diskWriting" has

been added to the aspect's only placeholder. The result of this composition would be that the conference call application works using the selected services, and that a logging aspect, which invokes the disk writing service, is deployed to the messaging service in order to log the messaging actions selected by the aspect's pointcut.

3.5 Verification

An important requirement of the SCE is that it guides users in creating correct compositions without requiring in-depth technical knowledge. The SCE accomplishes this by verifying whether compositions are correct while they are created: when a service is dragged onto a placeholder, the SCE checks whether the service's protocol is compatible with the composition template's protocol. If the service turns out to be incompatible, a report is generated that provides mismatch feedback to the user. Compatibility checking based on protocols rather than plain APIs is possible because every service is explicitly documented with a protocol specification expressed in WS-BPEL.

In literature, a wealth of research exists on the topic of protocol verification [9,21,33,27,38]. Our verification engine is based on the PacoSuite approach [37], which introduces algorithms based on automata theory to perform protocol verification. In order to provide protocol verification in the SCE, the WS-BPEL specifications of each service, aspect and composition template are translated into deterministic finite automata (DFA). By applying the algorithms introduced by the PacoSuite approach, the SCE can decide whether the service's protocol is compatible with the composition template's protocol.

In case a certain service is not compatible with a certain composition template placeholder, the user has two options: Either select another service for that placeholder, or edit the composition template on the fly as described in Section 3.3.

3.6 Code Generation and Deployment

When the composition is complete and verified, the user may choose to generate the resulting composition and deploy it on a WS-BPEL engine. This will start the code generation process, which will bind the unbound partner links in the composition templates. An aspect deployment is automatically generated for the aspects contained in the composition. The Padus weaver is then employed to weave the aspects into the resulting WS-BPEL processes based on the aspect deployment specification.

A resulting composition can also be imported back into the library as a new service. The generated WS-BPEL process then serves as documentation for the new service. Apart from specifying a name and some other properties, this process is also automated.

The SCE also includes a built-in WS-BPEL engine that can be used to immediately execute a resulting composition. This feature is meant to be able

to quickly assess the result rather than to be the real deployment target. We are currently working on improving the integration of this engine, so that it can be used as a debugger for compositions by providing feedback directly to the SCE.

4 Concern-Specific Languages

Aspect-oriented principles are supported by the SCE through the use of the Padus aspect-oriented programming language. One or more aspects describe a concern, and are written in Padus. If a user of the SCE wants to express aspects, the only possibility is to specify these aspects in Padus, which requires in-depth knowledge of the Padus language. This is in contradiction with our research objective, which states that the SCE should allow the description of management concerns in an intuitive, concise and declarative manner. Therefore, we need the ability to visually specify concerns on a higher level of abstraction.

The earliest aspect-oriented programming languages are each developed for a particular crosscutting concern; we name these *concern-specific languages* (CSLs). Examples of these early CSLs are COOL [19,20], a language for expressing the aspect of synchronization for programs written in Java, and RIDL [19,20], a language for expressing the aspect of data serializability in distributed environments. A recent concern-specific language is KALA [16]. KALA is a powerful aspect language for describing the use of advanced transaction models by an application, which also allows new models to be defined if needed. However, since a new language has to be devised for each concern, construction of concern-specific languages was quickly deemed too costly. Instead, more approaches opt for general-purpose aspect languages, such as AspectJ [18].

Our objective is to enable the definition of concern-specific languages on top of the Padus technology and integrated in the SCE. The implementation of a crosscutting concern is thus defined in a specific CSL, built on Padus. To apply the concern to the service process or service composition process, a user can select the relevant aspects, add them to the service process and concretize them.

The remainder of this section contains an example of a concern-specific language: Billing. First, we define this language, and next, we show how this language is integrated in the SCE and how it can be used in a concrete service composition.

4.1 The Need for a Billing Language

Billing is a concern that occurs in many systems. It can be as simple as deducting a fixed fee from a client's account after the execution of an operation, but it can also require complicated schemes based on the client's location, the

```
<concern language="billing" type="time" name="billcall">

    <!-- specify when billing should occur: -->
    <start when="invoking(Service, Port, 'connect', User)" />
    <end when="invoking(Service, Port2, 'disconnect', User)" />

    <!-- specify what should be charged: -->
    <advice>
        <begin> <charge type="setup" context="User" /> </begin>
        <success> <charge type="time" context="User, $Time" /> </success>
        <fail> <!-- do nothing --> </fail>
        <finally> <!-- do nothing --> </finally>
    </advice>
</concern>
```

Listing 1: Billing example

client's account type, which operation was executed, how long it took, etc.

We recognize two important patterns in the billing concern. On the one hand there is the issue of *when* billing starts and ends. On the other hand there is the issue of *what* should be charged. In our approach we separate these two parts. Our dedicated Billing language selects the points in a process execution where billing starts and ends, and allows us to add extra behavior at each of these points. Typically, we pass the information about the operation and associated timestamps to a dedicated charging service. This service keeps a complete log of all charged events. At a later time, a program may collect these logs and create bills for the customers, possibly affected by business rules. This is the issue of what should be charged, and can greatly vary on the context of the events. Therefore, the Billing CSL exposes the context of the process events to a large extent.

4.2 Definition of Billing

The Billing language allows expressing billing concerns in dedicated XML-based modules, which are specified separate from the main functionality of service compositions. Listing 1 provides an example of such a module.

The main element of a Billing module is the `concern` element. Its attributes specify the language and the type of the module. In our example, line 1 specifies that the module is specified using the `billing` language, and that its type is `time`. Modules that are expressed using another concern-specific language would also contain a `concern` element, but its `language` attribute would indicate that another language is used, and that its contents are thus different than those of a Billing module.

There are three types of Billing modules: *event-based* modules are used to perform billing based on events that occur during the execution of a service (e.g., when a text message has been sent), *time-based* modules are used to perform billing based on the time that has passed between two events (e.g., between the start and the end of a telephone call), and *data-based* modules are used to perform billing based on the volume of data that has been exchanged

between two services.

The children of the `concern` element specify *when* billing should occur, and *what* should be charged. Because our example is a time-based module, it specifies both when billing should start (using the `start` element in line 4) and when billing should end (using the `end` element in line 5). The `when` attributes of the `start` and `end` elements are Padus pointcuts that select certain points in the execution of a service.

Each module specifies what should be charged in the `advice` element. This element has four children: the `begin` element specifies what should be done when the concern is activated, the `success` element specifies what should be done when the concern terminates successfully, the `fail` element specifies what should be done when an exception is thrown while the concern is active, and the `finally` element specifies what should be done when the concern terminates, regardless of whether it terminates successfully or not.

Each of these four elements may contain regular WS-BPEL code in order to perform the charging. Alternatively, one may use the Billing language's dedicated `charge` element in order to perform the charging without writing WS-BPEL code. In our example, line 9 sends a message to the charging service which specifies that the user has started a connection, and line 10 sends a message to the charging service which specifies that the user has ended a connection with a certain duration. In the advice code, variables that were bound in the Padus pointcut may be used. Additionally, we expose some context of the process by means of the `$Time` variable. In the example, the duration of the call is retrieved from the concern's context using the `$Time` variable and passed to the charging process.

In order to perform the actual charging, each Billing module contains an implicit partner link that refers to a charging service. This partner link will be employed when the `charge` element is used in the module's advice, and it can be linked to a concrete service using the SCE's interface (see Section 4.3). If one wants to use another partner link or more than one partner link, this can be specified in the optional `using` child of the `concern` element.

4.3 Visualization of Billing

In Section 3, we illustrate how the SCE can be used to create new compositions, by dragging a composition template on the canvas, filling its placeholders with services, and adding an aspect to a service. This aspect is retrieved from a repository of crosscutting concerns, and is implemented using the Padus language. Using the SCE, it is straightforward to change when such aspects are applicable or which services are used by the aspect. However, changing what the aspect actually does (i.e., changing the aspect's advice) requires in-depth knowledge of the Padus language. Therefore, the SCE also allows adding concern-specific aspects, which allow specifying an advice without in-depth knowledge of Padus.

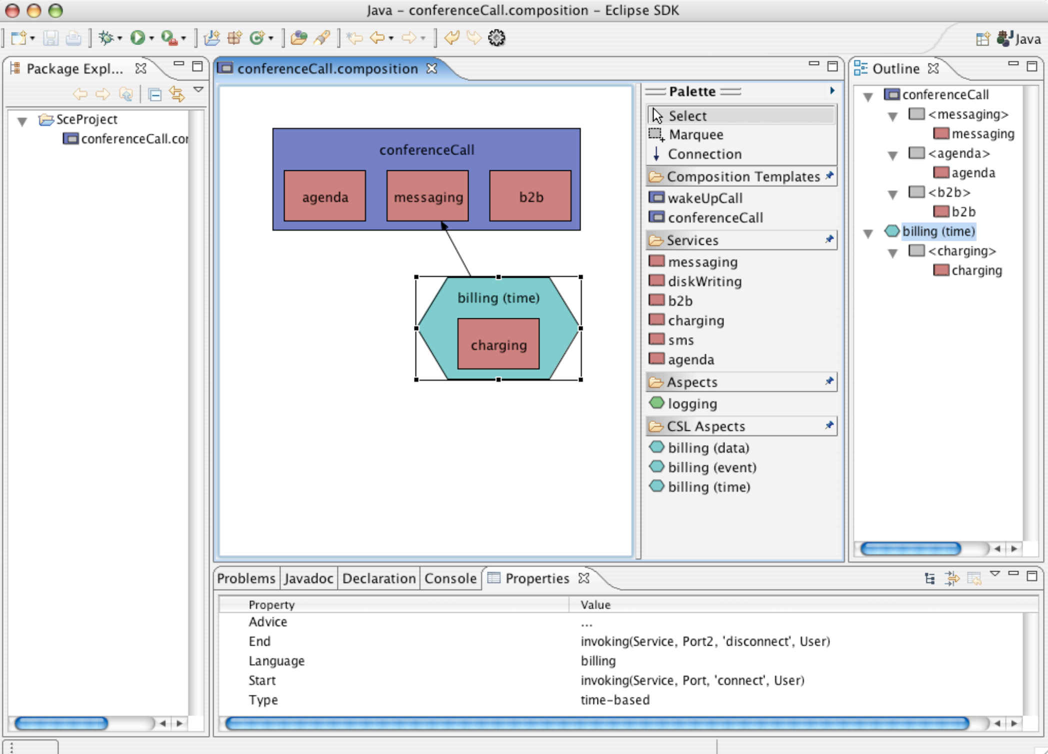

Fig. 3. A composition that contains a concern-specific aspect

Figure 3 provides an example of a composition that contains such a concern-specific aspect. The example is the same as the one in Figure 2, but the logging aspect has been replaced by a concern-specific billing aspect.

The palette contains a library of templates for concern-specific aspects, which may be instantiated by dragging them on the canvas. The palette in the example contains three such templates, i.e., "billing (time)", "billing (event)" and "billing (data)", which correspond to the three types of billing that are identified above. Each Billing aspect has at least one placeholder, which allows binding the implicit partner link that was mentioned above to a concrete web service.

When a concern-specific aspect is selected in the editor view, its properties appear in the properties view. A time-based billing aspect, for example, has five properties: "language", "type", "start", "end", and "advice". The first two properties simply show the language and the type of the aspect, respectively. The other properties, however, can be changed in order to define between which two points in the execution of the composition billing should occur, and how this billing should be achieved. Based on this information, most of the information for the corresponding Billing module (such as the one in Listing 1) is generated by the SCE.

When the composition in the editor view is complete, any concern-specific aspects are translated to Padus aspects, which are then applied to the appropriate services and/or composition templates similar to regular compositions.

5 Related Work

Several visual component composition environments already exist in the context of component-based software development (CBSD). CBSD advocates reusable and loosely-coupled components in order to realize flexible plug-and-play component composition of off-the-shelf components [31]. The main problem in CBSD is that wiring components together requires writing glue-code manually in order to resolve syntactic and semantic incompatibilities. A visual component composition environment allows to visually compose the components and supports the (semi-)automatic generation of glue-code that implements the composition. Current practice component composition environments, such as VisualAge for Java from IBM, JBuilder from Borland and BeanBuilder from Sun already allow some form of automatic glue-code generation from a given component composition. The main difference with our approach, apart from the focus on components instead of web services, is that they do not support a reusable encapsulation of composition logic. Furthermore, there is no support for verifying whether a certain composition is possible apart from syntactically checking messages and arguments. Another disadvantage is that they do not support modularizing crosscutting concerns.

Documenting components with protocol documentation is already well investigated in literature. Campbell and Habermann [9] introduced the idea of augmenting interface descriptions with sequence constraints already in 1974. More recent work includes the Rapide system [21] or the PROCOL system [33]. In the research area of component based software development, several component composition environments are available that lift the abstraction level for component composition. Yellin and Strom [38], Reussner's CoCoNut project [27] and PacoSuite [37] for example also employ automata to document components. PacoSuite is one of the most advanced component composition environments and supports higher-level component composition based on sequence charts. The main advantage with respect to the other work on protocol verification is that PacoSuite supports multi-party connectors, whereas other approaches typically only support binary connectors. The PacoSuite approach is, however, domain dependent, and is only targeted at the simple JavaBeans component model.

BPMN is a graphical notation for specifying workflows, and aims to become the de facto graphical standard similar to WS-BPEL for workflow languages. BPMN allows for a higher-level graphical notation for processes in comparison to WS-BPEL, and is in fact complementary to our approach. A BPMN-based editor that is able to import/export WS-BPEL can for instance be used to edit the specification of a composition template. As soon as there is a standardized file format for BPMN, the SCE can also directly support BPMN for the documentation of services and composition templates, instead of or next to WS-BPEL.

Several approaches exist that focus on the construction of concern-specific

languages, also referred to as domain-specific languages. Note that these languages do not have facilities for encapsulating crosscutting concerns and are hence not aspect languages. Such approaches provide environments for the more efficient and scalable construction of languages fit to express concepts from a particular domain. Examples are Draco [23], GenVoca [5], Babel [8] and Intentional Programming [28].

Agarwal *et al.* [1] present a service creation environment based on end-to-end composition of web services, but this environment does not allow visual composition of web services nor separation of concerns using aspect-oriented techniques.

6 Conclusions and Future Work

In this paper, we present a high-level service creation environment for composing web services. Our approach supports the modularization of crosscutting concerns through Padus aspects. Padus aspects can be visually deployed onto composition templates or services. Furthermore, support for concern-specific languages on top of Padus is available.

On the abstraction scale we situate our SCE on the same level as BPMN and the Padus language. The SCE is an advanced tool for configuring service compositions and augmenting them with separated concerns, by means of Padus aspects or higher-level concern-specific languages.

Our work is still in an early phase and as such several improvements are possible:

- Our approach supports visually deploying aspects onto concrete services. The pointcuts still have to be defined programmatically in Padus. Describing pointcuts at a higher level of abstraction would be an important contribution to our work. We are experimenting with existing pointcut visualizations such as Theme/UML [11], Join Point Designation Diagrams [30,29] and AOSF [22] to solve this problem.
- It is possible that an aspect adapts the external protocol of an existing service (e.g., by adding an invocation) so that it becomes incompatible with the composition template's protocol. Currently, our tool is not able to cope with this problem. In order to solve this, we are planning to include the aspect protocol documentation and verification algorithms proposed by *composition adapters* [34].
- The support for integrating concern-specific languages is currently quite limited. Apart from a set of common tool (such as XML parsing and transformation tools) and a simple visualization template, defining and implementing a new concern-specific language still largely happens in an ad hoc manner. A more in-depth solution based on existing work (such as Babel) is subject to future work.
- The repository of available composition templates, services and aspects is

a custom solution and limited to local files. In the future, we plan to investigate support for the industrial standard for discovery of web services called UDDI [32].

Acknowledgments

This research is partly funded by Alcatel Belgium and the Institute for the Promotion of Innovation Through Science and Technology in Flanders (IWT-Vlaanderen) through the WIT-CASE project.

References

[1] Agarwal, V., K. Dasgupta, N. Karnik, A. Kumar, A. Kundu, S. Mittal and B. Srivastava, *A service creation environment based on end to end composition of web services*, in: *Proceedings of the 14th International World Wide Web Conference (WWW 2005)* (2005), pp. 128–137.

[2] Alonso, G., F. Casati, H. Kuno and V. Machiraju, editors, "Web Services: Concepts, Architectures and Applications," Springer-Verlag, Heidelberg, Germany, 2004.

[3] Andrews, T., F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic and S. Weerawarana, *Business Process Execution Language for Web Services, version 1.1* (2003).
URL `http://www.ibm.com/developerworks/library/ws-bpel/`

[4] Arsanjani, A., B. Hailpern, J. Martin and P. Tarr, *Web services: Promises and compromises*, Queue **1** (2003), pp. 48–58.

[5] Batory, D., V. Singhal, J. Thomas, S. Dasari, B. Geraci and M. Sirkin, *The GenVoca model of software-system generators*, IEEE Software **11** (1994), pp. 89–94.

[6] *BPWS4J*.
URL `http://www.alphaworks.ibm.com/tech/bpws4j`

[7] Braem, M., K. Verlaenen, N. Joncheere, W. Vanderperren, R. Van Der Straeten, E. Truyen, W. Joosen and V. Jonckers, *Isolating process-level concerns using Padus*, in: *Proceedings of the 4th International Conference on Business Process Management (BPM 2006)* (2006), (to appear).

[8] Brichau, J., "Integrative Composition of Program Generators," Ph.D. thesis, Programming Technology Lab (PROG), Vrije Universiteit Brussel, Brussels, Belgium (2005).

[9] Campbell, R. and A. Habermann, *The specification of process synchronisation by path expressions*, in: *Proceedings of an International Symposium on Operating Systems*, 1974, pp. 89–102.

[10] Charfi, A. and M. Mezini, *Aspect-oriented web service composition with AO4BPEL*, in: L.-J. Zhang, editor, *Proceedings of the 2nd European Conference on Web Services (ECOWS 2004)* (2004), pp. 168–182.

[11] Clarke, S. and E. Baniassad, "Aspect-Oriented Analysis and Design — The Theme Approach," Addison-Wesley, 2005.

[12] Cottenier, T. and T. Elrad, *Dynamic and decentralized service composition with Contextual Aspect-Sensitive Services*, in: *Proceedings of the 1st International Conference on Web Information Systems and Technologies (WEBIST 2005)*, Miami, FL, USA, 2005, pp. 56–63.

[13] D'Hondt, M. and V. Jonckers, *Hybrid aspects for weaving object-oriented functionality and rule-based knowledge*, in: K. Lieberherr, editor, *Proc. 3rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2004)* (2004), pp. 132–140.

[14] Du, W. and A. Elmagarmid, *Workflow management: State of the art vs. state of the products*, Technical Report HPL-97-90, Hewlett-Packard Labs, Palo Alto, CA, USA (1997).

[15] *The Eclipse platform*.
URL http://www.eclipse.org/

[16] Fabry, J., "Modularizing Advanced Transaction Management — Tackling Tangled Aspect Code," Ph.D. thesis, Programming Technology Lab (PROG), Vrije Universiteit Brussel, Brussels, Belgium (2005).

[17] Gybels, K. and J. Brichau, *Arranging language features for pattern-based crosscuts*, in: M. Akşit, editor, *Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003)* (2003), pp. 60–69.

[18] Kiczales, G., E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. G. Griswold, *An overview of AspectJ*, in: J. L. Knudsen, editor, *Proc. ECOOP 2001, LNCS 2072* (2001), pp. 327–353.

[19] Lopes, C. V., "D: A Language Framework for Distributed Programming," Ph.D. thesis, College of Computer Science, Northeastern University (1997).

[20] Lopes, C. V. and G. Kiczales, *D: A language framework for distributed programming*, Technical Report SPL-97-010, Palo Alto Research Center (1997).

[21] Luckham, D., J. Kenney, L. Augustin, D. Vera, D. Bryan and W. Mann, *Specification and analysis of system architecture using Rapide*, IEEE Transactions on Software Engineering **21** (1995).

[22] Mahoney, M., A. Bader, T. Elrad and O. Aldawud, *Using aspects to abstract and modularize statecharts*, in: O. Aldawud, G. Booch, J. Gray, J. Kienzle, D. Stein, M. Kandé, F. Akkawi and T. Elrad, editors, *The 5th Aspect-Oriented Modeling Workshop In Conjunction with UML 2004*, 2004.

[23] Neighbors, J. M., *Draco: A method for engineering reusable software systems*, in: *Software Reusability — Concepts and Models*, ACM Press, New York, NY, USA, 1989 pp. 295–319.

[24] *Oracle BPEL Process Manager*.
URL
http://www.oracle.com/technology/products/ias/bpel/index.html

[25] Ossher, H. and P. Tarr, *Using subject-oriented programming to overcome common problems in object-oriented software development/evolution*, in: *Proc. 21st Int'l Conf. Software Engineering* (1999), pp. 687–688.

[26] Parnas, D. L., *On the criteria to be used in decomposing systems into modules*, Comm. ACM **15** (1972), pp. 1053–1058.

[27] Reussner, R. H., *Automatic component protocol adaptation with the CoCoNut tool suite*, Future Generation Computer Systems **19** (2003), pp. 627–639.

[28] Simonyi, C., *The death of programming languages, the birth of intentional programming*, Technical report, Microsoft, Inc. (1995).

[29] Stein, D., S. Hanenberg and R. Unland, *Query models*, in: *UML '04: Proceedings of the international conference on the Unified Modelling Language* (2004), pp. 98–112.

[30] Stein, D., S. Hanenberg and R. Unland, *Expressing different conceptual models of join point selections in aspect-oriented design*, in: *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development* (2006), pp. 15–26.

[31] Szyperski, C., "Component Software: Beyond Object-Oriented Programming," ACM Press and Addison-Wesley, New York, NY, USA, 1998.

[32] *UDDI*.
URL http://www.uddi.org/

[33] van den Bos, J. and C. Laffra, *PROCOL: A concurrent object-oriented language with protocols delegation and constraints*, Acta Informatica **28** (1991), pp. 511–538.

[34] Vanderperren, W., D. Suvee and V. Jonckers, *Combining AOSD and CBSD in PacoSuite through invasive composition adapters and JAsCo*, in: *Proceedings of the Net.ObjectDays 2003 International Conference*, 2004, pp. 35–50.

[35] Verheecke, B., W. Vanderperren and V. Jonckers, *Unraveling crosscutting concerns in web services middleware*, IEEE Software **23** (2006), pp. 42–50.

[36] White, S. A., *Business Process Modeling Notation (BPMN), version 1.0* (2004).
URL http://www.bpmn.org/

[37] Wydaeghe, B., "PacoSuite: Component Composition Based on Composition Patterns and Usage Scenarios," Ph.D. thesis, System & Software Engineering Lab (SSEL), Vrije Universiteit Brussel, Brussels, Belgium (2001).

[38] Yellin, D. M. and R. E. Strom, *Protocol specifications and component adaptors*, ACM Transactions on Programming Languages and Systems **19** (1997), pp. 292–333.

Model Driven Development of Security Aspects

Julia Reznik, Tom Ritter

{julia.reznik,tom.ritter}@fokus.fraunhofer.de
Fraunhofer Institute FOKUS, Kaiserin-Augusta-Allee 31,
10589 Berlin, Germany

Rudolf Schreiner, Ulrich Lang

{Rudolf.Schreiner,Ulrich.Lang}@objectsecurity.com
ObjectSecurity Ltd., St John's Innovation Centre, Cowley Road,
Cambridge, CB40WS, United Kingdom

Abstract

The development of security-critical large-scale distributed software systems is a difficult and error prone process. As we learnt from practical experiences, it is especially difficult to manually define security policies, for example for access control. A human security administrator is not able to cope with the high complexity of the interactions of the application and the low level, platform specific security policy. Therefore, a new approach is needed to ease the definition of appropriate security policies. This paper shows how realisation of security aspects of a system can be automated to a great extend by applying model-driven software development techniques not only on functional properties. In the presented approach, UML models of the application's functional properties are flexibly augmented with security relevant information. Together with a high level security policy defined by the security administrator, this augmented functional model is then used in an automatic model transformation to generate the platform specific security policy. With this approach, which supports the separation of concerns in model based software engineering, we can automatically generate security-critical applications for different middleware platforms like SecureMiddleware, which is an extended implementation of the CORBA Component Model with improved support for non functional properties like security. The concepts, platforms and tools presented in the paper are currently used for the development of several large-scale and secure applications, for example for building a Virtual Air-Space Management System with strong security requirements.

Key words: Security Aspects, MDA, Roles

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

1 INTRODUCTION

Model Driven Development (MDD) turned out to be most useful for the development of standard business applications. Many existing MDD solutions and tools support this domain. However, the handling of the non-functional aspects like Quality of Service, adaptability, assurance and security are mostly not sufficiently covered. In standard business applications, this is good enough; the existing MDD solutions are well suited to describe the structural parts of an application, e.g. components, classes or interfaces. But often these solutions lack in easy to use support of non-functional aspects.

Security, which is our main concern in this paper, is mainly implemented directly at the platform level, e.g. by configuring roles. If this is not sufficient, e.g. if more complex security policies have to be enforced, it has to be done as part of the implementation of the business code. This, of course, is a large obstacle to one of the main principles of component based software development, namely component reusability. In such cases the component implements not only its business functionality, but also a hard coded security policy. The component can only be reused if both the functional and the non-functional requirements match.

In recent years, more advanced middleware platforms became available, which were specifically tailored to meet the demanding requirements like adaptability, flexibility, robustness and security (e.g. based on CORBA Component Model). In contrast to standard business platforms, these new advanced platforms also offer support for a variety of non-functional aspects and allow separation of functional and non-functional aspects at implementation level. Some of them follow a container or capsule paradigm, which means the business functionality is implemented in a component implementation, the non-functional aspects such as access control rules are handled by the container, the component runtime environment.

Integrating MDA, UML and security is not a new approach. Jrjens defined UMLSec as an extension to UML, in order to model and verify secure systems, while we use the functional model for the generation of security policies. Our approach is more similar to Lodderstedt's SecureUML. In SecureUML, UML is extended by Role Based Access Control (RBAC). SecureUML allows an extension of UML models by access control rules and provides a direct mapping to a middleware supporting RBAC. We use similar concepts, but SecureMiddleware is not limited to a single security model like RBAC. In SecureMiddleware, we use a more flexible policy model and evaluator, and are therefore able to support other standard security models as well, for example Mandatory Access Control, or to freely define security policies as rules on attributes. While SecureUML is very well suited for standard business applications, where RBAC is the dominating security model, our far more flexible approach is required for applications in domains like defense, air traffic control or ubiquitous computing.

Since the MDA approach improves the overall software development process so significantly, it would be most beneficial to apply it also to the development of applications based on these advanced middleware platforms, with particular emphasis on the non-functional aspects. Therefore, in this paper we describe the integration of security as non-functional aspect into the overall MDA development process and tool chain, which is used to build a virtual airspace management system with strong security requirements, with respect to access control.

The paper is outlined as follows: Section 2 described the *SecureMiddleware* platform, which is our target platform for the development of secure applications. Section 3 gives an overview on how MDA principles are applied to functional and non-functional aspects.Section 4 explains the structure of the model-based tool chain we have built to support development of secure applications in a platform independent way. Section 5 describes an example on how we apply the presented approach in the air traffic management domain. Section 6 concludes the paper.

2 SECURITY IN COMPONENT BASED APPLICATIONS

2.1 SecureMiddleware

Although in model-based development environment the specific properties of the target platform become less important, a reliable and efficient execution environment is still crucial for successful system development. The CORBA Component Model (CCM) [1] defines a platform which is a good choice for developing large scale distributed systems. It is based on the CORBA middleware and adds some more advanced concepts, e.g automated deployment. It also simplifies the usage of some CORBA Services. The *SecureMiddleware* platform [2] we use for realizing systems consists of an implementation of the CORBA Component Model called Qedo [3], an extended Security Framework called OpenPMF [4,9] and both are based on top of MICO, a CORBA ORB with enhanced security functionality [11].

CORBA is well accepted by industry in mission critical application areas like Air Traffic Control, which is our target domain, because it is a reliable and mature technology and many interoperable implementations of good quality are available.

CCM enhances the Object Model of CORBA. Figure 2 depicts the features a CORBA Component can have. A component has a component interface (equivalent interface) and a component home. The equivalent interface provides operations for introspections and navigation regarding other components features; the home provides operations to manage component life cycles and must be declared for every component declaration. A component can provide a set of facets. A facet is a named port providing a specific interface. Clients

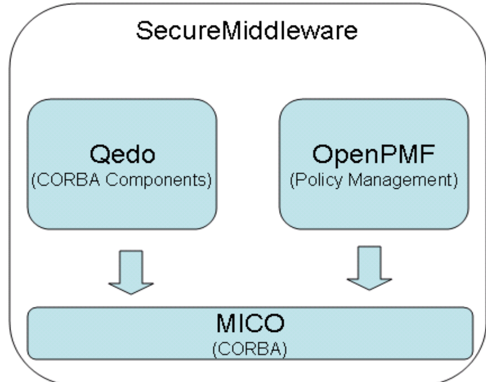

Fig. 1. SecureMiddleware platform

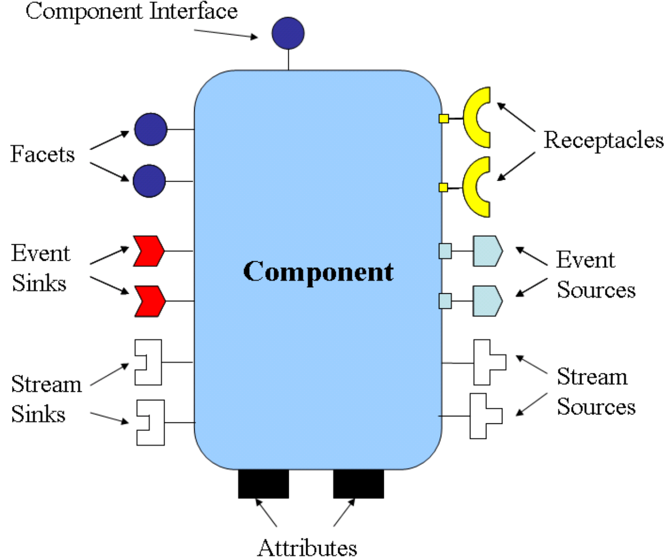

Fig. 2. Object Model of CCM

of this component call operations on a facet. The facet's counterpart, a receptacle, is a named port where a specific interface can be connected to. A facet of a CORBA Component in server role can be connected to a receptacle of a CORBA Component in a client role. Receptacle ports make dependencies to other interfaces explicit, which helps to minimize wrong configurations and run-time failures by providing type safety.

As facets and receptacles are used for operational interactions (method invocations from other components synchronously), the event sources and event sinks are used for event based interactions and message exchange (exchange event messages with other components asynchronously). An event source can publish or emit events of a certain type. Event sinks can consume events of a certain type. A similar port concept for continuous interactions (i.e. data streams) is lately introduced by the OMG to the CORBA Component Model. A stream source port produces streams of data of a specific type while a

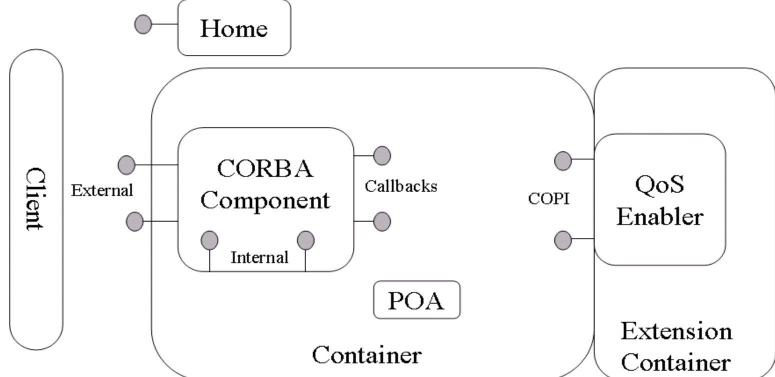

Fig. 3. QoS Enabler in extension container

stream sink port can receive such data. Attributes can be used to configure an instance of a CORBA Component.

The CORBA Component Model has defined the container model for providing a high level of abstraction to the component implementation. It also offers the possibility to load and to unload user code (components) dynamically by installing or de-installing Homes. Depending on mechanism used by the container vendor and programming language this is realized by loading and unloading shared libraries and it requires sophisticated management of such artifacts at run-time. This is facilitated by the fact that the interfaces between the component implementation and the container are standardised. The component implementation offers a specific set of interfaces to the container allowing it to manage the component implementation. These interfaces are called Callbacks. On the other side the container offers a set of interfaces to the component implementation which enables the component implementation to make use of a certain container services. These interfaces are called Internal interfaces.

An important feature of Qedo, which makes it in particular interesting for aspect orientation, is the possibility to extend the functionality of a system or of a component without modifyiing implementation code but by installing QoSEnablers in the run-time environment. A QoS Enabler is a specialized component that can be loaded into a specialized CCM container and is able to hook in additional functionality. Taking this approach allows usage of plain CCM mechanisms for development and deployment of QoS Enablers (see figure 3). Each QoSEnabler is responsible for a specific QoS category. In our case, a QoS Enabler is used to handle security aspects. The QoSEnabler concept is currently in final stage of standardisation at the OMG [16]. QoSEnabler may use an interception pattern to provide their functionality. To make use of it a QoSEnabler registers interception interfaces at the container. This interfaces are called Container Portable Interceptors (COPI). A more detailed description of these platfrom specifics can be found in [5].

In the SecureMiddleware we used the QoS Enablers to enforce Access Control policies. To make this possible on each relevant node a corresponding QoS

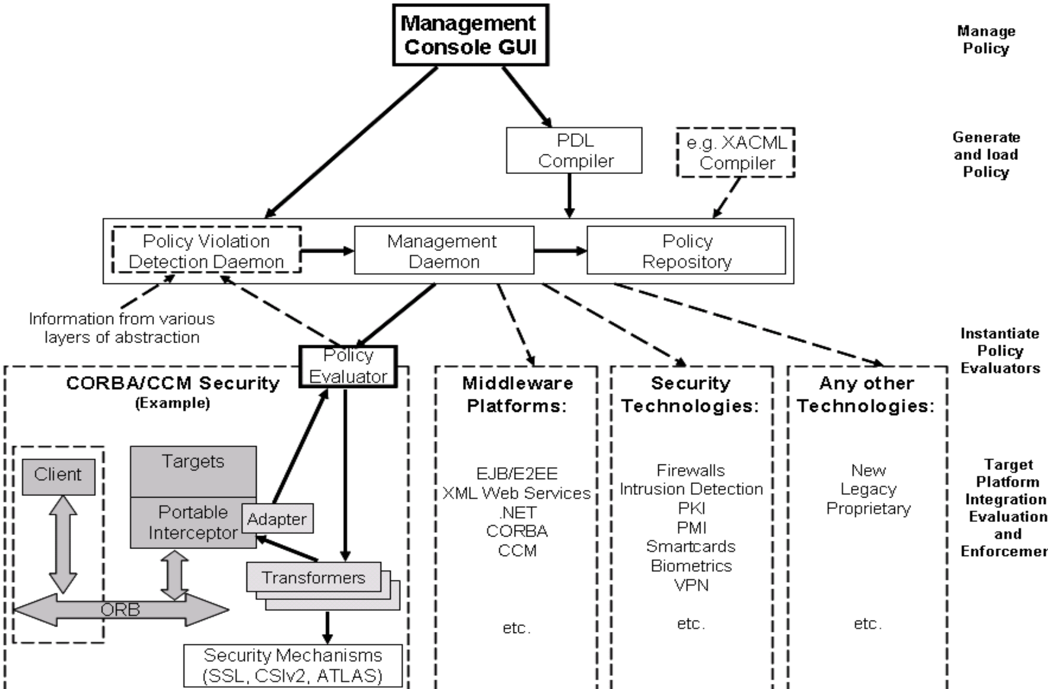

Fig. 4. OpenPMF Architectural Overview

Enabler is instantiated in the CCM run-time environment. The management of the security policies is accomplished by a policy management framework called OpenPMF. QoS Enablers are in contact with OpenPMF at run-time to get updates on the security policies that have to be enforced.

2.2 Policies and Policy Evaluation in OpenPMF

The OpenPMF Policy Management Framework was developed for the definition, management and enforcement of security policy in large scale distributed systems. Figure 4 shows an architectural overview of the OpenPMF framework. When the framework is initiated, the technology-neutral policy, written in a policy definition language (PDL), is loaded into a central policy repository. It is then obtained by the different systems, servers or applications and transformed into an efficient internal representation optimised for the evaluation of abstract attributes obtained from the underlying security technology and platform. At runtime, each incoming invocation triggers an evaluation process, after which the resulting decision is enforced on the particular underlying platform. In SecureMiddleware, the policy evaluation is done in an OpenPMF QoS Enabler for the components and in CORBA Portable Interceptors.

OpenPMF is administered through the management daemon and the management GUI. In addition, the policy violation detection daemon collects relevant information from various layers of the underlying IT infrastructure and detects violations of the security policy

The current way of specifying the policy is by using our human-readable, technology-independent Policy Definition Language (PDL), which supports different security models. PDL uses concepts of the Principal Calculus [12] which theorises about principals and its two different privilege delegation relations.

PDL supports rules that are expressed in terms of requests and replies (i.e. initiating invokers, intermediaries, actions, targets etc.). Some of the features supported by the language are wildcards, multiple sets, several (arbitrary) actions, sets of clients/targets, groups and roles, and hierarchical nesting. PDL also provides advanced support for delegation. The following is a short example of a security policy definition using PDL, which allows the usage of 3 operations of the `Account` interface for a client with the name `TestUser`:

```
policy /OS [*, *] {
  policy /OS/Bank [*, *] {

      (client.name == TestUser)
      &(operation.name == deposit)
      &(target.type ==  IDL:Account:1.0) : allow;

      (client.name == TestUser)
      &(operation.name == balance)
      &(target.type ==  IDL:Account:1.0) : allow;

      (client.name == TestUser)
      &(operation.name == withdraw)
      &(target.type ==  IDL:Account:1.0) : allow;
  };
};
```

In this small example of protecting CORBA interfaces, the definition seems to be simple, because of the simplicity of the demo application. In complex real world applications the security policies very quickly become long and complex, e.g. the policy for a small CCM application is about 500 rules. While a single rule is still simple, the vast number of rules makes writing them manually and later maintaining them almost impossible. There are also dependencies between the rules, e.g. between rules for protection at a server and a domain boundary, which makes the manual rules definition even more difficult for humans. Defining security aspects of an application by writing policies for access control in PDL also requires specific knowledge of the platform specific details, to cover the internal communication of the platform, e.g. for the deployment of components. With the work we presented in this paper we want to benefit from the Model Driven Architecture to greatly reduce the complexity of writing security policies, i.e. by doing it in a platform independent way and with a higher level of abstraction.

3 APPLYING MDA ON FUNCTIONAL AND NON-FUNCTIONAL ASPECTS

Why is the manual development of security policies so difficult, even for skilled security and middleware specialists? The first reason is the overall complexity of the systems, both of the user components and the platform internal communications. The second reason is that fulfillment of security requirements of the system is often done during the later system development phases; security is typically integrated into the resulting system in a post-hoc manner. Therefore, to make security manageable, we need to reduce the system complexity the persons in charge of security have to deal with and to improve the whole development process by providing approaches, languages and tools. Using model-centric and generative MDA approach for development of "security" systems brings following advantages:

- Abstraction and reduction of complexity: Security policies can be defined and integrated into system designs at a high level of abstraction; the human has only to make the high level decisions and the "hard work" is done by transformers.
- Well defined and structured procedures help to avoid overseeing something, which is in our experience more dangerous than making wrong decisions.
- High level models can be used to detect and correct design errors early in the deveopment process.

The main goal of our work was to provide a MDA tool chain that supports the rapid model based development process of security-critical software systems: starting from definition of "secure" system models in high-level modelling languages like UML, then transform them automatically into "secure" target system models like CCM that can be used for further steps like code generation of CCM components with security properties or plain security policy generation. In essence, apply MDA principles to functional and non-functional aspects in parallel.

The MDA tool chain and its components are described in the section 4. The idea was not to implement the whole tool chain, but to select most suitable existing modelling and other tools used during the development process with target platforms and combine these tools via model repositories in one open integrated environment. The implementation work that should be done for the tool chain is to reilize just model transformers and profiles to make it work. Artefacts like metamodels, profiles and transformers play very importand role for our tool chain realization. Metamodels provide means for management of models: all created models are stored in repositories: in our tool chain repositories are automatically generated from defined metamodels.

To support visual modelling of domain-specific aspects domain-specific languages or profiles are needed. To achieve the integration of different modelling techniques and for different modelling layers (PIM, PSM), the different repos-

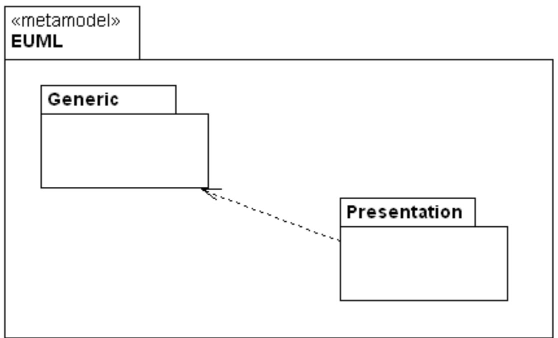

Fig. 5. Overview of the eUML metamodel

itories are interconnected together by specific model transformers, which map models to other models or to a programming or domain-specific language code.

As already mentioned, for system and security policy design we use the Unified Modeling Language (UML) [13,14] as the foundation in our work. In fact, UML is the standard for object-oriented modeling, many modeling tools support UML and a great number of developers use the language.

Since UML 2 as a whole is a language with a very broad scope and has a less precisely semantic definition (when it shall be used for automatic transformation or code generation) we have defined a subset of UML 2 together with a specialized and absolutely clear semantics. This subset we called eUML (essential UML). The eUML metamodel includes only the required UML 2 metamodel elements which formally define modeling elements, their semantic and relations. The eUML metamodel contains two main packages: Generic and Presentation as shown in figure below. The first package covers the generic modelling part (based on UML 2.0 metamodel) and the second package includes additional concepts, which define graphical modelling information of an UML element and UML diagram.

The Generic package contains the basic UML2 concepts grouped into the separate packages according to their nature. To avoid the "package merge" overhead found in the UML 2 specification, most of the generic UML 2 concepts (for example `Class`) are represented by a single element, rather than hierarchy of multiple elements having the same name and spread out in different packages. This way seems to be most practicable and is applied in the eUML metamodel throughout.

The Generic package contains following UML2 packages: Kernel, Components, CompositeStructures, UseCases, CommonBehaviours, Actions, Activities and Relationships. Figure 6 shows the dependency relationship of the packages.

The Presentation package based on UML 2 (Generic package) and extends metaclasses like `Element` or `NamedElement` to additional graphical information such as element dimensions, positions etc. The advantage of this extention is the ability to store not only eUML model elements into repository, but also

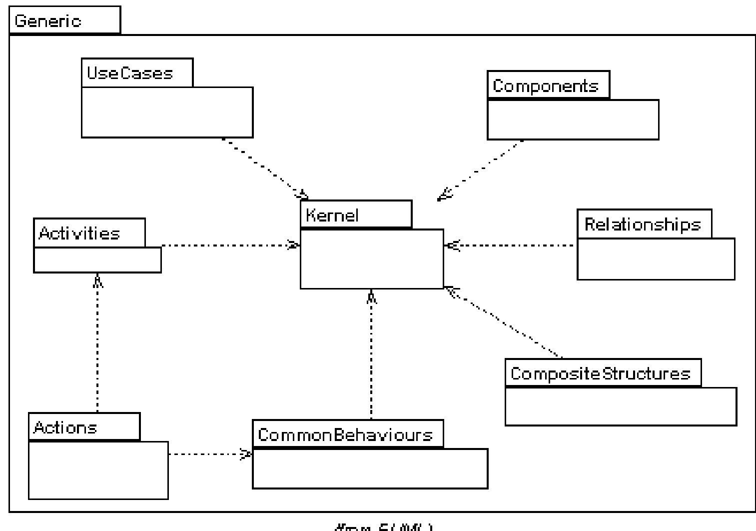

Fig. 6. Overview of the eUML Generic package content

complete diagramms with all graphical information.

Similar to UML, the eUML supports user-defined UML profiles. We can extend the eUML metamodel by using profiles to customize the language for particular domain like security. For modeling our security policies we have defined a small security profile for eUML called eUMLSec. By using this profile it is possible to model the security aspect of a system additionally to the system design, for example Role Based Access Control (RBAC)[1]. A role is a job or function within a system. The role comprises all operations on a set of targets that can be executed from the user (person or system component).To reflect this concept of a role, the profile definition includes a stereotype `<<Security_Role>>` that extends the eUML metaclass Class. In the section 5 we will give an example of the secure-aware system design.

With eUML we model our target security-aware system at a high level of abstraction, the implemented transformations are responsible for the generation of security infrastructure for platform that supports RBAC and CCM. We need a well-defined metamodel for this platform to be able to transform eUML models (PIM) into *SecureMiddleware* models (PSM). Since the *SecureMiddleware* platform is an implementation of the CCM and OpenPMF, we have extended the CCM metamodel [1], which is already defined and standardised by the OMG. We have added security concepts like `RoleDef` and `PermittedOperation`. We call this extended metamodel *CCMSec*:

The metaclass `RoleDef` inherits from the metaclass `Container` and contains `PermittedOperation` that user with this role are allowed to execute. In our approach we design also an initial configuration of component instances

[1] For the sake of simplicity, we use RBAC as an example. For other policies, like Mandatory Access Control, we need additonal policies at the client side as well. Here another set of transformation is used.

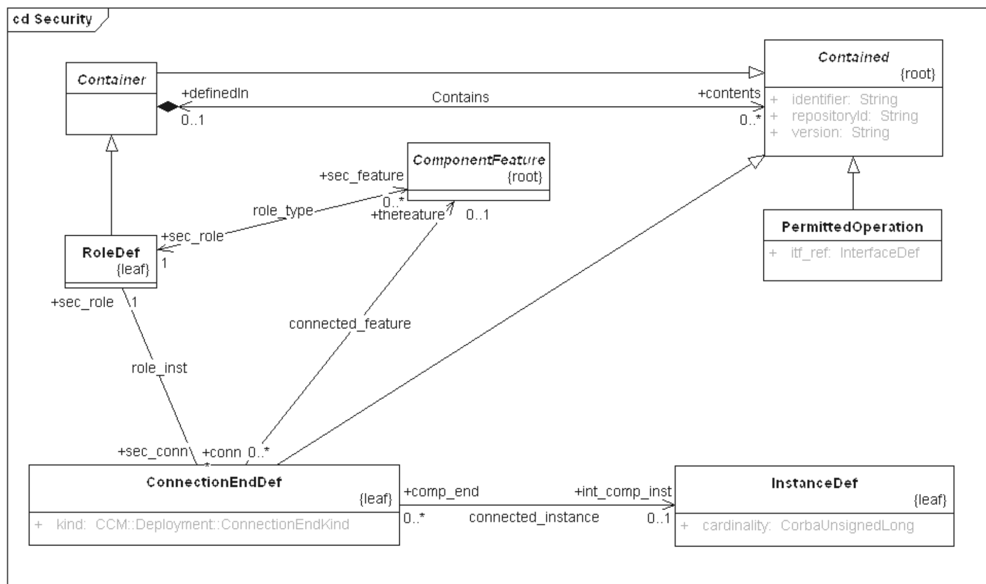

Fig. 7. CCM metamodel extension for security

of an application, which is called assembly. An assembly describes the initial configuration of the application at run-time, it defines which component instance (metaclass `InstanceDef`) to use, how many and how to interconnect them to each other. We defined an assembly as UML collaboration with component instances (see example in section 5). Each component instance has ports defined by corresponding component (-type). To the ports (metaclass `ConnectionEndDef`), we added the required security information (each port is represented as "port name/role", where the "role" of the incoming port gives the required role for the invocation).

The transformation defined in our approach consists of multiple steps, because the overall security policy of a component can be separated into two distinct parts.

- **Infrastructure.** The transformation starts with the implicit generation of the security roles for the component infrastructure. This includes activities like deployment and configuration of component instances. This has to be done independently from the application security policies. In pratice, these fully automatically generated roles cover the largest part of the security policy.
- **Component Instances.** The next step is the transformation of security information modelled in component assemblies. For this we derive from every association of a role to a port of a component instance a security rule, which allows the comunication on this component instance port for client which are in this role.

For deployment process the policies resulting from the transformation are mapped to the concrete platform, with now concrete security attributes like X.509 DN describing a component.

4 MDA TOOL CHAIN

Our MDA tool chain has been produced for the rapid model based development of CCM based security-critical software systems based on *SecureMiddleware*. As already mentioned, it supports a platform independent modelling of systems, there is no need to model, e.g., platform-specific data types, or to decide early in the development process which particular platform to use. By using transformers, PIM models can be automatically transformed into PSM models which then can be used for further steps (e.g. C++code or security policy generation). The tool chain is a set of modeling and other development tools with adaptations to additionally support the security aspects; its architecture is shown in figure 8.

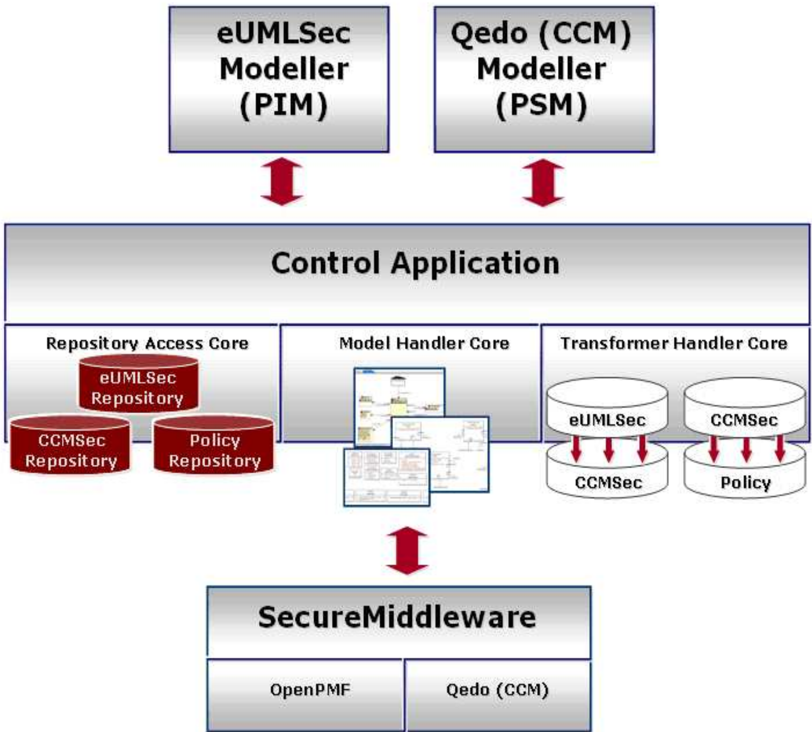

Fig. 8. Overview of the tool chain architecture

As already mentioned, we start modeling with the eUMLSec and use the eUMLSec Modeller Tool, which is a Plug-In implementation for Sparx Enterprise Architect [15]. This Plug-in is synchronized with the eUMLSec repository in both directions (load and store of eUMLSec models), including the synchronization of graphical information of the models stored in the repository. Furthermore, the eUML Plug-in contains dialogs and wizards specific for the eUMLSec language to specify details of language elements. At this level, the model shall not contain any specific platform dependent information.

The next step in the tool chain is to transform designed model into the platform specific model, in our example the *SecureMiddleware* platform. As

already mentioned in this paper, *SecurityMiddleware* is an extended CCM implementation provided by Qedo with security support provided by OpenPMF. Thus, we have two additional repositories for further transformation steps: The CCMSec repository for the management of platform specific information, and the Policy repository for management of policies defined in the platform. These repositories are synchronized with Qedo and OpenPMF and are parts of the tool chain architecture.

The heart of our tool chain architecture is a generic control application component which is used to manage and control the various components of the tool chain. The whole management of repositories, transformers and models (serialize them to the file system, incarnate them from the file system, import and export them to and from centralized shared repositories, transform them to other models) is done via the control application GUI. In the standard configuration of the tool chain control application loads three repositories: the eUMLSec, the CCMSec and Policy, and two transformers: The eUMLSec2CCMSec and CCMSec2Policy. The list of loadable tool chain components can be arbitrarily extended, we also can support other specific technologies or platforms (e.g. J2EE) and transform eUMLSec models to new platform specific models. Such extensions require at least two new tool chain components: a platform-specific repository and a transformer for eUMLSec to platform-specific transformation.

The eUMLSec2CCMSec transformer transforms an eUMLSec model from the eUMLSec repository into the CCM based specific model with security information. This transformed model will be stored in the CCMSec repository after the transformation has been done. The second transformer in the tool chain transforms CCMSec models into the Policies stored in the Policy repository. After the last transformation, the OpenPMF Policy Evaluators, here embedded in QoS Enablers and CORBA Portable Interceptors, can obtain the security policy from the policy repository and enforce it at runtime.

CCMSec models can be modified or completed (e.g. by adding deployment information or other non-functional aspects like QoS) by using the Qedo Modeller Tool implemented as an Eclipse Plug-In. The Qedo Modeller handles the connection to the CCMSec repository to propagate models into the repository and also load models from the repository and display them graphically.

From CCMSec models Qedo Modeller is able to generate the pre-generated implementation skeletons, deployment descriptors and security policy description (PDL) for the CCMSec model. After completing the business code implementation of pre-generated components the tool chain finally creates component and assembly packages and loads generated policies into the central policy repository of OpenPMF. After this step created components are immediately deployable and executable on top of the Secure Middleware platform.

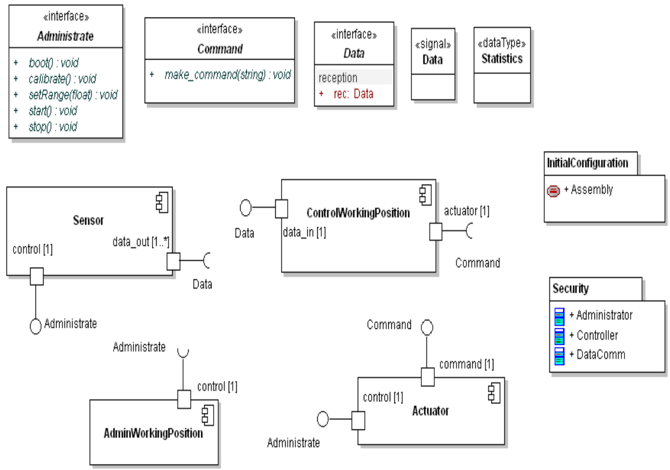

Fig. 9. Extract from the ATM system design

5 USING THE APROACH IN THE AIR TRAFFIC MANAGEMENT DOMAIN

By applying MDA in our tool chain, we are able to generate automatically platform specific security policies out of platform independent models. We present a simplified and small fraction of a "Secure_ATM" (Air Trafic Management) system we are currently working on to demonstrate this. The example defines four components: Sensor, ControlWorkingPosition, AdminWorkingPosition and Actuator as outlined in figure 9.

The Sensor component represents a radar station which sends data (Data) to the ControlWorkingPosition component. The Sensor component can be controlled and managed via the interface Administrate provided by the port control. The ControlWorkingPosition collects and presents the data to flight controllers, for example. The Actuator component represents an example actuator that communicates with the ControlWorkingPosition component via interface Command. The AdminWorkingPosition component manages components via the interface Administrate. Using mentioned above profile definition for Security we defined following roles as stereotyped with `<<Security_Role>>` UML classes (figure 10).

Fig. 10. Roles overview

The Controller role includes all (_all_) operations from the interface Command, the role gives a permission to execute operations of the Command interface. The same principle is for the role Administrator that includes all operations of the interface Administrate, and the role DataComm enables

Data signal based communication. It is of course also possible to select certain operations on a set of targets.

As outlined in section 3 the definition of an assembly on PIM level is done by using UML collaboration diagram. Figure 11 shows one configuration of our example application with some security information attached to it. In particular, the roles in which the component instances interact which each other are attached to the ports of the component instances. This collaboration diagram gives sufficient information for the generation of securitypolicies which are enforced by the *SecureMiddleware* platform.

The tool chain also automatically transforms the eUML component definitions into CCM specific implementation skeletons, which allows the implementation of the business code of the components. And also starting from the collaboration diagram the tool chain generates deployment descriptors and component packages which are later used for the automatic deployment of the application.

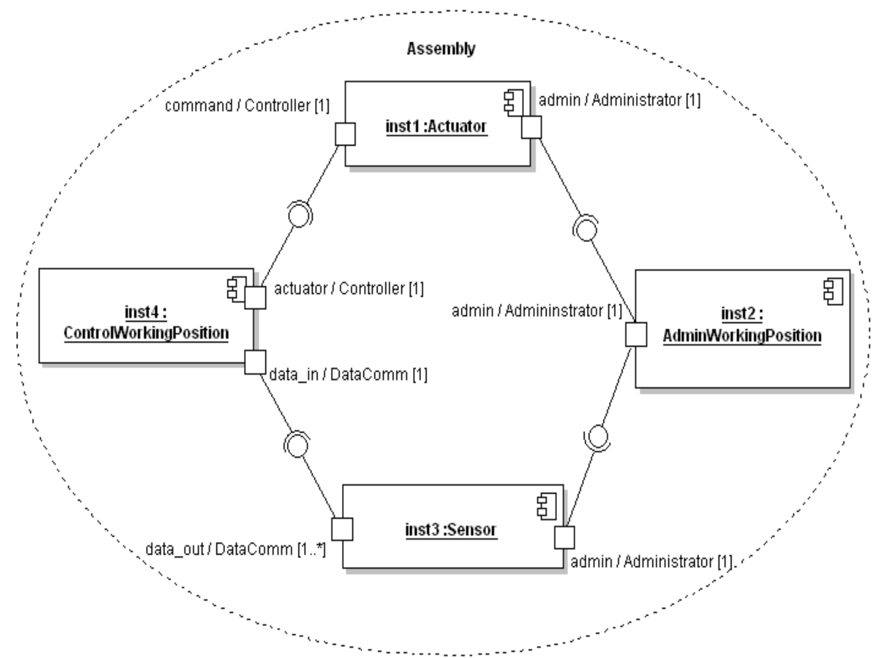

Fig. 11. Example Assembly

6 CONCLUSION

The Model Driven Architecture greatly improved the development of large scale distributed applications. In this paper we have described how the concepts of Model Driven Development are applied to build a security aware tool chain, which allows the platform independent definition of Air Traffic Management systems in conjunction with definition with security policies. Both, the design and the operation of such systems greatly benefited from such a se-

curity aware MDD based tool chain, i.e. more rapid development and a higher level of safety is possible. The tool-supported definition of security policies at a high level of abstraction reduces the complexity of the task of definition of security policies. The automated transformation of high level system and security models to platform specific artifacts greatly reduces development time.

The presented tool chain is successfully used for the development of a secure prototypical Air Traffic Control visualization application as part of the European AD4 project [10]. The tool chain and the `SecureMiddleware` platform currently supports two very commonly used security models, Role Base Access Control for the invocation of operations and Mandatory Access Control for controlling information flow, but can be easily extended to other security models as well. In addition to the generation of the rules for the component interaction, our tool chain is also able to automatically generate the complex rule sets for the underlying infrastructure daemons.

References

[1] Object Management Group. "CORBA Component Model". OMG document number formal/02-06-65

[2] ObjectSecurity. SecureMiddleware Project. URL: http://www.securemiddleware.org

[3] Qedo Team. Qedo (Quality of Service Enabled Distributed Objects) CCM Implementation Web Page, URL: http://www.qedo.org, March 2006

[4] ObjectSecurity. OpenPMF Project. URL: http://www.openpmf.org

[5] Ritter, T., Lang U., Schreiner R.: Integrating Security Policies via Container Portable Interceptors, Adaptive and Reflective Middleware Workshop (ARM2005) at Middleware 2005

[6] Lodderstedt T., SecureUML: A UML-Based Modelling Language for Model-Driven Security. In UML 2002 - The Unified Modelling Language. Model Engineering, languages, Concepts, and Tools. 5th International Conference, Dresden, Germany, September/October 2002, Proceedings, volume 2460 of LNCS p. 426-441, Springer, 2002

[7] Basin D., Doser J., Lodderstedt T., Model-Driven Security: from UML Models to Access Control Infrastructures. September 4, 2005

[8] Object Management Group. Meta Object Facility Core Specification 2.0, OMG document number, formal/2006-01-01

[9] Lang, U., Schreiner, R. OpenPMF Security Policy Framework for Distributed Systems. Proceedings of the Information Security Solutions Europe (ISSE 2004) Conference, Berlin, Germany, September 2004

[10] AD4 Consortium. EU FP6 R&D project AD4 - 4D Virtual Airspace Management System, URL: http://www.ad4-project.com/

[11] ObjectSecurity. URL: http://www.mico.org/

[12] Lampson, B., Abadi, M., Burrows, M., Wobber, E. Authentication in Distributed Systems: Theory and Practice. ACM Transactions on Computer Systems 10, 4, pp 265-310, November 1992

[13] Object Management Group: OMG ptc/04-10-02: UML 2.0 Superstructure Revised Final Adopted Specification

[14] Object Management Group: OMG ptc/03-09-15:UML 2.0 Infrastructure Final Adopted Specification

[15] Sparx Systems: URL: http://sparxsystems.com.au/

[16] Object Management Group: OMG ptc/06-04-12: QoS4CCM Final Adopted Specification

On the Dominance of Decompositions in Models and their Aspect-Oriented Implementations

Tommi Mikkonen [1]

Institute of Software Systems
Tampere University of Technology
P.O. Box 553, FI-33101 Tampere, Finland

Abstract

Aspect-oriented approaches have commonly advocated separation of concerns. Some approaches have applied this separation in a symmetric fashion, like Hyper/J, whereas some others have relied on asymmetric separation, like AspectJ. The difference in the approaches is that the different concerns play a symmetric role in the former, whereas the latter explicitly includes a conventional implementation on top of which other concerns are woven onto as aspects. The question then arises, how are the concerns of the conventional implementation special in the latter, and will the opportunity to use symmetric separation lead to a fundamentally different decomposition. In this paper, we discuss the dominance in decompositions in specifications and corresponding aspect-oriented implementations. As examples, we use the specification method DisCo which allows modeling of concerns in a fashion that separates the different concerns to specification branches, and aspect-oriented implementations using Hyper/J and AspectJ that can be composed for DisCo specifications. As the final outcome, we propose that any aspect-oriented approach addressing the system at the level of program code necessarily has some concerns that are more dominant than some others due to the control flow of programs.

Key words: Dominant decomposition, symmetric and asymmetric aspect-orientation

1 Introduction

Tyranny of dominant decompositions is the term introduced to address problems related to the inability to address all concerns of a software system using the same facilities. Reported in [17], this has lead to the introduction of a paradigm where all the concerns can be treated in a similar fashion, with a

[1] Email: `tommi.mikkonen@tut.fi`

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

practical programming-level implementation in Hyper/J [20]. Then, all concerns can be treated symmetrically, which enables the creation of systems so that both conventional modularity and cross-cutting properties are enabled using hypermodules.

In contrast to symmetric approaches to manage cross-cutting properties, asymmetric approaches have been introduced. For instance, AspectJ [18] introduces facilities for augmenting a baseline implementation with additions referred to as aspects. Then, one can advance such that an existing system, given as a conventional program, is taken as the starting point, and new behaviors are woven into the system with aspects. Furthermore, provided with a convention where aspects are used for certain issues, the developers can anticipate the injection of aspects, and overlook such parts of the system in the baseline implementation.

In general, the relation between symmetric and asymmetric approaches to aspect-orientation [6,7,4] has been an interesting topic of research (e.g. [9]). In this paper, we discuss the dominance of decompositions in terms of a specification, where one type of separation of concerns is provided, and sketch aspect-oriented implementations for it using Hyper/J and AspectJ. The purpose is to address differences and commonalities of the techniques, and to compare their different properties to each other as well as the underlying specification. Towards the end of the paper, we connect our results to the framework provided by model-driven architecture, MDA [8,21] by concluding that implementation-level techniques essentially require a dominant decomposition for executability and meaningful binding to control flow. The experiences are based on a case study, where the behavior of a mobile switch has been re-engineered [10].

The rest of this paper is structured as follows. Section 2 introduces the specification and modeling method we use as the starting point. Furthermore, we discuss the structure of specifications that has been commonly used for separating concerns. Sections 3 and 4 sketch implementations for DisCo specifications using Hyper/J and AspectJ, where symmetric and asymmetric decompositions of concerns are offered. In addition, we discuss the dominance of decompositions in described systems. Then, Section 5 finally concludes the paper.

2 Specification

As the method of specification in this paper, we use DisCo [12,19], a formal method that is based on Temporal Logic of Actions [13], and allows the development of specifications and models in a fashion where different concerns are incrementally introduced.

2.1 DisCo Principles

DisCo specifications are composed in terms of layers that contain classes and actions. Classes are containers of data, and actions can be understood as multi-object methods that can alter values of variables. Actions are executed in an interleaved fashion without any interference from the rest of the system and their execution is bound to be finished once it has been initiated, which makes actions atomic units of executions. The language used for composing specifications is textual. However, animation facilities have been provided to ease the analysis of specifications [16,3]. Furthermore, the relation of DisCo specifications and their denotation using UML has been studied in [15].

Layers can build on top of other layers, which is referred to as refinement in DisCo terminology. A restriction is made that actions can only alter values of variables given in the same layer, which guarantees that safety properties will be preserved by construction. To satisfy this restriction, refinements can introduce new variables and operations on them as well as augmentations to actions, provided that the new action logically implies its ancestor. In fact, one can consider that the concept of ancestors plays a key role in the DisCo approach. All classes and actions can be considered as the refinements of their ancestors in earlier layers, with the layer defining an empty system as the origin of the hierarchy. This gives an explicit structure for any DisCo specification, with the opportunity to define concerns in individual layers.

When two or more layers are merged, actions may be merged into more complex ones or be kept apart from one another. This corresponds to weaving in the aspect-oriented setting, to which we will return in Sections 3 and 4, where aspect-oriented techniques are considered.

2.2 Concern-Based Modeling of Abstractions with DisCo

The use of layers allows modeling of systems using several levels of abstraction. For instance, when modeling a telephony exchange, it is possible to model the system using abstract concepts, such as call control, connections, and legs, which are individual connections from the exchange to a caller or callee, and charging, as well as in terms of processes used for implementing the abstract concepts. This allows each layer to focus on a certain concern the modeler wants to address separately.

As long as the relation between abstractions is simple, for instance each abstract concept can be associated with a certain low-level concept, this scheme is relatively simple. For instance, a low-level concept can be a process or interprocess communication. However, when an abstraction is defined that requires cooperation of several low-level concepts, more complex implementations result. In this paper, we will refer to these two types of abstractions as primitive and non-primitive, respectively. Another way to consider such abstractions is to refer to them as local and cross-cutting.

The refinement of a non-primitive abstraction to a directly implementable

form is yet another concern. Therefore, this issue is commonly addressed using a DisCo layer that defines the relation between an abstract concept and its more concrete implementation. Techniques have been introduced for such transformations, including the option to use archived design steps that resemble design patterns [11].

2.3 Specification Architectures in DisCo

Due to the definition of the methodology, specifications given in DisCo contain two different types of architectures. One is composed in terms of classes and actions, and the other is composed with layers. The architecture composed with classes and actions resemble those composed with conventional techniques, but the architecture composed with layers is less conventional. Therefore, this topic will be addressed in more detail in the following.

Layered specifications in DisCo allow layers where individual concerns are addressed. Layers are truly symmetric in the sense that the refinement relationship between layers preserves (safety) properties of all component layers, and the order in which layers are given can vary. Furthermore, refinements can only make more restrictions, which resembles the constraint-oriented design style introduced in connection with LOTOS [5]. Another difference to commonly used aspect-oriented approaches is that layers are complete in the sense that they only describe behaviors in terms of the variables included in them. Then, when layers are composed, a new universe is created where the rules of behavior of all component layers are satisfied.

Such layers allow an approach where the control of the system is given in one layer, which is then composed with other layers. When considering the architecture created with layers, a control layer can play two different roles.

- The use of primitive (or local) abstractions only allows an approach where fundamental properties are defined in the beginning, leading to a bottom-up approach.
- The use of non-primitive (or cross-cutting) abstractions requires an implementing layer, resulting in a top-down construction of the system. This layer can be given as a separate specification branch to promote separation of concerns.

These two approaches are illustrated using a simplified version of a telephony exchange as an example (Figure 1). There are three main functions in the exchange. Layer `Legs` introduces abstract concept legs, i.e., connections between the caller/callee and the exchange; layer `Processes` implements the abstract concept using available techniques, with different actions for incoming and outgoing calls and routing; and layer `Charging` introduces the capability to charge for established legs. As it is the relation of these functions that are important, we will only discuss them in an abstract fashion. Moreover, operations given in them have been included in the figure, as they are meaningful for implementations we will describe later on.

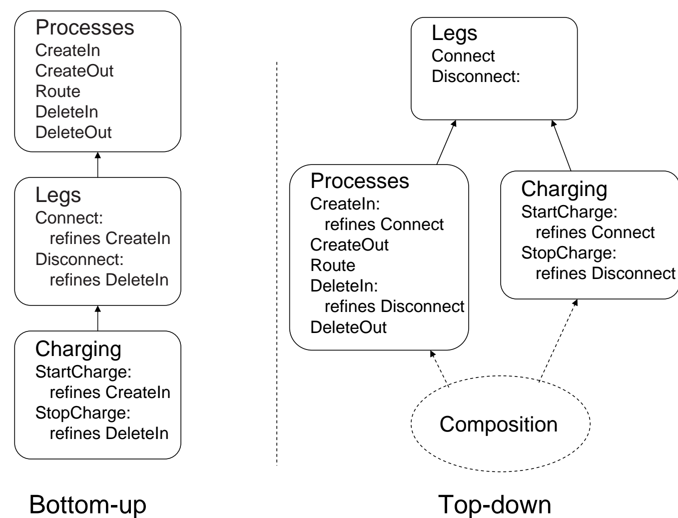

Fig. 1. Specification architecture of DisCo specifications

2.4 Implementing DisCo specifications with conventional techniques

When using conventional techniques, a one shot implementation aims at attacking the complete composition of all the layers, where the resulting architecture1 obeys the structure defined in layers, but overlooks the layer structure. In Figure 1, the bottom-up approach is straightforward to implement, but the top-down approach requires the creation of a composite specification shown in Figure 1. In other words, architecturally significant parts given inside the layers form the dominant decomposition, and the layered structure as a whole is overlooked [14].

More recently, we have also considered aspect-oriented techniques [1]. This enables us to preserve the structure created with layers. Moreover, we have considered applying the same design methodology in the design of aspect-oriented systems. In the following sections, we shift the focus on composing aspect-oriented implementations in a fashion that preserves the layered structure of the specification.

3 Hyper/J Implementation

Hyper/J [20] is probably the most prominent approach to symmetric aspect-oriented programming. It can intuitively be used for a similar separation of concerns as DisCo. However, the level of abstraction in the two approaches is different, which has some consequences on how systems can be constructed in a pragmatic fashion. In the following, we outline a mechanism for implementing the above DisCo specification using Hyper/J.

3.1 Source of Symmetry

When composing a Hyper/J system, independent subsystems are defined in the beginning. They can be individual classes or collections of them, and they can be even tested in isolation from one another. The goal is to create all the necessary operations of the eventual implementation in isolation.

Once a collection of implementation classes exist, the designer uses hypermodules to define how the different parts of the system are integrated. The means provided by Hyper/J basically require that two methods are equated, and enable the definition of call order of the methods. However, more complex facilities for relating the different classes can be imagined. Because the starting point for the development is a collection of classes and definitions that combine them, it is obvious that from the technical perspective all classes play a similar role.

3.2 Aligning Hyper/J Implementation with DisCo Specification

As the starting point of implementing a DisCo specification with Hyper/J, one should compose classes out of DisCo layers, reflecting the contents of the layers in detail. Then, once Java implementations for the contents are available, one should match the classes and equate the operations that correspond to each other in different classes. Sequential and branching architectures can be treated in a similar fashion, which makes Hyper/J a natural candidate for implementations of DisCo specifications. Moreover, equating the methods can be implemented in pieces, which allows an incremental approach to the development not unlike that of DisCo (Figure 2).

However, there is one major drawback. Before one includes the branch that defines the master control flow only declarative goals can be achieved. In other words, by combining branches that define operations that do not trigger themselves to execution, one cannot create any runnable programs. As a result, the semantics of different branches have different contribution to the development. Therefore, the branch defining the control flow can be considered as a dominant dimension in the semantic sense.

As a result of the special role of control flow, in cases where an early phase of the DisCo specification defines executions the situation is simple. The corresponding Hyper/J implementation can be given and tested in a straightforward fashion. Then, as the design advances, new features can be immediately augmented with normal routines. Furthermore, also testing by running the system is enabled. However, in cases where some later branch introduces the control, combinations where the implementation branch is missing result in declarative statements. This in particular applies to non-primitive abstractions, which often define no control as such but are conceptually important. They cannot be tested in a Hyper/J implementation independently of the rest of the system, but they can still be defined and used as intermediate systems that can be studied with reviews.

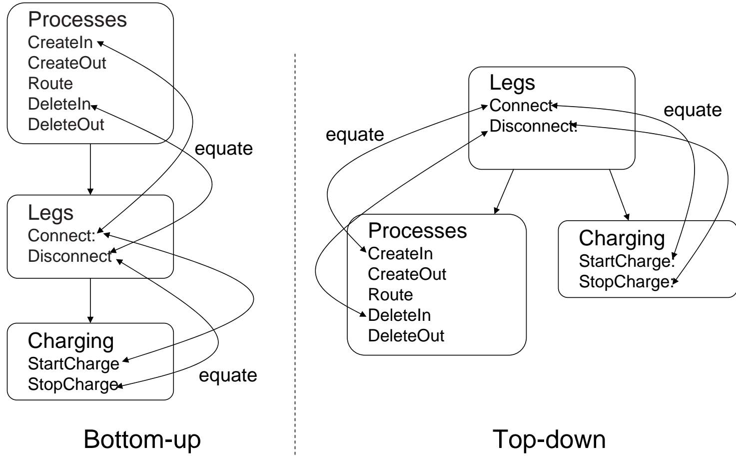

Fig. 2. Architecture of Hyper/J implementation preserving the structure

4 AspectJ Implementation

In contrast to Hyper/J, AspectJ [18] aims at the definition of systems in a fashion where a baseline implementation is given first. This baseline is given using conventional Java, and it will be extended with aspects that are woven into code. In other words, aspects can be taken as extensions of the baseline system, and they most naturally follow the control given in the baseline, although an option has been provided to override operations. In the following, we outline a mechanism for implementing the above DisCo specification using AspectJ.

4.1 Source of Asymmetry

The fundamental source of asymmetry in AspectJ is that there are two types of artifacts. One type is the conventional Java classes, and the other type is the aspects. The types also play a different role in the development process, which is addressed in the following.

When the development begins, a baseline implementation is created with conventional Java classes. This system can be tested and run normally. Then, once the baseline is satisfactory, aspects are woven to it to introduce new properties. However, unlike in Hyper/J where additional information was used to combine operations, aspects of AspectJ have information on where they should be woven. In fact, a common goal is to use aspects such that the baseline needs not to know about the use of aspects, because this would liberate the developers of the baseline to focus on its goals and requirements.

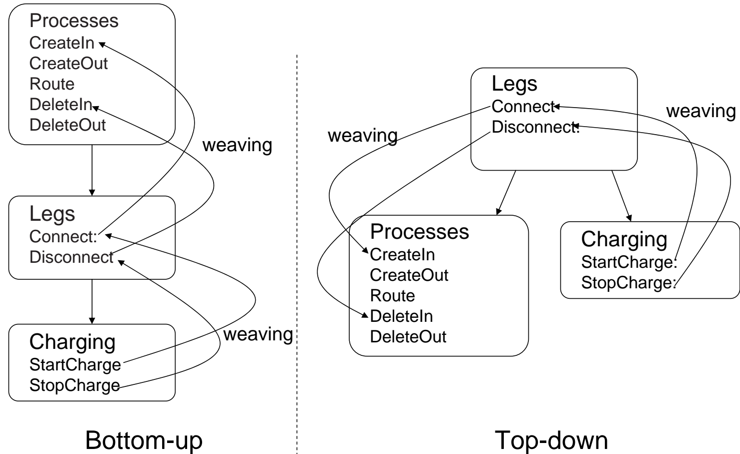

Fig. 3. Architecture of AspectJ implementation preserving the structure

4.2 Aligning AspectJ Implementation with DisCo Specification

When composing an AspectJ implementation of a DisCo specification, the starting point is always an executable that essentially defines the master control flow. The definition can be given either in the main branch, or in some other branch, but we consider that this is the fundamental dominant decomposition of the system in the sense of its behaviors. Moreover, in order to create a runnable baseline it is an obvious necessity.

Once the baseline is completed, other features are integrated to it using aspects. In this case, non-primitive abstractions can become abstractions that implement the abstraction in one module, assuming that the control of the system is introduced in some later branch (Figure 3). This has already been addressed in detail in [1].

5 Discussion

Specification-level separation of concerns can be different from implementation-level separation of concerns. In this paper, behavioral dominant decomposition at a level of specification and its structural dominant decomposition in aspect-oriented systems were shown to have the potential to be different. The reason for the difference lies in the control flow, which can be abstracted away in a specification but forms a necessary element in an implementation. When considering the behaviors of the system, executions that must take place commonly introduce a dominant decomposition any case in implementations, as otherwise only declarative definitions can be given. As a result, structure-preserving implementations of the specifications, whose architecture had prescribed differences regarding the order in which certain concerns were

introduced, are similar for both Hyper/J and AspectJ.

We believe that the reason for the difference lies in cross-cutting abstractions, which extend from one implementation-level abstraction to another. Without aspect-oriented techniques, such abstractions would result in tangled code, but with them the architecture of the composed implementation can be aligned with the specification. The rationale for the above is that non-primitive abstractions are cross-cutting, and can therefore benefit from aspect-oriented techniques for obvious reasons. However, without an executable main branch, it is difficult to locate the correct methods to equate or associated pointcuts. Yet they are an important tool when composing models, where the abstractions can be studied in a meaningful fashion. In particular, as already mentioned, they can be used for re-engineering the structure of systems into a form that easily lends itself to an aspect-oriented implementation, which has also given inspiration for the example used in this paper.

The consequences of the observation are many. Firstly, even if technically symmetric facilities are offered for composing systems in an aspect-oriented fashion, the fact that the introduction of control flow that necessarily is a cross-cutting concern implicitly creates asymmetry may be an argument on what types of systems benefit from aspect-orientation. Secondly, the fact that in some cases abstractions are not executable as such but bear declarative meaning is something that may affect unit testing of aspects, and give weight for research aiming at unit testing at the level of aspects, which seems to require a test driver or stub that models the expected control flow. Finally, we believe that it is essential for early use of aspect-oriented techniques to overlook control flow oriented dominance, and focus on problem-oriented decompositions. Later on, implementation techniques, where control flow is intimately present, can then be used to introduce this concern, following the practices of Model-Driven Architecture, MDA [8,21]. In essence, both Hyper/J and AspectJ require considering similar issues at the level of Platform Specific Models without offering much support for Computation or Platform Independent Models of MDA. Therefore, the facilities the developer can be benefitted from can be considered restricted.

Finally, we believe that in addition to the low-level relation of different concepts discussed in this paper, the relation between specification level concepts and corresponding aspect-oriented techniques enable a more sophisticated view to early use of aspect-orientation. Although based on different origins and terminology, the possibility to compose such an alignment gives a raise to an extended discussion [2].

References

[1] Timo Aaltonen, Joni Helin, Mika Katara, Pertti Kellomäki, and Tommi Mikkonen. Coordinating objects and aspects. *Electronic Notes in Theoretical Computer Science*, 68(3), March 2003.

[2] Timo Aaltonen, Mika Katara, Reino Kurki-Suonio, and Tommi Mikkonen. On horizontal specification architectures and their aspect-oriented implementations. Accepted to Transactions on AOSD, to appear.

[3] Timo Aaltonen, Mika Katara, and Risto Pitkänen. DisCo toolset – the new generation. *Journal of Universal Computer Science*, 7(1):3–18, 2001. At URL http://www.jucs.org.

[4] AOSD-Europe. Survey of aspect-oriented analysis and design approaches. At URL http://www.aosd-europe.org, May 2005.

[5] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[6] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming. *Communications of the ACM*, 44(10):29–32, October 2001.

[7] Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit. *Aspect-Oriented Software Development*. Addison-Wesley, 2005.

[8] Object Management Group. MDA guide version 1.0.1. OMG Document Number omg/2003-06-01, June 2003.

[9] William H. Harrison, Harold L. Ossher, and Peri L. Tarr. Asymmerically vs. symmetrically organized paradigms for software composition. Technical Report RC22685, IBM Thomas J. Watson Research Conter, December 2002.

[10] Seppo Isojärvi. DisCo and Nokia: Experiences of DisCo with modeling real-time system in multiprocessor environment. Formal Methods Europe Industial Seminar 1997, Otaniemi, Finland, February 20, 1997.

[11] Pertti Kellomäki and Tommi Mikkonen. Design templates for collective behavior. In Elisa Bertino, editor, *Proc. ECOOP 2000, 14th European Conference on Object-Oriented Programming*, number 1850 in Lecture Notes in Computer Science, pages 277–295. Springer-Verlag, 2000.

[12] Reino Kurki-Suonio. *A Practical Theory of Reactive Systems — Incremental Modeling of Dynamic Behaviors*. Springer, 2005.

[13] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[14] Tommi Mikkonen. A development cycle for dependable reactive systems. In Y. Chen, editor, *Proc. IFIP International Workshop on Dependable Computing and its Applications, DCIA98*, pages 70–82. University of Witwatersrand, Johannesburg, South Africa, 1998.

[15] Risto Pitkänen and Petri Selonen. A UML profile for executable and incremental specification-level modeling. In Thomas Baar, Alfred Strohmeier, Ana Moreira, and Stephen J. Mellor, editors, *UML 2004 - The Unified Modeling Language. Model Languages and Applications. 7th International Conference, Lisbon, Portugal, October 11-15, 2004, Proceedings*, volume 3273 of *LNCS*, pages 158–172. Springer, 2004.

Impact of Evolution of Concerns in the Model-Driven Architecture Design Approach

Bedir Tekinerdoğan, Mehmet Akşit & Francis Henninger
Dept. of Computer Science, University of Twente,
P.O. Box 217 7500 AE Enschede, The Netherlands.
{bedir|aksit|henninger@cs.utwente.nl}

Abstract. Separation of concerns is an important principle for designing high quality software systems and is both applied in the Model-Driven Architecture (MDA) and Aspect-Oriented Software Development (AOSD). The AOSD and MDA techniques seem to be complementary to each other; historically AOSD has focused on modeling crosscutting concerns whereas MDA has focused on the explicit separation of platform independent concerns from platform specific concerns and the model-driven generation processes. In order to assess the benefits of AOSD for MDA we provide a systematic analysis on crosscutting concerns within the MDA context. The analysis consists of three steps. First, we define an abstract model of MDA transformation with respect to concerns. Second, we define a number of evolution scenarios that correspond to a selected list of crosscutting concerns. Third, we analyze the model transformations in MDA with respect to the abstract model, the evolution scenarios and the related crosscutting concerns. This analysis results in the definition of a number of key problems related to the integration and evolution of crosscutting concerns in the MDA approach. Based on this analysis we provide a set of recommendations for the language and the process that is used in the MDA approach.

Keywords: Crosscutting concerns, Evolution, MDA

1 Introduction

One of the most important principles to cope with the complexity in software engineering is the separation of concerns principle. This principle states that a given problem involves different kinds of concerns, which should be identified and separated to cope with complexity, and to achieve the required engineering quality factors such as robustness, adaptability, maintainability, and reusability.

In this context, Model Driven Architecture (MDA) is a framework defined by the OMG that separates the platform specific concerns from platform independent concerns to improve the reusability, portability and interoperability of software systems. To this end MDA separates Platform Independent Models (PIMs) from Platform Specific Models (PSMs). The PIM is a model that abstracts from any implementation technology or platform. The PIM is transformed into one or more PSMs which include the platform specific details. Finally the PSM is transformed to code providing the implementation details. Obviously by separating the platform specific concerns and providing mechanisms to compose these concerns afterwards in the code MDA provides a clean separation of concerns and as such the systems are better reusable easier to port to different platforms and have increased interoperability.

However, current software systems also have to cope with other concerns than platform specific concerns. Very often software systems also need to deal with other important concerns such as distribution, persistence, synchronization, and error detection. These concerns tend to crosscut various components of the software architecture and as such increase the complexity and decrease the maintenance of software systems.

Aspect-Oriented Software Development (AOSD) aims to cope with these crosscutting concerns by providing explicit abstractions called aspects. By separating the crosscutting concerns in aspects and providing the composition of aspects with the components the impact of crosscutting concerns is better managed.

Both AOSD and MDA provide in essence useful techniques for separating the concerns and in that sense AOSD and MDA techniques are complementary to each other; historically AOSD has focused on modeling crosscutting concerns whereas MDA has focused on the explicit separation of platform independent concerns from platform specific concerns and the model-driven generation processes. As such we think that both AOSD and MDA can benefit from each other to even further tackle the challenges of current large and complex software systems.

Since AOSD is primarily focused at solving the problems related to crosscutting concerns we will provide a systematic analysis of crosscutting concerns within the MDA context. The analysis consists of three steps. First, we define an abstract model of MDA transformations as defined by so-called *concern transformation patterns* (CTP). CTPs characterize the corresponding transformation and help to pinpoint the key problems in the transformation. Second, we define a number of evolution scenarios that correspond to a selected list of crosscutting concerns. The evolution scenarios are applied to a Concurrent Versioning System which is developed using an MDA-based approach. Third, we analyze the model transformations in MDA with respect to CTPs, the evolution scenarios and the related crosscutting concerns.

This analysis results in the definition of a number of key problems related to the integration and evolution of crosscutting concerns in the MDA approach. Based on this analysis we provide a set of recommendations for the language and the process that is used in the MDA approach.

In section 2 we will present the concern transformation patterns as an abstraction of different potential transformations. Section 3 explains the case on concurrent versioning systems. This case will be used to explain the notion of crosscutting concerns and to analyze the impact of crosscutting concerns in the MDA process. Section 4 will define the number of scenarios including the concerns that will be applied to the CVS in the MDA life cycle. Section 5 defines the lessons learned and provides recommendations for coping with crosscutting concerns in the MDA-approach. Finally section 6 will provide the conclusions.

2 Concern Transformation Patterns

To analyze the impact of crosscutting concerns in MDA, we will first consider the major modeling concepts and transformations as defined within the MDA context. As shown in *Figure 1*, a typical MDA development process consists of model building and transformation activities, starting with a computation independent model (CIM), which is subsequently transformed to a platform independent model (PIM), platform specific model (PSM) and finally to an executable code. It is also possible to transform models at the same abstraction level. Figure 3 shows the following set of transformations: CIM-to-CIM, CIM-to-PIM, PIM-to-PIM, PIM-to-PSM, PSM-to-PSM, and PSM-to-code.

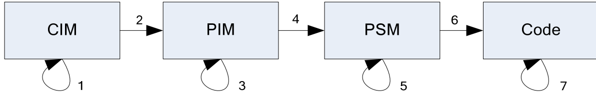

Figure 1. *Simplified MDA process*

Obviously, each model building and transformation process considers a set of possibly new concerns that are relevant to the model being considered. Since AOSD aims to model crosscutting concerns explicitly, it would be logical to analyze the impact of aspects to MDA (or vice versa) from a concern

modeling and transformation perspective. Our analysis, therefore, will be based on the following three assumptions:

1. ***A model is a representation of concerns***. Each model defined within the MDA context consists of two kinds of concerns: the concerns that are distinguished based on their platform dependency and the concerns that are derived from the requirements and the corresponding problem/solution domain.
2. ***Model transformations are concern transformations.*** Since every model consists of concerns, naturally every transformation is also a concern transformation.

Based on these assumptions we define eight concern transformation patterns as described in Table 1. In the table A, B and C represent models of concerns, the arrow labeled with T represents a transformation. The brackets <> in both the models and transformations include the concerns that are addressed by these. In addition two composition operators are defined • and ⊗. The null-composition operator • represents the required conceptual composition and is defined in the input to the transformation. The operator ⊗ defines a composition in which the models can be separately identified. If the models can not be separated after the composition then we use the <> symbols. For example, $A_{<B,C>}$ refers to a merged composition of concerns B and C in model A, whereby B and C can not be localized. Note that patterns 1, 3, 5, 6 and 7 lead to such a merged composition in which one or more concerns are not explicitly localized. Finally, patterns 5 to 8 can be considered as composite concern transformation patterns because each of them could be in essence defined as a composition of two other primitive transformation patterns. Pattern 5 could be seen as a composition of pattern 1 and 3, pattern 6 as a composition of patterns 1 and 4, pattern 7 as a composition of patterns 2 and 3, and finally pattern 8 as a composition of patterns 2 and 4.

In the first two patterns the transformations do not include an explicit concern but are mainly used to compose the provided concerns (in this case A and C). In the first pattern the composition result is a merged output in which concern C is not separated anymore. In the second pattern, the concern C is kept separate indicating a more composable design.

Patterns 3 to 8 include transformations that add concerns themselves. In pattern 3 and 4 include situations in which no additional concerns are provided to the transformations. The third pattern represents a transformation in which the model A is transformed to the target model $A_{<B>}$. Here, the concern B is introduced by the transformation process $T_{<B>}$ and it is not separable from the concerns that are originally defined in A. Assume for example that the model A represents a PIM which is transformed to a Java PSM. In case of pattern 3, it is assumed that the Java specific concerns introduced in $A_{<B>}$ cannot be separated from the original concerns of A.

In pattern 4, the model A is transformed to the composition of the models A and B as represented by the composition operator ⊗. Similar to pattern 3, model B represents the newly introduced concern by the transformation process. The difference is that the result of transformation $T_{<B>}$ is a modular composition in which model A and B can be separately identified.

The patterns from 5 to 8 assume that the source model consists of two modularly composed modules A and C as represented by the composition operator ⊕. In pattern 5, the original concerns A and C together with the concern as defined in the transformation $T_{<B>}$ are merged in the target model $A_{<B, C>}$ and therefore they cannot be treated separately anymore. In pattern 6, concern C is merged but concern B from the transformation is provided as a separate model. In pattern 7 concern B from the transformation is merged while concern C is represented as a separate model. Finally pattern 8 shows the case in which concerns B and C are separated. In fact this can be considered as the most maintainable alternative because the concerns are fully separated, both in the source and the target.

Table 1. *Concern transformation patterns*

Id	Concern Transformation Pattern	Explanation
1.	A $\xrightarrow{T_{<>}}$ $A_{<C>}$ • C	A is transformed with concern C to a model A<C>. A is the dominant decomposition, concern C cannot be separately identified.
2.	A $\xrightarrow{T_{<>}}$ A • ⊗ C C	A is transformed along with concern C to a model A and model C. Concern C is separately modeled in model C.
3.	A $\xrightarrow{T_{<B>}}$ $A_{<B>}$	A is transformed to $A_{<B>}$ via $T_{<B>}$. A includes new concern B that is inherent in the transformation but which is not separable.
4.	A $\xrightarrow{T_{<B>}}$ A ⊗ B	*A* is transformed to a composable model of *A* and *B*, which are separable in the final result.
	Composite Concern Transformation Patterns	
5.	A $\xrightarrow{T_{<B>}}$ $A_{<B, C>}$ • C	A is transformed along with concern C to a single model $A_{<B, C>}$. In the resulting model concern B and C can not be identified any more as separate models.
6.	A $\xrightarrow{T_{<B>}}$ $A_{<C>}$ • ⊗ C B	A is transformed along with concern C to a single model $A_{<C>}$ and model B. In the resulting model $A_{<C>}$ the concern C cannot be separated. Concern B is separately modeled in model B.
7.	A $\xrightarrow{T_{<B>}}$ $A_{<B>}$ • ⊗ C C	A is transformed along with concern C to a model $A_{<B>}$. and model C. In the resulting model $A_{<B>}$ concern B from the transformation cannot be separated. Concern C is separately modeled in model C.
8.	A $\xrightarrow{T_{<B>}}$ A • ⊗ C B ⊗ C	A is transformed along with concern C to a model A, B and C. In the resulting model all the three concerns A, B and C can be separated.

LEGEND

Symbol	Meaning	Symbol	Meaning
A	Model	$\xrightarrow{T_{<>}}$	Transformation (no implicit concern)
A ⊗ C	Resulted Composition of Models	$\xrightarrow{T_{<C>}}$	Transformation with explicit concern
A • C	Required Composition of Models	$A_{<C>}$	Model A with non-localized concern C

3 Example: Concurrent Versioning System

To analyze the evolution of concerns in the MDA approach we will utilize the Software Configuration Management (SCM) case. The SCM deals with control of software changes, proper documentation of changes, the issuing of new software versions and releases, the registration and recording of approved software versions. An important functionality in SCM forms the concurrent version control system (CVS), which keeps a history of the changes made to a set of files that can be concurrently accessed.

In Figure 2, the conceptual architecture of a CVS system is shown. This architecture consists of four major sub-systems: programmer's environment, administrator's environment, session management system and repository management system.

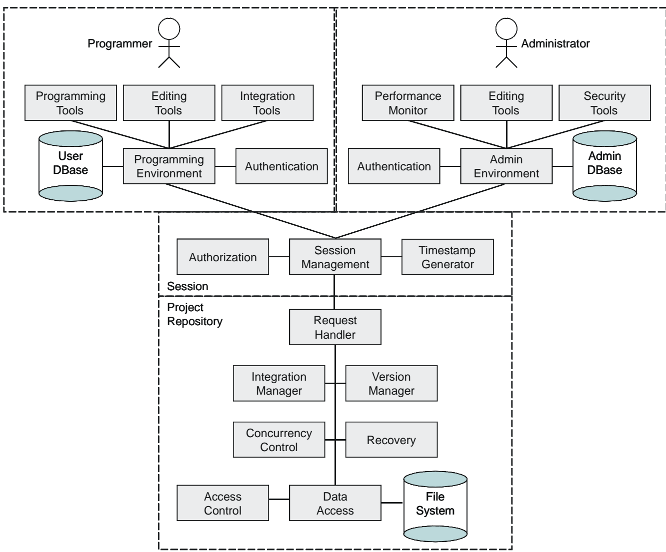

Figure 2. *The conceptual architecture of the CVS system.*

The programmer's environment provides a set of programming tools, such as compilers, interpreters and debuggers, editors and tools for integrating program modules into a consistent program. Programmers have to be authenticated before they start using the system. When a programmer wants to edit a file which is stored in the project repository, a request is made to the session manager. The session manager authorizes the request, associates a timestamp with it, and initiates an editing session by calling on the request handler of the project repository. The request handler *checks out* the requested file and passes it to the programmer's environment. When files are checked out they can be edited and compiled and *check in* the modifications to the file. Checking out a file does not give a programmer exclusive rights to that file. Other programmers can also check it out, make their own modifications, and check it back in. The concurrency control module administrates all the simultaneous accesses to the same file, together with the identity of the users, access time and the version numbers. This module is also responsible in identifying the read/write conflicts in accessing the files. If a conflict is detected, the integration manager is called. The integration manager provides a set of functions to resolve the conflicting accesses. For example, the integration manager may notify the programmers, or may ask assistance from the authorized person to resolve the conflict. The recovery manager makes a copy of all the modified files so that the original files can be restored if necessary. The version manager generates version ID's, compares versions of files and notifies if there are inconsistent versions in the same configuration. The access control unit provides low-level access control functions such as protecting read only files from updates, etc.

The administrator's environment provides a set of management tools. For example, the performance monitor is used to generate reports on the average time of accesses, the affect of data size and simultaneous accesses to performance, number of aborts, etc. The administrator is also responsible for installing the session workflow attributes, i.e. priorities, responsibilities in resolving conflicts. Authentication and authorization rights are also determined by the administrator.

4 Evolution Scenarios

Given the models and the transformation of MDA we can now focus on the analysis of crosscutting concerns in MDA. To illustrate the impact of crosscutting on model transformations we will add the concerns *security, logging, versioning strategy* and *persistence.* Obviously, these concerns might be introduced at the CIM, PIM, PSM or even the code level. If we also consider the various orderings in which these concerns might be introduced in the model transformation process, it becomes clear that the number of model transformations increase dramatically. An exhaustive analysis of all the possible alternative model transformations with these specific concerns could provide us a complete overview of the problems that we might encounter. However, it is from a practical point of view not possible to analyze all these possibilities. Further, our goal in this paper is not to provide a complete analysis for the specific set of concerns but rather to pinpoint practical recurring problems that might appear in MDA while applying crosscutting concerns. Secondly, in principle it also does not seem to be necessary to provide an exhaustive analysis to identify the key problems. This is because we can easily group the transformations and reason about crosscutting concerns in the model transformations in a more abstract manner.

For our analysis we will define a set of evolution scenarios to the given example, in which crosscutting concerns will be added. The set of scenarios that we apply is depicted in Table 2. Note that we have included each model abstraction level (CIM, PIM, PSM and code) as well as the transformations among these. In addition, for the horizontal transformations we have included one or two concerns. Later in the discussions we will also analyze the various possible orderings of these scenarios. The ordering of the scenarios that we will discuss in this paper is presented in the scenario transition diagram as depicted in the right column of Table 2. Hereby, the labeled circles present the scenarios on the left, whereas the arrows represent the transition between scenarios.

Table 2. *List of scenarios including crosscutting concerns applied in the MDA Transformations*

Transformation	Scenario	Order of scenarios
CIM to CIM	S1. Adding new concern security to use case model	
CIM to PIM	S2. Transforming use case model to PIM	
PIM to PIM	S3. Add Security at the PIM level	
	S4. Add/Update Logging at the PIM level	
PIM to PSM	S5. Transform to Relational DB Platform model	
	S6. Transform PIM to Java platform model	
PSM to PSM	S7. Adding Versioning Strategy	
	S8. Adding Persistence	
	S9. Upgrade to Remote Security invocation	
PSM to Code	S10. Transformation to Java Code	
	S11. Transformation to Relational Tables	

4.1 CIM to CIM

The first transformation (scenario S1) that we consider is the CIM to CIM transformation (Table 2). The Computation Independent Model (CIM) focuses on the environment and the requirements of the system; it does not concern with any structural or processing details of the system. We assume that the CIM is expressed as a set of use case models. We consider in this example the addition of the concern security to the CIM model.

For the given scenario S1 we can apply concern transformation pattern 1 and 2. It appears that we can model and compose the use case *security* and as such transformation pattern 2 is applied, as depicted in Figure 3. The left part of the figure shows the null-composition of the source model with the security use-case. The right part shows the result after the transformation, in which the use cases have been composed using the «uses» relation to the new concern *security*. Although the concern security could be separated at this level, the number of relationships already denotes that there is possibly crosscutting in the later models that will be derived from this model. We could also imagine a case in which a concern can not be easily localized in the use case model. This could be for example a concern like, *optimize time performance in accessing CVS*. Obviously it is very hard to specify this in a separate use case, and typically all the time performance optimization must be performed within each use case itself. As such we could characterize this transformation as an application of pattern 1 specified as $CVS_{<time\ performance>}$.

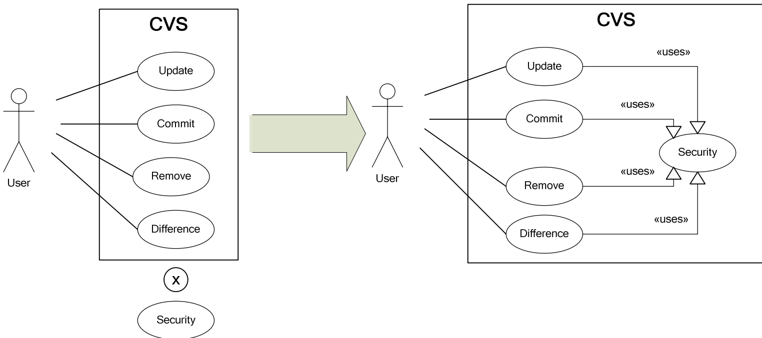

Figure 3. *CIM to CIM transformation while adding a security concern*

4.2 CIM to PIM

Figure 5 presents the result of the transformation to a Platform Independent Model (PIM), as defined by scenario S2 (transformation to PIM) in Table 2. In Figure 5, *Repository* is used to group the different versions of the system and can be used for tracking changes in a configuration management system. The class *Branch* includes class *Directory* that can store *File.* Class *Version* defines a particular version and multiple versions of a file may exist. The attributes *deleted* in the classes *File* and *Directory* denote whether the file or directory has been deleted or is still used. Files might be labeled using class *Tag.*

The transformation from a computation independent model to a platform independent model can be considered as a transformation in which the concern is computation concern itself. Naturally, it is very hard to separate the computation concerns in the PIM and as such we can state that pattern 3 is applied.

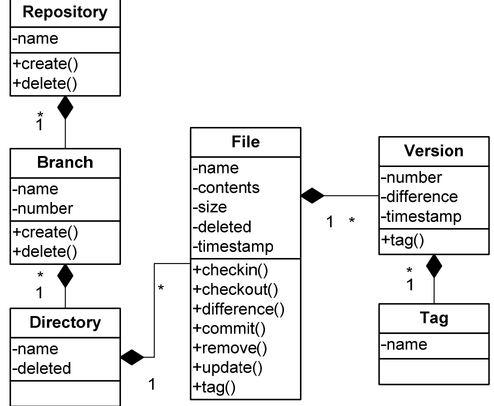

***Figure 4**. Transformation from CIM to Platform Independent Model for CVS*

4.3 PIM to PIM transformation

For the PIM to PIM transformation we apply two scenarios S3 (security) and S4 (logging) of Table 2. Figure 5 shows this transformation of the initial PIM in Figure 4 to a target PIM in which the security and logging concerns are added. In principle both concerns could be applied together, or sequentially. In case, the concerns are applied sequentially then the ordering of the scenarios will need to be considered explicitly. We will first discuss these separately and then analyze the ordering of these scenarios.

4.3.1 Adding Security

To apply the security concern to the CVS, the system should store the different users and their data access permissions. For this, in Figure 5 the classes `User` and `Permission` represent the users and their permissions respectively. Class `User` can be either a normal user (programmer) or an administrator depending on the value of the `type` attribute. Class `Permission` includes the attributes `read` and `write` indicating whether a user can read or write from or to a specific object in the system. The attributes `InstanceType` and `InstanceName` identify the object on which permission for a user has been placed. To uniquely identify a specific object, a unique identifier has to be chosen for the `InstanceName` attribute. This can be for example an object reference or the complete name inside the structure, such as *'/branch/directory/filename'*. Class `SecurityManager` checks whether a particular user with the related permissions can execute the requested operation.

Ideally we would like to separate the security concern in the target model to be able to enhance it if necessary. In fact, at a first glance, the security concern also seems to be nicely separated. There are three separate classes and one interface that implement the security protocol. Unfortunately, the classes `Repository`, `Branch`, `Directory`, and `File` that implement the interface also require implementing the permission lookup. The arrows marked with s_3 indicate which methods need to check the requested permission with the actual permissions stored by the security manager. This means that the security concern is spread over the different classes and is *tangled* with the concerns that are implemented in the corresponding classes. As such, the transformation of the initial PIM to a PIM with security concerns is realized by applying pattern 1 again. In order to apply pattern 2 so that security concern is separated in the end-result, we might need explicit abstractions to modularize this crosscutting behavior. Typically, aspect-oriented implementation techniques could be useful for this purpose.

4.3.2 Adding Logging

Figure 5 also includes the application of the logging concern as defined by scenario S4. Logging of the system is required to identify bottlenecks within the system and to get statistical information. The classes *Log* and *LogManager* implement the logging concern. Class *LogManager* includes the *log*-operation for updating the log. Class *Log* implements the log containing the entries timestamp of the operation, the host and the user, the requested operation, the branch, the status, and the result. Although the logging concern seems to be localized in the two classes *Log* and *LogManager*, a close analysis shows that the concern crosscuts over the methods of various classes. In Figure 5 the methods that are affected by adding the logging concern are denoted by S4. As such, similar to adding the security concern we can state that we have applied pattern 1 for this case.

4.3.3 Adapting Logging concern

Assume now that the logging functionality needs to be enhanced with verbosity levels to the log messages. For this the following levels need to be added: `severe`, `warning`, `config`, and `debug`. During the transformation this requirement can be accomplished by adding an attribute *level* to class *Log*. Further attributes *severityLevel, warningLevel, configLevel and debugLevel* must be added to class *LogManager* (these changes are not shown in Figure 5). The changes to the classes `Log` and `LogManager` can be automatically applied by a transformation. However, a change of logging concern also requires changing the methods that are logged. Again, all the methods indicated by the arrows labeled with S3 in Figure 5 need to be adapted to meet this concern. To resolve this issue typically the transformation itself must be aware of the crosscutting nature of the concern.

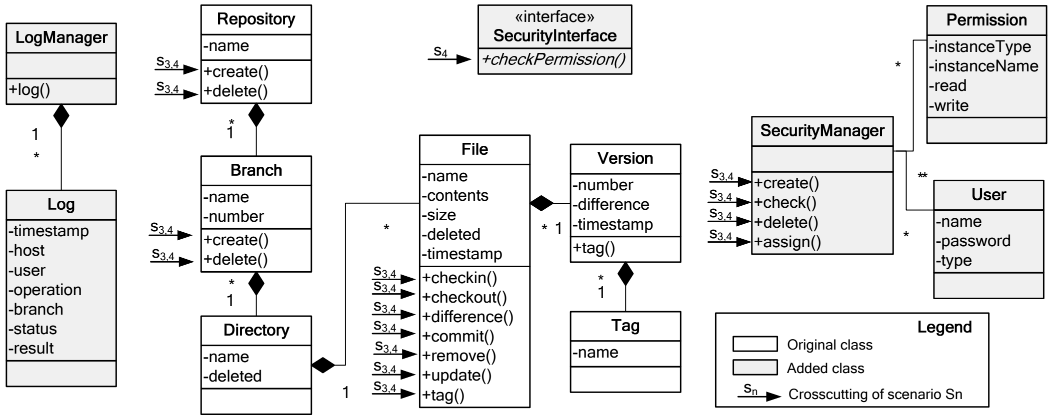

Figure 5. *PIM with security and logging*

4.3.4 Impact of sequential transformation and ordering

Interestingly, the logging concern seems also to crosscut the security concern. This has an impact on the ordering of the transformation. In case the logging concern transformation would take place before the security concern transformation, then this would imply that the methods of the *security* concern classes can not be logged. For consistency, a retransformation would then be required. Equally, security concern might crosscut the logging concern, in case logging is not allowed freely. This shows that the transformation must also aware of the ordering of the concerns.

In the above cases we have assumed that the concerns were transformed one by one. We could imagine a case in which both concerns are transformed together in the model. If we assume that security and logging concerns would be again crosscutting in the end-result then we could specify this as follows:

CVS ○ Security ○ Logging $\longrightarrow$ CVS$_{\langle Security,\ Logging \rangle}$

To provide modularity of both concerns typically the transformation must be aware of the crosscutting nature, and the semantic conflicts (ordering) of the concerns.

4.4 PIM to PSM Transformation

For the PIM to PSM transformation we have defined scenarios S5 and S6, which transform the PIM to a relational DB Platform model, and a Java Platform model, respectively. Both scenarios can be applied after scenario S4. Below we discuss both alternatives.

4.4.1 PIM to Relational PSM

Figure 6 shows the result of the transformation from PIM to Relational PSM. Here the classes have been mapped to tables and operations of classes are implicitly mapped to database operations. It is hard to spot any crosscutting in Figure 6, so one might assume that this model is crosscutting free. However, the PSM shown in the figure is only the static structure of the system. If we consider the transformation of the behavior to SQL statements then we might observe several problems. One major restriction is that queries can only do relatively simple operations on the data within the database, as SQL is never meant to be a generic programming language. Although this is not directly visible in Figure 6 the concerns security and logging seem both to be crosscutting. For example, concern security needs to check for each data access whether the permission constraints are met. This could be implemented by some value checking, before data is inserted or updated in the database via trigger or stored procedures. In this case, some of the checking behavior of the system could be moved to the database. In any case the transformation itself does not highlight the crosscutting. We could state that the crosscutting nature has changed. Compared to a Java implementation (next), for example, some crosscutting concerns are more localized in database operations. In any case we can state that the transformation pattern 1 has been applied to realize scenario S5 (transform to relational PSM).

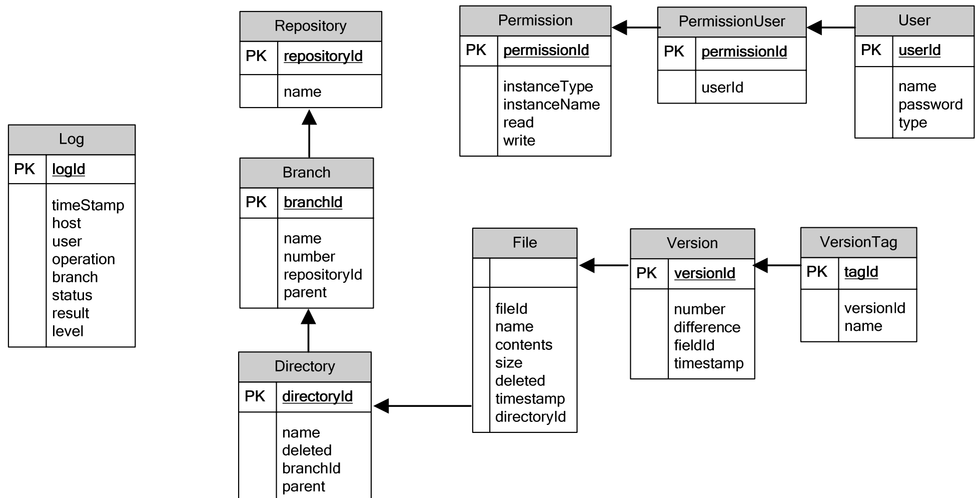

Figure 6. *Relational PSM*

4.4.2 PIM to Java PSM

Figure 8 defines the Platform Specific Model of the CVS for a Java platform. Contrary to the relational PSM the operations of the classes are explicit and crosscutting becomes more transparent. Like in the previous case, it is clear that the selection of the platform has a direct impact on the whole model

and as such can be considered as a crosscutting concern. Similarly we can state that also for scenario S5 pattern 1 is applied.

We also observe that the ordering of the transformations is important. The scenario S4 (update logging with verbosity levels), for example was applied at the PIM level as it has been explained in the previous section. This scenario could equally be applied during the PIM to PSM transformation. In that case the crosscutting occurs at the PSM level and as such becomes specific to the platform. For the relational PSM this would mean that the logging is mainly adapted in the database operations, whereas in the Java PSM the methods of the classes that call the *log()* operation need to be adapted. The bottom line is that the time at which a concern is introduced has a clear impact on the crosscutting nature.

4.5 PSM to PSM Transformation

For the PSM to PSM transformations the scenarios S7 (versioning strategy), S8 (adding persistency) and s9 (upgrading to remote security checking) are applied. These scenarios are applied either after scenario S5 (relational PSM) or scenario S6 (Java PSM). It appears that both transitions have different implications.

4.5.1 Add Versioning Strategy

The result of scenario S7 (add versioning) of Table 2 is shown in Figure 7. Here we assume that we have chosen for the Java PSM. Note that only a partial view of the complete class model is given, because the other classes remain practically unmodified. There appears to be no (crosscutting) problems, because the effect of scenario S7 is localized to only the class *Version*. It should be noted that the interface of the class *Version* has changed and that these changes may need to be processed in some other classes, such as the user interface, so from that perspective, the effect is not completely localized. In that case pattern 1 is applied.

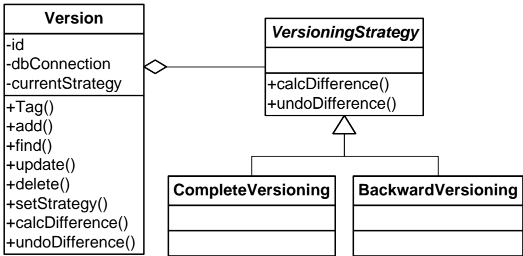

Figure 7. *Addition of Versioning Strategy to PSM*

This case also shows that adding a new concern can have an impact on previous transformations. In this scenario we should be aware that the new methods introduced by the versioning concern may need to make calls to the `SecurityManager` and `LogManager` for checking privileges and logging messages. That is to say that adding the new concern will require rechecking of conflicts and retransformations might be necessary.

If we would have chosen the relational PSM, that is, from S5 to S7, then we had to make changes there too, because the relational PSM has now to reflect the used versioning strategy per row. This is required to allow correct retrievals of the previous versions of a file. Further, also in this case a total recheck of the applied concerns and the retransformations become necessary for consistency.

4.5.2 Adding persistency

Figure 8 defines the Platform Specific Model of the CVS for a Java platform including the persistency concern. To add persistency to the Java PSM (scenario S8 of Table 2) every class has private attribute called `dbConnection`. This shows that the persistency concern also leads to crosscutting in the PSM transformation. As such we are dealing again with pattern 1. In case we would have adopted aspect-oriented techniques then we would have been able to separate these concerns better and as such could characterize the transformation as an instance of pattern 2.

In case of the relational PSM the persistency will be put in the database operations which are not directly visible.

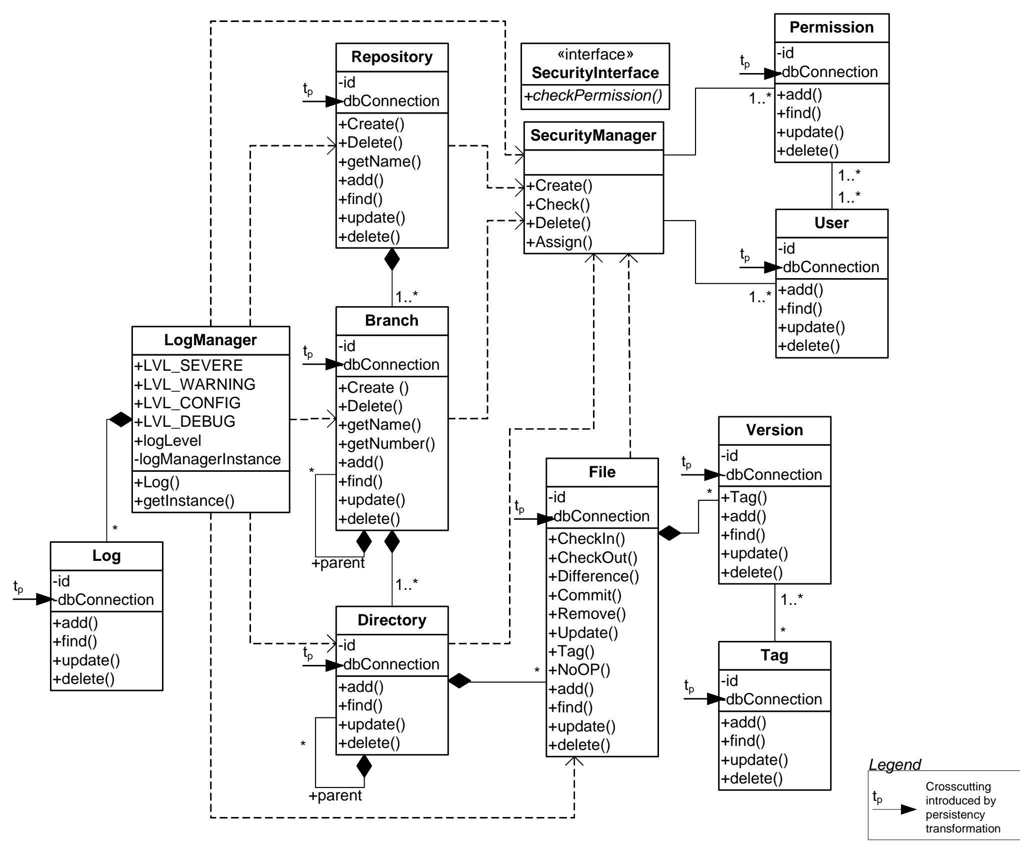

Figure 8. *Platform Specific Model for Java Platform with Persistency Concern*

4.5.3 Update Security

Scenario S9 allows the moving of the security concern to another, perhaps a remote centralized, computer. The impact of crosscutting for this scenario is two-fold. First of all, the classes that make use of the `SecurityManager` require special initialization code to obtain a reference to the `SecurityManager`. That code will be present in every class after the transformation. Although the code could be generated, it is scattered code and a replacement of the already existing crosscutting code. Furthermore, remote method invocation might more work such as in the case of exception handling to handle for example failures of the network connections. The logging concern has also to be adapted to allow remote operation invocation, because both concerns, logging and security, can be on separate computers and the security concern still needs to perform logging. This is again an example in which a newly added concern has a direct impact on the previous transformations.

4.6 PSM to Code Transformation

Scenarios S10 and S11 define the transformation to Java Code, and to relational database, respectively.

4.6.1 Transformation to the Java code model

For the transformation to Java Code model sufficient information is required to provide a complete and executable code. Unfortunately this is not always the case. For example for scenario S9 (update security) for the transformation of the Java PSM to the Java code model, the PIM and PSM do not have enough information to generate the code automatically. Normally the following example code would suffice to implement security for scenario 1 (adding security):

```
SecurityManager secMan = SecurityManager.getInstance();
```

However in case of scenario S9 in which security is handled remotely then additional information is required in the code. For example, in case we assume that the reference for security access is obtained via a file and the remote invocation is done via a CORBA ORB, then we would have to insert the following code:

```
try
{
  orb = org.omg.CORBA.ORB.init(args, props);
  try
  {
    java.io.BufferedReader in = new java.io.BufferedReader(new java.io.FileReader(refFile));
    String ref = in.readLine();
    obj = orb.string_to_object(ref);
  }
  catch(java.io.IOException ex)
  { ex.printStackTrace();
    System.exit(2);
  }
  secMan = SecurityManagerHelper.narrow(obj);
}
catch(Exception ex)
{
  ex.printStackTrace();
  System.exit(1);
}
```

Obviously this is now more difficult to generate automatically and a large part of the code must still be written by hand. In fact this is also a specific kind of crosscutting, which is actually introduced due to lack of information in the transformation. In this case pattern 3 is applied.

4.6.2 Transformation to relational database code

The transformation of the relational PSM to ‘code’ (scripts) consists of generating ‘create table’-scripts and possibly the generation of update-queries. We can consider the transformation as an instantiation of pattern 1. Here the crosscutting remains hidden, as well as the part that still needs to be adapted by hand.

5 Discussions and Recommendations

So far we have defined transformation patterns and illustrated their application to the CVS case using a predefined set of selected scenarios. During the discussion of the case we have already seen several interesting issues related to concerns, concern evolutions and model transformations in the MDA process. In the following we discuss the main issues that were identified in the analysis and based on this we derive some recommendations for coping with concern evolution in the MDA process.

- *Crosscutting is introduced in the target model because of lack of expression of the language*

If the target model is not expressive enough this might lead to scattered concerns. This could occur even in case the concern was separated at the source model. In the given scenarios the concerns *security, logging, versioning* and *persistency* were crosscutting due to lack of explicit abstraction mechanisms in Java and the relational database platform. The lack of a modeling language for aspects, leads to the problem that aspects cannot be neatly separated from the other concerns. This means that a solution has to include a modeling language for aspects for the different modeling stages of the MDA process.

- *Crosscutting in source model is inherited in target model due to lack of expressiveness*

In the scenarios that we have applied we can derive that crosscutting can be inherited from higher abstraction models. The problem starts at the CIM where some possible crosscutting is introduced and with the transformation to the PIM, to the PSM and eventually to the code model, it became clear that the crosscutting problems propagated, thus inherited, throughout the entire process and different models. Furthermore, the inheritance of crosscutting also applies to crosscutting that was later introduced than at the CIM, that is, at any phase in the model-driven engineering process crosscutting might be introduced. An example is the persistency concern that was introduced by scenario S8. The crosscutting problems of this concern were also inherited into the code model. The main reason for the inheritance of crosscutting is obviously the lack of expression power of the target language. To avoid crosscutting in each phase from CIM to code, we could delay the introduction of the crosscutting concern until the code. However, to provide a complete solution naturally it is required to apply aspect-oriented modeling techniques in which crosscutting concerns are represented using explicit language abstractions.

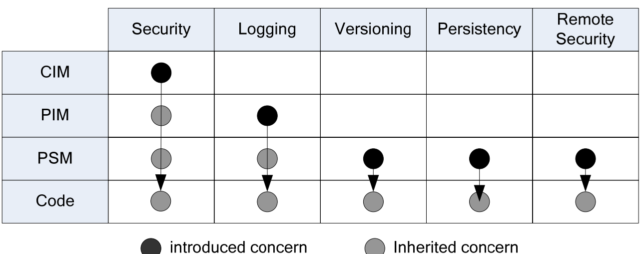

Figure 9. *Inherited Concerns*

- *Introduced crosscutting in later phases might conflict crosscutting concerns introduced in earlier models.*

Earlier introduced concerns might propagate through the lower (concrete) model abstractions, and therefore it is worthwhile to consider introducing concerns later in the process. However, later introduced concerns on the other hand could have the problem that existing transformations need to be redone. We have seen this in introducing versioning (scenario S7), persistency (scenario S8) and remote security (scenario S9). All of these scenarios required the retransformations of the previous concerns. An example of a concern that has an impact on previous transformations is shown in Figure 10. Here we see that introducing versioning at the PSM level has an impact on the transformations of the concerns security and logging in earlier phases.

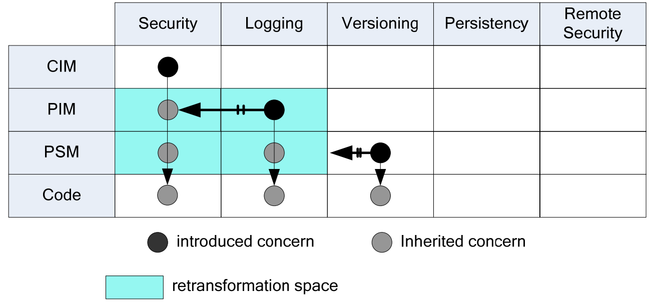

Figure 10. *Impact of introducing concerns later (e.g. versioning)*

- *Transformation order can result in different crosscutting*

The ordering of the transformation plays an important role in the quantity of crosscutting that can or will occur. The previous two points have already showed the impact of introducing concerns earlier or later in the MDA process. Moreover, the ordering is also important within the same abstraction level. For example the security and logging concerns both impact each other as it has been discussed in section 4.3. In that case we have to explicitly reason about the semantic conflicts and the impact of the ordering on the model transformations. It would be worthwhile if the transformation would be aware of the ordering semantics.

- *Crosscutting in the target model introduced by the transformation*

Obviously all the vertical model transformations including CIM to PIM, PIM to PSM and PSM to code introduce some crosscutting behavior. This crosscutting behavior is the platform concern itself. Unfortunately, it is very hard to separate the platform concern. At least we have not been able to show this in our analysis. The transformations are easier in case the models are closer to each other. For example, in the given analysis we have chosen for transforming a UML-based PIM to a Java based PSM. This is naturally different than transforming a UML-based PIM to a relational PSM, which results in different kind of crosscutting (next).

- *Transformation might lead to non-transparent crosscutting*

In section 4.4.1 in which we have mapped a PIM to a relational database PSM we have seen that crosscutting is not visible in the model. Compared to a Java implementation some crosscutting concerns are more localized in database operations. This shows that the selection of the platform does not only impacts how much the concern is crosscutting, but also how much this crosscutting is visible. In our relational PSM the crosscutting is not directly visible and as such, was compared to the Java PSM more difficult to address. The visibility of the crosscutting is of course also important for the next transformation (code).

- *Crosscutting might disappear after transformation*

Crosscutting in the source model might disappear in the target model if the transformation can map the crosscutting to a more modular concern in the lower model. As discussed before this requires that the target model includes notations to express aspects, and the transformation must be aware of the crosscutting nature of the concerns and map this to aspects in the target model. We have not shown an aspect-oriented model for this but refer to existing aspect-oriented modeling techniques.

6 Conclusion

In this paper we have provided a systematic analysis of crosscutting concerns in MDA. We have identified the points of investigation in the MDA process, starting from the computation independent model to the Platform Specific Model and code. By applying a selected set of scenarios including various concerns, we have analyzed the various problems related to crosscutting concerns in the MDA process and provided recommendations.

7 References

[1] Bernstein, Lewis & Kifer, "Database and Transaction Processing – An Application-oriented Approach", 2002, Addison-Wesley, ISBN 0-20-170872-8

[2] G. Booch, J. Rumbaugh & I. Jacobson. The Unified Modeling Language User Guide, Addison-Wesley, 1999.

[3] "Concurrent Versions System – The Open Standard for Version Control", online resource http://www.cvshome.org, February 2004

[4] Dsouza, "Model-Driven Architecture and Integration – Opportunities and Challenges", OMG document ab/2001-03-01

[5] Gerber et al., "Transformation: The Missing Link of MDA", Lecture Notes in Computer Science, 2002, vol. 2505, pp. 90-105, Springer-Verlag

[6] T. Elrad, R. Fillman, & A. Bader. Aspect-Oriented Programming. Communication of the ACM, Vol. 44, No. 10, October 2001.

[7] A. Kleppe, J. Warmer, W. Bast. MDA Explained, The Model-Driven Architecture: Practice and Promise, Addision-Wesley, 2003.

[8] I. Krechetov & B.Tekinerdoğan. Integrated Aspect-Oriented Software Architecture Specification Approach, University of Twente, European Network of Excellence AOSD project deliverable, 2005.