Aspect Orientation for Your Language of Choice

Florian Heidenreich, Jendrik Johannes, and Steffen Zschaler

Technische Universität Dresden
Institut für Software- und Multimediatechnik
D-01062, Dresden, Germany
{florian.heidenreich|jendrik.johannes|steffen.zschaler}
@tu-dresden.de

Abstract. Modern software development uses lots of so-called domain-specific languages (DSLs), providing domain-specific abstractions as a means to cope with the increasing complexity of modern software systems. While such languages are developed with a strong focus on the domain issues they are to address, more technical considerations of language engineering are typically left out. This can become problematic when the size of descriptions or programs in such a DSL increases, leading eventually to a need for advanced modularisation techniques, such as aspect orientation. To counter the complexities involved in designing modularisation for every new DSL, this paper shows a generic approach for implementing aspect orientation for arbitrary languages. The approach is especially useful for declarative DSLs, but can be used for other languages as well.

1 Introduction

Modern software development faces continually more complex domains and requirement sets. To cope with these complexities and separate them from the complexities inherent to programming and software development, many projects use so-called domain-specific languages (DSLs) as an abstraction of typical concepts from the domain and of their realisation in a software system. As a result, we see a large number of computer languages being developed, often under some amount of pressure to meet deadlines. These languages are typically focused on the specific domain issues they are intended to describe; other more technical issues of language development—for example, modularisation support—are often left out.

However, as programs and descriptions in these DSLs become larger, modularisation techniques, such as, for example, aspect orientation, are required for them. Supporting these modularisation techniques normally requires manual implementation of specialised support for every language. This is a costly and potentially error-prone step.

To reduce effort and potential for errors, this paper presents a generic approach that can be easily adapted to provide aspect orientation for arbitrary languages. This approach is based on invasive software composition (ISC) [1] and its implementation in Reuseware [2, 3]. It is, therefore, based on rewriting source code. Thus, the approach presented is especially useful for declarative DSLs that will later be interpreted. However, it can just as well be used for any other type of DSL, although more efficient results may be achieved if the larger effort of a manual implementation is invested.

To enable language-independent aspect-orientation, we must enable aspect composition to be expressed independently of the software artefacts' implementation language. For this purpose we use the language-independent composition concepts of Reuseware. We enhance these concepts by introducing the notion of *fragment queries* to group model or source code fragments. This enables us to implement the properties of quantification [4], which is a powerful notion to select sets of joinpoints. To illustrate applicability of the concepts, we demonstrate two example usages of our implementation of aspect orientation: one based on UML and one based on Java.

To give an overview of ISC and Reuseware we briefly discuss the fundamental concepts in Sect. 2. After this, we introduce and exemplify new concepts for supporting language-independent aspect-orientation in Sect. 3. The paper concludes with a discussion of related work and a summary.

2 Reuseware Composition Framework

The work we present in this paper is implemented based on the Reuseware Composition Framework [2, 3]. The concepts behind the framework have their roots in Invasive Software Composition [1]. Hence, we start by summarising the most important ISC concepts. Additionally, we introduce some new concepts of Reuseware relevant for this paper.

2.1 Composition Interfaces

Figure 1a shows an Ecore-metamodel of ISC concepts relevant for this paper. The fundamental concept in ISC is the notion of fragment components—referred to as fragments from now on. A fragment is a (possibly partial) sentence of a language. It declares *variation points* that, together, form the fragment's *composition interface*.

In ISC, the concept of variation points as building blocks for composition interfaces is defined in a language-independent manner. In addition, here we introduce three different types of variation points—slots, hooks, and anchors. A slot marks elements as replaceable. The elements can (and often have to) be replaced during composition. A hook marks positions where additional fragments can be inserted. An element associated with an anchor can be accessed to act as an extension to a hook or a replacement for a slot.

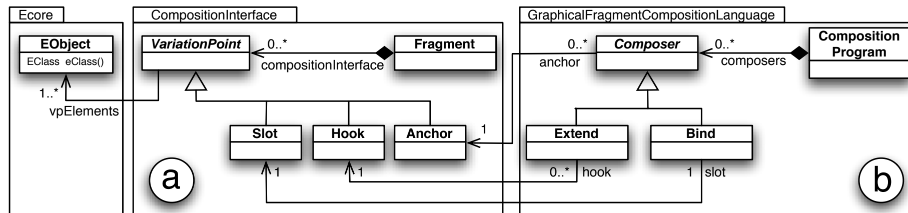


Fig. 1. Metamodels of: a) the Composition Interface b) the Fragment Composition Language

2.2 Reuse Languages

The language used for expressing a fragment must allow demarcation of the different variation points existing in the fragment. That is, such a language must provide special constructs for expressing variation points. These constructs then provide concrete syntax for the concepts from Fig. 1a. A technique for automatically constructing such a language as an extension of an existing one is presented in [2]. The original language is called the *core language* and the extended language is called the *reuse language*.

2.3 Composition Language

In order to define a composition of fragments, we need a language to express such compositions. Fragment composition is performed by so-called *composers*. In Reuseware, two *primitive composers*[1] are defined: 1) *Bind* connects slots with anchored elements, and 2) *Extend* connects hooks with anchored elements. Typically, *Bind* and *Extend* will copy the anchored element before putting it in the place of the slot or hook. However, in certain cases this is inappropriate. Instead, the anchored element should only be referenced from the place where the slot or hook have been. Consequently, each primitive composer exists in two forms: one copying elements and one referencing them. Note that these primitive composers work independently of the language utilised for fragment definition by using only the concepts defined in Fig. 1a. As long as there are means to derive the composition interfaces from the fragments, the composers can address and compose them.

Figure 1b depicts the metamodel of the fragment composition language. It refers to the concepts from Fig. 1a for expressing the composition interface of fragment components, i.e. `Slot`, `Hook`, `Anchor`, and contains concepts for the graphical composition language we provide. This language covers the available composers `Bind` and `Extend`; we have implemented the language in a generic graphical composition editor. The composition language allows to connect variation points (on the composition interface of a fragment) with composers. The resulting description of the composition (the *composition program*) is then executed by the composition engine of Reuseware.

3 Fragment-Based Aspect Orientation

In this section, we extend the concepts of ISC discussed so far with notions for aspect-orientation. Because ISC is language independent, and because we introduce these new notions in a language-independent manner, we obtain a means of expressing aspect-oriented models and programs for arbitrary languages. To support this claim, we will apply the concepts for models in UML as well as for programs in Java. First, however, we introduce the basic concepts.

3.1 Fragment Queries and Aspect-Oriented Invasive Software Composition

In [4] Filman and Friedmann state that aspect orientation (AO) [5] must provide support for *quantification* and *obliviousness*. Quantification means that aspect code can be

[1] In contrast to *complex composers* not examined in detail here.

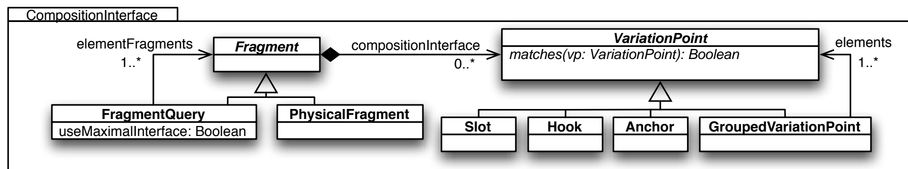


Fig. 2. Extension of the metamodel to include fragment queries

distributed in a cross-cutting fashion into many places in the code of a software system. Obliviousness means that the locations where aspect code is to be inserted need not be specially prepared for this. More recent research shows, that these properties are existent in many AO approaches but far from being the core characteristics of aspect orientation (cf. [6–8]). Furthermore, [4] distinguish *black-box* and *clear-box aspect orientation,* where 'black-box' refers to approaches that "[..] quantify over the public interface of components [..]". In ISC, code can only be inserted into places where hooks or slots have been defined and are, thus, part of a fragment's composition interface. Such variation points must be explicitly defined, requiring the code of the fragment to explicitly mention the possibility of modification. This means that ISC can be used to implement black-box, but not clear-box aspect orientation.

Quantification can be achieved by grouping a set of fragments and treating the complete group like a single fragment. We call such a grouping of fragments a *fragment query* for reasons that will become clearer later on. Figure 2 shows how we have extended the metamodel from Fig. 1a to include fragment queries. A fragment query is, thus, a fragment collecting other fragments—possibly fragment queries themselves. Basic fragments are represented as instances of the `PhysicalFragment` class.

The composition interface of a fragment query reflects the interfaces of its element fragments. However, variation points of the same name and type that occur in different element fragments are merged into one variation point for the fragment query. In addition, variation points of equal type—but different name—can be merged using regular expressions over the names. This allows for user controlled merging of variation points which is effectively a point-cut definition. Merged variation points are represented as `GroupedVariationPoint`s in the extended metamodel from Fig. 2.

To define how the composition interface of a fragment query is derived from the composition interfaces of its element fragments, we need to introduce a few helper concepts. To do so, in the following, we use the Object Constraint Language (OCL) [9] to formally express additional concepts for our metamodel classes. Notice, that for lack of space, the formalisations do not consider the case where variation points of different names are merged using regular expressions. This has, however, been implemented in our prototype.

First, we need to define which variation points should be merged. To this end, we have introduced the operation `matches()` on variation points, that returns `true` if two variation points are sufficiently equal to be merged into one. Listing 1.1 shows the definition of `matches()` for variation points.

```
1  context VariationPoint::matches (vp: VariationPoint) : Boolean
   post: result = (typeMatch (vp)) and
                  (name = vp.name) and
                  (vpElements.eClass()->forAll (c1 | vp.anchorNodes.eClass()->forAll
                      (c2 | c2 = c1)))

6  context VariationPoint::typeMatch (vp: VariationPoint) : Boolean
   post: result = (self.oclIsKindOf (Anchor) and vp.oclIsKindOf (Anchor)) or
                  (self.oclIsKindOf (Slot) and vp.oclIsKindOf (Slot)) or
                  (self.oclIsKindOf (Hook) and vp.oclIsKindOf (Hook)) or
                  (self.oclIsKindOf (GroupedVariationPoint) and elements->forAll(vpe
                      | vpe.typeMatch(vp))) or
11                (vp.oclIsKindOf (GroupedVariationPoint) and vp.elements->forAll(vpe
                      | self.typeMatch(vpe)))
```

Listing 1.1. Definition of matching between variation points. Two variation points should be merged if they have the same name, are of the same type, and the type of the elements they reference is the same.

```
   context VariationPoint::merge (vps: Set(VariationPoint)): VariationPoint
   pre:  vps->forAll (vp | self.matches (vp))
   post: (typeMatch (result)) and
4        (result.oclIsKindOf (GroupedVariationPoint)) and
         (result.name = name) and
         (result.elements = self.elements->union (vps.elements)) and
         (result.vpElements = result.elements->vpElements)
```

Listing 1.2. Merge operation defined for variation points.

The merging of matching variation points is represented by another operation on variation points: `merge()`. Listing 1.2 shows the specification of `merge()`. This operation always creates a `GroupedVariationPoint` collecting all the merged variation points. Note that anchors can only be merged with anchors, slots with slots, and hooks with hooks. The introduction of a `GroupedVariationPoint` allows composition interfaces of fragment queries to be viewed in two ways:

1. From the outside, the variation points in the composition interface of a fragment query look just like any other variation point. In particular, the elements they refer to can be accessed through the `vpElements` association end.
2. The composition system can further inspect grouped variation points and identify for each element variation point the fragment it came from and the elements it refers to.

There are at least two ways of defining a fragment query's composition interface from the composition interfaces of its element fragments. We can a) *use the maximal interface*, which, intuitively, is the union of all composition interfaces of all element fragments where matching variation points have been merged as defined by the `merge()` operation, or b) *use the minimal interface* containing only those variation points that exist in *all* element fragments, also merging them according to the `merge()` operation. The metamodel supports these types of interfaces by introducing an attribute `useMaximalInterface` for `FragmentQuery`. Listing 1.3 shows how they are used

```
context FragmentQuery
inv compositionInterface =
3   elementFragments.compositionInterface
      ->iterate (vp: VariationPoint; cmpIntf: Set(VariationPoint) = Set{} |
        if (cmpIntf->exists (vp1 | vp.matches (vp1))) then
          -- merge variation points
          cmpIntf->excluding (vp1 | vp.matches (vp1))
8               ->including (vp.merge (cmpIntf->select (vp1 | vp.matches (vp1))))
        else
          cmpIntf->including (vp)
        endif
      )
13    ->reject (vp: VariationPoint |
          (not useMaximalInterface) and
          elementFragments->exists (f | not f.compositionInterface->exists (
                  vp1 | vp.matches (vp1)
                )
18        )
        )
```

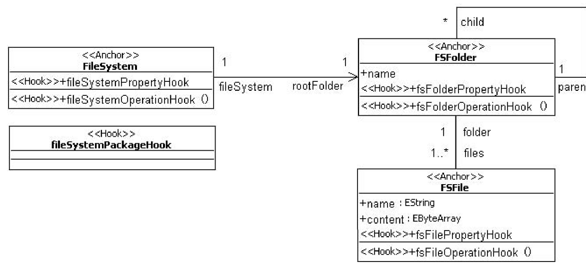
Listing 1.3. Composition interface of a fragment query.

to derive the composition interface of a fragment query. Lines 3–12 define the maximal interface. Lines 13–19 optionally restrict the interface to the minimal interface. Which of the different interface generation strategies is better may depend on the specific usage context. Determining the relative advantages and disadvantages remains for further study.

Further, we need to define how fragment queries can be expressed. In general, we can distinguish between *extensional* and *intentional queries.* An extensional query is simply an enumeration of its element fragments. An intentional query is given by some kind of expression over the fragments in a fragment repository. Intuitively, this is querying the fragment store for a set of fragments, much like an SQL query requests a set of records from a database. In our implementation, we support intentional queries using regular expressions over fragment names.

The processing of merged variation points during composition requires an enhanced composition algorithm. Since describing this algorithm in all its variations is space consuming, we limit our study to a set of scenarios that are required for Aspect-Oriented Modelling (AOM) in this paper. The restriction is that we only allow one query in a composition program to represent a core and use ordinary fragments to represent advices. Anchors in advices may only be bound to variation points in the core (and not in other advices) and only by copying. However, anchors in the core may be bound to variation points in advices by referencing or copying (to configure advices with core information).

In most cases, executing the composition means replacing variation-point elements with anchored elements. However, it may happen that a merged anchor in the core is bound to a slot in an advice. Usually, anchors which are bound to slots are only allowed to have one associated element—a limitation that can not be enforced in fragment queries. Thus we define: when a merged anchor is bound to a slot, additional copies of the advice containing the slot are produced—one for each elements associated with the anchor. These additional advice copies are bound to the core re-applying



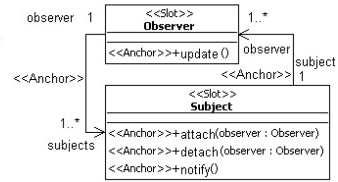


Fig. 4. Advice model: Observer

Fig. 3. Core model: FileSystem

the corresponding composers. In the next section we show how a concrete example is processed.

3.2 Aspect-Oriented Modelling with Fragment Queries

As a first example, we discuss how our approach applies to aspect-oriented modelling using two UML class diagrams. The first—the *core* model—represents the core, into which the second—the *advice* model—shall be woven. To define the *pointcuts* over the core, we apply a fragment query. Anchors in the advice model are then bound to slots and hooks in the core model. Additionally, the bound advice fragments have to be configured with core information. Thus, certain anchors in the core model are bound to slots in the advice model. The concrete example models were modelled using the TOPCASED UML editor [10] and are shown in Fig. 3 (the core) and Fig. 4 (the advice). The core model represents a conventional file system. The advice model depicts the Observer design pattern [11].

The composition program describing the weaving, as it can be modelled in our editor, is shown in Fig. 6. Boxes represent fragments or queries, circles represent variation points and arrows represent applications of composers. The `FileSystem` acts as observer by extending the appropriate hooks with the properties and operations from the `Observer` advice class. The classes `FSFile` and `FSFolder` both are assigned the `Subject` role which is expressed by the pointcut `FS.*` in the composition program. The fragment query thus merges their property (`fs*PropertyHook`) and operation (`fs*OperationHook`) hooks. These merged hooks are extended with properties and operations from the `Subject` class. `FileSystemPackageHook` is extended with the associations between `Observer` and `Subject`. Anchors in the core are bound to slots in the advice to configure them with core information. Here, the correct types for references and operations are bound. The composition result is displayed in Fig. 5.

3.3 Aspect-Oriented Programming with Fragment Queries

To evaluate the language independence of our approach, we apply it to aspect-oriented programming of Java programs. Thus, we need to perform a language extension (as in [2]), enhancing Java with the following constructs:

– `<< vpElementName >>` defines a slot element.
– `<+ vpElementName +>` defines a hook element.
– `<$ vpElementName : anchoredElement $>` defines an anchor element.

To allow comparison, we reuse the UML example and implement it in Java. Listing 1.4 shows the three core classes and Listing 1.5 the two advice classes.

Fig. 7 presents the model of the composition program. Note the similarity to the composition programs in Fig. 6. The most important difference is that the advice is defined in two fragments here.

```
public class <$  fileSystemAnchor : FileSystem $> {
  private FSFolder rootFolder;
  <+ fileSystemMemberHook +>
  public void setRootFolder(FSFolder folder) { ... }
}

public class <$ fsFileAnchor : FSFile $> { ... }
public class <$ fsFolderAnchor : FSFolder $> { ... }
```

Listing 1.4. Core classes FileSystem.rjava, FSFile.rjava, and FSFolder.rjava

```
public class Observer {
  <$ observerOperationAnchor: private <<SubjectSlot>>[] subjects; $>
  <$ observerOperationAnchor: public void update() {...} $>
}

public class Subject {
  <$ subjectMemberAnchor: private <<ObserverSlot>>[] observers; $>
  <$ subjectOperationAnchor: public void attach(<<ObserverSlot>> observer){...} $>
  <$ subjectOperationAnchor: public void detach(<<ObserverSlot>> observer){...} $>
  <$ subjectOperationAnchor: public void notify() {...} $>
}
```

Listing 1.5. Advice classes Observer.Class.java and Subject.Class.java

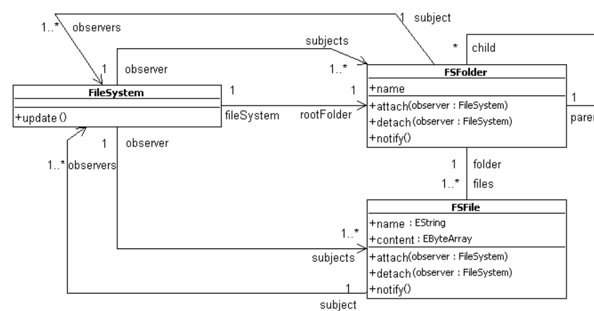


Fig. 5. Composition result

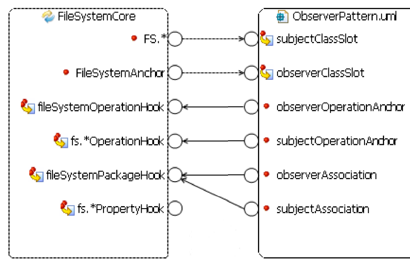


Fig. 6. Composition program for AOM with UML extended with Reuse concepts

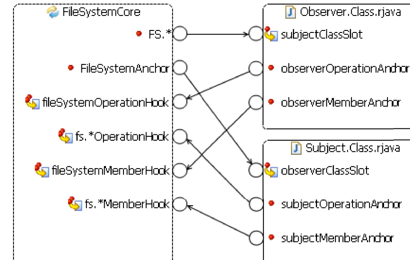


Fig. 7. Composition program for AOP in Java extended with Reuse concepts

4 Related Work

In the literature, several approaches exist that provide aspect orientation for the .NET platform and go on to claim that this makes their approaches language agnostic or independent. For example, Aspect.NET [12] uses static weaving based on binary assemblies to provide AspectJ-like AOP for .NET, Compose* [13] is an implementation of Composition Filters for .NET. Our approach goes beyond these techniques for it is not restricted to languages that can be compiled into .NET assemblies. Furthermore, our approach is sufficiently generic to allow a variety of languages for describing aspects to be defined.

Fractal Aspect Components (FAC) [14] is an extension of the Fractal Component Model [15] to support Aspect-Oriented Programming. It aims at bridging the gap between Component-Based Software Engineering and Aspect-Oriented Programming. FAC introduces several additional concepts to the Fractal Component Model to compose Aspectisable Components and Aspect Components. FAC is similar to the Reuse approach, because the component model is designed in a language-independent way. This allows for conceptual reuse within different implementations. It is, however, also different in many ways. For example, aspect binding and composition programs do not abstract from the implementation of the component model in FAC. With the graphical fragment composition editor, the Reuse approach provides a general-purpose way to express compositions independently of the fragment component's core language.

Model Weaving is also strongly related to the work presented in this paper. It allows for combining two or more models to form a composed or woven model. AMW, the Atlas Model Weaver [16] is a tool that allows generating model transformations based on a so-called *Weaving Model*. The Weaving Model consists of links between two or more models that are used to generate model transformations and model weavings.

Another approach to model weaving presented in [17] by Heidenreich and Lochmann stems from Product-Line Engineering and provides means to express *Aspectual Features* in separate models which are woven into a core model according to the feature selection of the product line. The authors are using graph-rewrite systems to weave the Aspectual Features to the core model. This idea was adopted in the design of the XWeave [18] tool by Groher and Völter. XWeave is integrated in the openArchitec-

tureWare tool chain and uses name correspondence and regular expressions for model weaving as the Reuse editor does.

However, compared to the existing model weaving approaches, the work presented in this paper goes beyond model weaving. It unifies weaving and composition operations on both model and text artefacts through the general concepts of variation points and fragment queries.

5 Conclusions and Outlook

We have presented a technology that can provide aspect-oriented concepts for an arbitrary language. The approach has been implemented based on the Reuseware framework as a plug-in for the Eclipse platform. We have demonstrated the power of the approach by applying it to two languages—UML as an example of a modelling language and Java as an example of a programming language. The most important feature of our approach is the fragment query, which allows groups of fragments to be treated like one fragment—enabling quantification.

A limitation of our current approach is the asymmetry of core and aspect composition. In the future, we plan to study how fragment queries can be used to express cores and aspects alike and how several core and aspect fragments can be composed in a symmetric fashion.

Acknowledgement

This research has been co-funded by the European Commission within the 6th Framework Programme project MODELPLEX contract number 034081 (cf. `http://www.modelplex.org`) and by the German Ministry of Education and Research (BMBF) within the project feasiPLe (cf. `http://www.feasiple.de`).

References

1. Aßmann, U.: Invasive Software Composition. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2003)
2. Henriksson, J., Johannes, J., Zschaler, S., Aßmann, U.: Reuseware – adding modularity to your language of choice. In: TOOLS EUROPE 2007: Special Issue of the Journal of Object Technology (to appear). (2007)
3. Software Technology Group, Technische Universität Dresden: Reuseware Composition Framework (2007) URL http://www.reuseware.org.
4. Filman, R., Friedman, D.: Aspect-oriented programming is quantification and obliviousness. In: Workshop on Advanced Separation of Concerns, OOPSLA 2000. (2000)
5. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In Akşit, M., Matsuoka, S., eds.: 11th Europ. Conf. on Object-Oriented Programming (ECOOP'97). Volume 1241 of LNCS., Springer (1997)
6. Aldrich, J.: Open modules: Modular reasoning about advice. In: 19th Europ. Conf. on Object-Oriented Programming (ECOOP'05). Volume 3586 of LNCS., Springer (2005) 144–168

7. Sullivan, K., Griswold, W.G., Song, Y., Cai, Y., Shonle, M., Tewari, N., Rajan, H.: Information hiding interfaces for aspect-oriented design. In: 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, New York, NY, USA, ACM Press (2005) 166–175
8. Rashid, A., Moreira, A.: Domain models are not aspect free. In Nierstrasz, O., Whittle, J., Harel, D., Reggio, G., eds.: 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS'06). Volume 4199 of LNCS., Springer (2006) 155–169
9. Object Management Group: UML 2.0 OCL specification. OMG Document (2003) URL http://www.omg.org/cgi-bin/doc?ptc/03-10-14.
10. The Topcased Project Team: TOPCASED (2007) URL http://www.topcased.org.
11. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, Reading, MA (1994)
12. Safonov, V., Gratchev, M., Grigoryev, D., Maslennikov, A.: Aspect.NET – aspect-oriented toolkit for Microsoft.NET based on Phoenix and Whidbey. In: International Conference .NET Technologies, Plzen, Czech Republic. (2006)
13. García, C.F.N.: Compose* – a runtime for the .Net platform. Master's thesis, Vrije Universiteit Brussel, Belgium (2003) More information also at http://composestar.sf.net/.
14. Pessemier, N., Seinturier, L., Coupaye, T., Duchien, L.: A model for developing component-based and aspect-oriented systems. In Löwe, W., Südholt, M., eds.: 5th Int'l Symposium on Software Composition (SC'06). Volume 4089 of LNCS., Springer (2006)
15. The Fractal Project Team: The Fractal Project (2007) URL http://fractal.objectweb.org/.
16. The AMW Project Team: Atlas Model Weaver (2007) URL http://eclipse.org/gmt/amw/.
17. Heidenreich, F., Lochmann, H.: Using graph-rewriting for model weaving in the context of aspect-oriented product line engineering. In: 1st Workshop on Aspect-Oriented Product Line Engineering (AOPLE'06) co-located with the Int'l Conf. on Generative Programming and Component Engineering (GPCE'06), Online Proc. (2006) URL http://www.softeng.ox.ac.uk/aople/aople1/.
18. Groher, I., Völter, M.: XWeave: Models and Aspects in Concert. In: 10th Workshop on Aspect-Oriented Modeling (AOM@AOSD'07) co-located with the 6th Int'l Conf. on Aspect-Oriented Software Development (AOSD'07), Online Proc. (2007) URL http://www.aspect-modeling.org/aosd07/.