Project acronym: MODELPLEX
Project full title: MODELling solution for comPLEX software systems
Contract n° 034081

Workpackage 3: Model Engineering

Deliverable D3.1.b: Model Weaving

Project co-funded by the European Community under the “Information Society Technology” Programme
**Contract Number:** 034081  
**Project Acronym:** MODELPLEX  
**Title:** MODELLing solution for comPLEX software systems

<table>
<thead>
<tr>
<th>Deliverable N°:</th>
<th>D3.1.b</th>
<th>Due Date:</th>
<th>10/2007</th>
<th>Delivery Date:</th>
<th>10/2007</th>
</tr>
</thead>
</table>

**Short Description:**  
Report on Model Weaving

**Lead Partner:** Technische Universität Dresden (TUD)  
**Made available to:** Public

<table>
<thead>
<tr>
<th>Rev</th>
<th>Date</th>
<th>Author</th>
<th>Checked by</th>
<th>Internal Approval</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>15/08/07</td>
<td>Jendrik Johannes, TUD</td>
<td>Steffen Zschaler, TUD</td>
<td></td>
<td>Initial Version</td>
</tr>
<tr>
<td>0.2</td>
<td>24/09/07</td>
<td>Jendrik Johannes, TUD</td>
<td></td>
<td>1st Task Internal Version</td>
<td></td>
</tr>
<tr>
<td>0.3</td>
<td>24/09/07</td>
<td>Jendrik Johannes, TUD</td>
<td>Hugo Brunelieire, INRIA</td>
<td></td>
<td>Integration of Section 6 from Task 3.1’s AOM tool evaluation document</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Dimitrios Kolovos, YORK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.3</td>
<td>26/09/07</td>
<td>Hugo Brunelieire, INRIA</td>
<td></td>
<td></td>
<td>Minor updates and formatting corrections</td>
</tr>
<tr>
<td>0.4</td>
<td>05/10/07</td>
<td>Jendrik Johannes, TUD</td>
<td></td>
<td></td>
<td>Integrated Comments: Steffen Zschaler, TUD Steven Völkel, TUBS</td>
</tr>
<tr>
<td>0.9</td>
<td>29/10/07</td>
<td>Jendrik Johannes, TUD</td>
<td></td>
<td>1st Internal Reviewer: Tor Neple, SINTEF</td>
<td>Updates corresponding to reviews received</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2nd Internal Reviewer:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Wasif Gilani, SAP</td>
<td></td>
</tr>
<tr>
<td>1.0</td>
<td>31/10/07</td>
<td>Final Deliverable</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Table of Content

1. Executive Summary .......................................................................................... 6
2. Introduction ....................................................................................................... 6
3. Glossary of Abbreviations ................................................................................ 6
4. Terminology and Scope .................................................................................... 7
5. Aspect-Oriented Modelling and its Application ............................................. 7
   5.1. AOM Fundamentals .................................................................................. 7
      5.1.1. Central AOP concepts ....................................................................... 7
      5.1.2. From AOP to AOM ......................................................................... 8
      5.1.3. Differences between AOP and AOM .............................................. 8
5.2. Application areas of AOM and Requirements for an AOM tool .................. 9
   5.2.1. Simplifying Model Transformations .................................................... 9
   5.2.2. Generating Model Views through Tracing and Synchronization .......... 10
   5.2.3. Improving Tool Integration ............................................................... 11
   5.2.4. Improving Model Reusability ............................................................ 11
   5.2.5. Varying Aspect Weaving Time ........................................................... 12
5.3. AOM related Requirements ..................................................................... 13
   5.3.1. Requirement 103 .............................................................................. 13
   5.3.2. Requirement 82 ................................................................................. 14
   5.3.3. Requirement 85 ................................................................................. 14
6. Evaluation of Tools for AOM ......................................................................... 15
   6.1. An Example of Aspect Oriented Modelling in UML ................................. 15
   6.2. Reuseware Composition Framework ....................................................... 16
      6.2.1. Pointcut Definition: Defining Composition Interfaces and Composition Programs ........................................................................................................ 17
      6.2.2. Executing the Aspect Weaving: Reuseware’s Composition Engine ..... 20
      6.2.3. Conclusion ......................................................................................... 20
   6.3. Atlas Model Weaver ................................................................................. 20
      6.3.1. Pointcut Definition: Defining Weaving Metamodels and Models ....... 20
      6.3.2. Interpreting the input weaving model ............................................... 21
      6.3.3. Conclusion ......................................................................................... 22
   6.4. Epsilon Platform ....................................................................................... 22
      6.4.1. Pointcut Definition: Defining Link Models and Correspondences ....... 22
      6.4.2. Executing the Aspect Weaving: Merging the File System and Observer Pattern Models ................................................................. 24
      6.4.3. Conclusion ......................................................................................... 26
6.5. Evaluation ..................................................................................................... 27
   6.5.1. Realization of AOM Concepts ............................................................. 27
   6.5.2. Comparison ......................................................................................... 28
6.6. Conclusion ...................................................................................................... 28
7. The AOM Prototype ......................................................................................... 29
   7.1. AOM Prototype Requirements .................................................................. 29
   7.2. Basic Concepts ......................................................................................... 29
      7.2.1. Model Fragments .............................................................................. 29
      7.2.2. Composition Interfaces ..................................................................... 30
      7.2.3. Composition Programs ..................................................................... 31
      7.2.4. Language Extension ......................................................................... 32
      7.2.5. Multiple Fragment Realizations ....................................................... 32
   7.3. AOM Tool Architecture ............................................................................ 33
      7.3.1. Involved Models and Metamodels ..................................................... 33
      7.3.2. Prototype Components .................................................................... 34
8. Concrete Application in MODELPLEX ......................................................... 35
   8.1. SAP Case Study ......................................................................................... 36
      8.1.1. Reuse Language Extension for UML ................................................ 36
      8.1.2. User-friendly AOM: Developing Composite Applications ............... 36
      8.1.3. Fine-grained AOM: Evaluating and Improving System Performance .... 40
      8.1.4. Conclusion and Outlook .................................................................. 42
   8.2. Telefónica Case Study ............................................................................... 42
      8.2.1. Reuse Language Extension for a DSL .............................................. 42
      8.2.2. Enabling Model-based Configuration through Language-Independent Pointcut Definitions ................................................................. 44
      8.2.3. Conclusion ......................................................................................... 45
9. Conclusion and Outlook .................................................................................. 45
10. References ....................................................................................................... 46
List of Figures

Figure 1 – Concepts of AOP .................................................................................................................. 7
Figure 2 – AOP concepts applied on UML models ............................................................................... 8
Figure 3 – Simplifying a complex model transformation through AOM ................................................. 10
Figure 4 – Generating editable views with the AOM tool ..................................................................... 10
Figure 5 – Reusing advice models on different abstraction layers ......................................................... 12
Figure 6 – Weaving aspects on different abstraction levels with the same pointcut definition .......... 12
Figure 7 – The core model: a model of a file system ............................................................................. 15
Figure 8 – An advice model: observer pattern ....................................................................................... 16
Figure 9 – The aspect weaving result .................................................................................................. 16
Figure 10 – An advice model with composition interface definition .................................................... 18
Figure 11 – The core model with composition interface definition ..................................................... 19
Figure 12 – A composition program as pointcut description ................................................................ 19
Figure 13 – The link model for the file system core model .................................................................. 23
Figure 14 – The composition interface metamodel (reuse2.ecore) ....................................................... 30
Figure 15 – The composition program metamodel ............................................................................. 31
Figure 16 – AOM Prototype Architecture ............................................................................................ 33
Figure 17 – Language Extension Wizard: Introduce variation point declarations into a language . . . 34
Figure 18 – The composition program editor ....................................................................................... 35
Figure 19 – Book order process activity (a core model) ...................................................................... 36
Figure 20 – Properties of an annotated Anchor stereotype .................................................................. 37
Figure 21 – Credit card check activity with Slots and Roots (an advice model) ..................................... 37
Figure 22 – Properties of an annotated Slot stereotype ....................................................................... 37
Figure 23 – Debit card check activity with Slots and Roots (an advice model) ....................................... 38
Figure 24 – A composition program for activities ............................................................................... 39
Figure 25 – A view on the composed order activity ............................................................................... 39
Figure 26 – Each Anchor now has a separate composition role ............................................................. 40
Figure 27 – Composing check activities parallel .................................................................................. 40
Figure 28 – Composing check activities sequentially .......................................................................... 41
Figure 29 – The composed model with sequential checks ................................................................... 42
Figure 30 – Metamodel of the CIM-based DSL ..................................................................................... 43
Figure 31 – Extended metamodel of the CIM-based DSL ................................................................. 43
Figure 32 – A network topology with Hooks (a core model) .............................................................. 44
Figure 33 – A configuration (an advice model) ..................................................................................... 44
Figure 34 – Composition program for model configurations ................................................................. 44
Figure 35 – Composed network topology ............................................................................................. 44
List of Tables
Table 1 – Overview of the three different AOM realizations.............................................................27

List of Code Listings
Listing 1 – The composition interface extractor for UML profiles......................................................18
Listing 2 – The weaving metamodel extension......................................................................................21
Listing 3 – An ATL transformation rule ..................................................................................................21
Listing 4 – The ObserverLink metamodel ..............................................................................................22
Listing 5 – ECL definition of the correspondences between core and advice model ..................23
Listing 6 – EML program for merging advice and observer models .........................................25
Listing 7 – ANT script for executing the aspect weaving with Epsilon .............................................26
Listing 8 – Configuration file with Hooks (core) ..................................................................................45
Listing 9 – A configuration fragment with Roots (advice) .................................................................45
Listing 10 – Composed configuration file ............................................................................................45
1. Executive Summary

This report forms the month 14 incarnation of deliverable D3.1.b "Model Weaving". It extends the previous incarnation from month 5. In the last version, two technologies were used to demonstrate their usage on Model Weaving tasks. One of them showed applications of Model Aspect Weaving for Aspect-Oriented Modelling and the other one used Model Weaving to deal with meta-model evolution.

In this document we clarify that Activity 3.1.b will concentrate on the Aspect-Oriented Modelling (AOM) part as stated in the DoW. We will handle both, theoretical concepts and implementation for an AOM prototype that fits the needs in MODELPLEX. We show, how requirements that were identified as related to aspect weaving can be met by this prototype and present examples that were developed in collaboration with the case study providers SAP and Telefonica.

2. Introduction

Aspect-Oriented Modelling is a technique that builds on concepts from Aspect-Oriented Programming. Its main goal is to separate cross-cutting concerns (aspects) of a system description into separate models to improve manageability, variability, reuse and other issues with such concerns in the modelling of complex systems.

To show applicability of AOM, we explain how techniques from AOP are transferred to AOM. We identify some useful applications of AOM—which differ from applications of AOP. From these application scenarios, we derive requirements for the AOM prototype and show how it can be used to tackle aspect related requirements in MODELPLEX.

We define some formal concepts for the AOM prototype that extend concepts from TUDs earlier work in software composition and AOP [12,13] and show how they can be implemented in the AOM prototype. Then we use a first version of the prototype to demonstrate the concepts and their applicability on examples that were developed together with SAP and Telefonica.

The remainder of this document is structured as follows. Section 3 contains a glossary of abbreviations. Section 4 clarifies in more detail the differences between AOM and Model Weaving and clarifies the scope of this Activity. Fundamentals of AOM, its application, and relations to MODELPLEX requirements are described in Section 5. Section 6 introduces the concepts and architecture of the prototype and Section 7 shows its application on examples. Section 7 concludes and outlines the next steps in the prototype’s development and evaluation.

3. Glossary of Abbreviations

- AMW – ATLAS Model Weaver
- AOM – Aspect Oriented Modelling
- AOP – Aspect Oriented Programming
- ATL – ATLAS Transformation Language
- ATLAS – Atlantic Data Systems (INRIA Team located in Nantes)
- CIM – Common Information Model
- DoW – MODELPLEX Description of Work document
- EMF – Eclipse Modelling Framework
- Ecore – Metamodelling Language of EMF
- WP – Work package
- TUD – Technische Universität Dresden
- INRIA – Institut National de Recherche en Informatique et en Automatique
- SNMP – Simple Network Management Protocol
- YORK – University of York
4. **Terminology and Scope**

This section clarifies the scope of Activity 3.1.b and puts this report into the context of the activity.

There has been confusion about the scope of Activity 3.1.b that originated from its title and its description in the DoW. The title “model weaving” (DoW, p. 125) is misleading because of the well-defined notion of model weaving from [5] for which INRIA has already developed the ATLAS Model Weaver (AMW) tool [4] in the course of MODELWARE.

In contrast to the title, the DoW description states: “In this activity we will combine aspect-oriented and model-based techniques. The aspect-oriented will be positioned in the traditional model-driven layered approach…” [6, p. 125]. This description clarifies that the “weaving” in the title refers to the weaving of aspects as known from Aspect-Oriented Programming. Because of the involvement of partners in those different areas (INRIA: Model Weaving; TUD: Aspect-Oriented Programming) a clarification of terminology becomes necessary for effective communication. A detailed definition of terminology can be found in Section 2 of Deliverable 3.1a.

The transfer of techniques from AOP to modelling has since become known under the term Aspect-Oriented Modelling (AOM). To avoid confusion in the future, Activity 3.1.b’s area of interest will from now on be referred to as Aspect-Oriented Modelling. For this huge research area a prototype with selected features that tackles aspect-related requirements in MODELPLEX is developed in this Activity. As a technique, aspect weaving will be performed by the prototype. This should not be confused any more with general model weaving.

This report describes AOM concepts of interest in the scope of MODELPLEX. It motivates the development of a prototype implementing such features and applies a first version of this prototype on examples. In the course of MODELPLEX, the prototype will be constantly improved to implement the requirements identified in this report.

5. **Aspect-Oriented Modelling and its Application**

Aspect-Oriented Modelling (AOM) is a modelling paradigm based on ideas from Aspect-Oriented Programming (AOP). Since it is a fairly new development, different areas of application are investigated and proposed in the research community. In this section we describe our view on AOM and the application areas we see for it in Model-Driven Engineering in general and in MODELPLEX in particular.

5.1. **AOM Fundamentals**

To understand the fundamental ideas behind AOM we first look at the concepts behind AOP, how they are used in programming, and how they can be transferred to modelling in a useful manner.

5.1.1. **Central AOP concepts**

The main motivation for AOP is scattering of code related to one concern over the code base of a system. Such concerns are called cross-cutting concerns. Because of their decentralized character, they are difficult to implement in complex systems and are even more difficult to maintain as the system evolves.

![Figure 1 – Concepts of AOP](image)

```plaintext
class X {
    method1 {
        callA();
        ...
    }
    method2 {
        callB();
        ...
    }
}
```

```plaintext
class X {
    method1 {
        callC();
        callA();
        ...
    }
    method2 {
        callC();
        callB();
        ...
    }
}
```
A common terminology in AOP (cf. Figure 1) is the distinction of core artefacts and aspect artefacts. The core is a set of source code (or byte code) files implementing a system (green). The aspect is composed of the advice code (red) and a pointcut that describes at which points the advice code is inserted into the core code when the aspect is woven into the core. The individual points are also referred to as joinpoints.

The advice code can be woven into the core code at several positions simultaneously. Through this, cross-cutting concerns can be expressed in an aspect—the advice code implementing what the concern is doing, the pointcut describing where the concern is doing something. Executing a pointcut is called quantification [7] (since it selects distinctive points in the core code). Use of quantification to isolate cross-cutting concerns is an important concept of AOP and also of AOM as we will see in Section 8.

Another property often claimed as central to AOP is obliviousness of aspects [7]. The idea is that a core system can function without additional advice code and thus does not have to be aware of the possibility that additional code can be woven in. However, this was challenged lately in recent work [8-10]. In our opinion, in AOM obliviousness is only useful to a certain degree as we will demonstrate in Section 8 and should thus not be regarded as a necessary property of AOM systems.

### 5.1.2. From AOP to AOM

We can transfer the concepts of AOP to the modelling world directly. The only difference is that we now treat core and advice models instead of code. Figure 2 illustrates this using UML class diagrams as example.

![Diagram showing AOP concepts applied on UML models](image)

**Figure 2 – AOP concepts applied on UML models**

### 5.1.3. Differences between AOP and AOM

The last section showed that the basic concepts of AOP could be applied to modeling. However, should they also be applied, seeing that there are some important differences between modeling and programming? In the following we discuss these differences and their consequences for AOM.

- **Abstraction:** A model is an abstract description of a system. Depending on the abstraction level it omits details of the actual system.

  Many common examples of aspects in AOP are dealing with implementation details (e.g., debugging or logging aspects). It was observed [14] that abstract and domain specific models tend to have less cross-cutting concerns. However the closer the models become to the implementation during the development process, the more cross-cutting concerns arise. One should keep this observation in mind when designing AOM systems.
• **Executability**: When we program, we write an executable piece of code. When we model, we do not describe a system in all details. While models can be formal and machine readable, information is missing to execute them as if they were the system themselves\(^1\).

In AOP, the actual weaving of an aspect is required to merge the code and obtain one executable piece. This cannot be the motivation for performing weaving in AOM. The two main applications we see for weaving aspect models are 1) to use it as one specific step in a transformation chain from models to code and 2) to apply it for creating views on a system by only taking a selected set of pointcuts into account when executing the aspect weaving. The next section will describe these AOM applications in more details.

• **Language (in)dependence**: AOP was originally introduced as an extension to the object-oriented paradigm. Only later it was ported to other kinds of languages. In such a case, the AOP paradigm is adjusted to the language and supporting tools are implemented (with a certain effort) from scratch. This is fine for a general-purpose programming language, where the whole system is written in this language.

In a model-driven development process of complex systems several languages, following different (domain dependent) modelling paradigms, are involved and have to collaborate. Consequently, in the field of AOM it is much more important to investigate how aspect-orientation can be used in different modelling languages. A definition of fundamental concepts that can be easily ported to arbitrary languages is desirable.

Because of these differences the motivations for and application areas of aspect-orientation vary. The next section explores important application areas and derives the requirements for the AOM prototype from them.

### 5.2. Application areas of AOM and Requirements for an AOM tool

The last section pointed at the differences between AOP and AOM. Because of this, new and interesting application areas exist for AOM. In this section we show the application areas interesting for MODELPLEX and justify the need and derive the requirements for an AOM specific tool. In Section 5.3 we will then show how concrete MODELPLEX requirements can be met with such a tool.

#### 5.2.1. Simplifying Model Transformations

We have explored the concepts of AOM formally in Section 4. Now we examine how they can be technically implemented in the AOM prototype.

Many model transformations need to manipulate and transform many diverse elements of models. Such transformations, however, tend to become complex and error prone. This is a cross-cutting issue—elements from one model influence the transformation of several elements of another model and have as such to be considered at many points of the transformation. On the other hand, transformations which do not have to consider such cross-cutting to distribute information over the whole model are much simpler.

AOM can improve the situation by centralising these cross-cutting manipulations. A specific AOM tool, which supports the developer in explicitly defining pointcut definitions, can take over the distribution of cross-cutting concerns. After the distribution, simple model transformations can be used for further transformation actions, if necessary. Figure 3 illustrates this: Without the AOM tool, the transformation has to handle the distribution (a). With AOM tool, such complexity is avoided. Instead a simpler transformation and a pointcut definition are utilized (b).

---

\(^1\) There are approaches like executable UML [15], which deal with executable models. However, those are graphical programming languages (i.e., programming languages with graphical syntax) rather than modeling languages in our understanding.
An additional advantage is that the pointcut definition is now separated from the actual transformation description, and can as such be used for other useful functionalities of the AOM tool, as we will see in the following sections.

5.2.2. Generating Model Views through Tracing and Synchronization

Tracing and synchronisation of different models describing a complex system is a well-known problem. The problem is related to AOM because it also concerns cross-cutting: Tracing becomes especially complex when information that is central in one model is distributed in another model. As a consequence, keeping such models in sync is difficult and hard to realize in an automatic fashion with conventional model transformation tools.

Figure 3 – Simplifying a complex model transformation through AOM

**AOM Prototype Requirement 1:**

The tool has to support the user in defining pointcuts. This should be realized by providing easy to use editors and other GUI tooling. These tools have to support the definition of pointcuts for advices and cores modelled in arbitrary modelling languages.

Figure 4 – Generating editable views with the AOM tool

D3.1.b Model Weaving

10/46
The AOM prototype can handle the tracing and synchronization of distributed model elements following the aspect-oriented paradigm: When aspects are woven, elements from the advice model are copied to several positions in the core. These elements themselves, however, are not modified during copying. For each of these copies, the original element can be traced. Through this, any copy can be used as a representative of the original element and modifications of a copy can be reflected on the original and all other copies.

This can be used, for instance, to obtain editable views on the system models as illustrated in Figure 4. The pointcut description in this case determines what should be presented in the view. The AOM prototype executes the aspect weaving and keeps synchronization links between originals and copies. The weaving result—which appears as a single model to the outside—can then be viewed and edited in an arbitrary model editor. The AOM tools synchronization engine, which observes the model, reflects changes onto the original model elements.

**AOM Prototype Requirement 2:**
The tool has to incorporate a synchronization mechanism that reflects changes on composed models back to the original core and advice models.

### 5.2.3. Improving Tool Integration

A problem in modelling large systems is the integration of different modelling tools required for different modelling languages and formalisms. The synchronization and generation of views described in the last section can be used to improve tool integration. As demonstrated there, existing modelling editors are used to explore and edit model views.

To realize tight tool integration we have to settle for one modelling infrastructure—which is EMF [1] because it is the one used in MODELPLEX. Tool integration for EMF-based editors will be totally automatable in many cases. If manual adjustment is required anyway, the AOM tool will provide means to do so.

**AOM Prototype Requirement 3:**
The tool has to run in an environment in which it can communicate with other modelling tools (primarily editors) at runtime to react on changes on the models and activate the synchronization mechanism when needed. If integration cannot be automated, the tool should provide easy means (e.g., integration Wizards) to allow the modeller to do such integrations without consulting an expert on the AOM tool.

### 5.2.4. Improving Model Reusability

Often parts of models are reusable to design different parts of a complex system or even of other systems. Many modelling tools provide insufficient support for reusing such models or partial models. AOM offers the possibility to extract such parts into advice models, which can then be reused in several aspect-oriented compositions of models. To enable such reuse, a repository in which reusable advices are stored and easily retrievable has to be integrated into the AOM tool.

Examples of such reusable advice models are aspects specific to one abstraction layer in an MDA stack. These aspects are usually added during transformations between abstraction layers. In this process they are often hidden in the transformations, which makes them difficult to maintain. Modelling such aspects separately in advice models and combining model transformation and AOM as mentioned, improves the handling of transformations between abstraction layers (cf. Figure 5).
AOM Prototype Requirement 4:
The tool has to be connected to a model repository to register and discover reusable advice models.

5.2.5. Varying Aspect Weaving Time
Looking again at a model-development stack with several abstraction layers, another interesting application of AOM is to vary the actual aspect weaving between the layers. This means, that the physical weaving is not necessarily performed on the abstraction layer it was defined (cf. Figure 6). Cores and advices can be transformed individually to another abstraction layer (and eventually to code). This eases transformation handling and preserves variability throughout the abstraction layers. If the weaving is executed only at the code level, variability in the end-system is achieved (i.e., different pointcut definitions can be used to obtain different variants of the system).
To enable such functionality, the AOM tool needs to realize language-independent aspect-orientation. Pointcut definitions have to be reusable on different abstraction layers, and should therefore not depend on the concrete language in which the cores and advices are defined.

**AOM Prototype Requirement 5:**
Pointcut descriptions should be reusable on different abstraction levels in a model-driven process.

### 5.3. AOM related Requirements

While the last section pointed at useful applications of AOM in general, this section describes how such applications relate to MODELPLEX project requirements defined in Deliverable D1.2.a [16]. In particular, we examine three requirements assigned to Activity 3.1.b and point out, where the functionalities of our AOM tool will be utilized when tackling these requirements.

#### 5.3.1. Requirement 103

The description of Requirement 103 “Aspect-Oriented Modeling” states:

“We [Telefonica] will have models covering different aspects of the system. This will facilitate separation of concerns between key aspects of the system, for example between functional and non-functional properties. These models will have to be put together to obtain a more complete one, that will have all the information.”

Here a separation of cross-cutting concerns and tooling to handle their merging is requested. The requirement is justified by giving two example scenarios:

“One example could be the weaving of a device configuration protocol model (the core model) with a transport protocol model (the advice model). For example, a network device can support a certain configuration protocol over more than one transport (TCP/IP, HTTP, SOAP, Telnet, etc.). So modelling the configuration protocol and the transports separately and then weaving them is desirable.”

“Another example: we may have a device model and a network topology model that will be combined into a model that will have information about the network topology and about the features of the devices involved.”

This requirement can clearly be met by tooling based on AOM concepts. We look at the second example to illustrate this: The network topology can be modelled in a core model and several models describing the devices can be regarded as advice models. The AOM tooling should provide simple means to define the pointcut that relates the advices to the core. Then several useful applications are thinkable:

- The aspect weaving can be executed for all pointcuts resulting in a model that combines all information with the cross-cutting concerns integrated. This is a good candidate for transformation to more specific models or code in the development process—the transformation can be kept simple since they do not have to handle the distribution of advice elements themselves.
- Only parts of the pointcuts can be executed to provide a view on the model. For instance, one can look at the details of one or two devices, while seeing them in the topological context. Given that the AOM tool can synchronize between the combined model and the original ones, this view is also editable.
- If the AOM tool can maintain pointcut definitions over different abstraction levels, the core and advice models can be transformed to the next abstraction layer (or to code) and actual distribution can be postponed. This is another way to simplify transformation definition: Since smaller artefacts are transformed separately, errors in the transformations are easier to track.

Section 8.2 will take a first step in tackling this requirement by demonstrating an example based on the Telefonica case study and this requirement.
5.3.2. Requirement 82
The description of Requirement 82 "Need to support definition and management of alternatives within a model" requests a system with the following features:

1) **Granularity:** Defining alternatives at (a priori) any granularity level on a model
2) **Grouping:** Characterising/grouping these alternatives
3) **Lifecycle:** Supporting the selection of one option at different points in the lifecycle (design time, deployment time, operation time)
4) **Foundational Concepts:** Need for foundational concepts
5) **Editing Tooling:** Need for tool mechanisms for editing, navigating, comparing, change management, etc.

Our AOM tool can support the functionality enumerated above as follows:

1) **Granularity:** The tool will be able to handle arbitrary modelling languages. Thus, it can be used with models designed on arbitrary abstraction (granularity) levels.
2) **Grouping:** A management system for advice models and pointcut definitions aimed at reuse will be available. Alternatives that consist of advice models and pointcut definitions can thus be exchanged and reused.
3) **Lifecycle:** The tool can support this to a certain degree, if it is able to maintain pointcut definitions over different representations of the same model/system. For instance, instead of composing the models, advices could be transformed to code individually and connected to the core by generating glue code from the pointcut description. That makes aspects exchangeable at runtime.
4) **Foundational Concepts:** The major aim of Activity 3.1.b is to identify fundamental AOM concepts and provide tooling based on them. An initial attempt for defining foundational concepts is performed in this document.
5) **Editing Tooling:** Through the tight tool integration and synchronization mechanism, the AOM tool will allow reuse of arbitrary (EMF-based) model editors.

In the requirement it is also stated that SAP would like to express different design options in a business process model to identify the optimal alternative by comparing analysis results (e.g., performance analysis). This can be achieved through AOM: The common parts of the business process can be modelled in a core model and different advice models can represent the alternatives. Different designs can then be described through exchanging the advice and adjusting the pointcut (if different alternative influence different model elements). Consequently this will be easily doable in the AOM tool. The aspect weaving can be executed to obtain one variant of the business process, which can then be analyzed by further processing (as, for instance, provided by Task 4.2).

An initial step for the described AOM application will be demonstrated in Section 8.1 on the base of toy models provided by SAP.

5.3.3. Requirement 85
The description of Requirement 85 "Need to support definition and management of alternatives within a model" reads:

"Need for facilities to support human trade-off analysis along the different cross-cutting dimensions of the system, as represented by the different cross-cutting modelling domains and their Validation & Verification support (performance, security, availability, cost). This requires visualisation and possibly editing facilities at different abstraction levels, across these different dimensions."

It is mentioned that the requirement needs refinement and indeed it is not clear what kind of system is desired. What can definitely be said at this point is that our AOM tool will help to provide visualization and editing facilities across different cross-cutting dimensions. This is because of its D3.1.b Model Weaving
tight integration with editing tools for which it can provide composed models as views. These views are editable (i.e., they reflect changes on the original artefacts) through the synchronization mechanism.

6. Evaluation of Tools for AOM

To leverage understanding of technologies that can be used in the realization of the AOM prototype, we compare three tools provided by MODELPLEX partners in this section. For this, we introduce a simple AOM example, which is an extension of an example used in the first iteration of this deliverable. Solutions for solving the example problems are then provided by different tools: Reuseware (TUD), Atlas Model Weaver (INRIA), and Epsilon Platform (YORK). This leads to a better understanding of where the strengths and weaknesses of these different technologies lay with respect to AOM.

This section is structured as follows. Section 6.1 introduces a toy example using AOM to separate concerns in a model. Realizations of the example are described in Sections 6.2 to 6.4 using the tools Reuseware (TUD), AMW (INRIA), and Epsilon (YORK), respectively. The section concludes with a comparison of the realizations and a discussion in Section 6.5.

6.1. An Example of Aspect Oriented Modelling in UML

The example we introduce utilizes UML class diagrams. This formalism was chosen because it is the most used modelling language in MODELPLEX.

The basic idea is to design a system in which the relationship between different objects is modelled in a core model, while the communication between the objects is separately modelled in an advice model. Three main advantages are gained by this approach:

1) It leads to a better understanding of the overall system, since its model is decomposed in two different models describing separate concerns of the system.

2) Communication between classes can be modelled in different ways by creating different aspect models. These aspect models can be easily exchanged as different alternatives in the weaving.

3) An advice model, once defined, can be reused in combination with different core models or at different positions within one core model.

We present a concrete example that models a file system. The core model, shown in Figure 7, models the structure of a file system that consists of a central class (FileSystem), files, and folders. While the structure of the system is described in the core, no communication between different elements is modelled.

![Figure 7 - The core model: a model of a file system](image)

D3.1.b Model Weaving
To model communication, we utilize a variation of the observer design pattern [1] as an advice model shown in Figure 8.

![Figure 8 – An advice model: observer pattern](image)

The task for the different tools is, to compose the two models such that the Observer class is woven into the FileSystem class and the Subject class is woven into all classes with names matching the regular expression FS.* (that is, FSFolder and FSFile). By weaving a class into another one, we mean adding all properties and operations from the advice to the core class and adding the associations from the advice to the core (changing the association end types from advice classes to the corresponding core classes). The result should look similar to the model displayed in Figure 9.

How the pointcut definition is formulated and how joinpoints are addressed is left to the means provided by the different technologies. For the sake of pointcut and joinpoint definitions, additional information can be added to the core and advice models.

![Figure 9 – The aspect weaving result](image)

### 6.2. Reuseware Composition Framework

This section describes the realization of the example with the Reuseware Composition Framework\(^2\) [12,13] that is developed at TUD and extended for Aspect-Oriented Modelling in the course of MODELPLEX. It is a framework that can be used to extend languages with additional notions of components (which include aspects). In this section we will concentrate on the features of Reuse-

\(^2\) [http://reuseware.org](http://reuseware.org)

D3.1.b Model Weaving
ware that are required for the realization of this example. The version of Reuseware used in this survey is an extension of the one used in the month 11 Deliverable 3.1.b. More conceptual details behind Reuseware are described in Section 8.2.

6.2.1. Pointcut Definition: Defining Composition Interfaces and Composition Programs

To define pointcuts with Reuseware we use two features of the framework: 1) composition interface definitions and 2) composition programs. Composition interfaces describe which parts of a component (called fragment) can be accessed and modified during a composition. Composition programs describe a concrete composition of fragments and can be executed by Reuseware’s composition engine.

For the AOM context and the example this means: 1) the core and advice models need to be annotated to define their composition interface (the possible joinpoints) and 2) a composition program needs to be written (the pointcut definition).

Composition interfaces consist of model elements that are marked as variation points. Each variation point has a name through which it can be addressed. If variation points have the same name, they can be addressed together. We distinguish between three kinds of variation points:

1) Slot: A slot marks elements as replaceable. The elements can (and often have to) be replaced during composition.

2) Hook: A hook marks positions where additional fragments can be inserted.

3) Anchor: An element associated with an anchor can be accessed to act as an extension to a hook or a replacement for a slot.

While such composition interfaces can be utilized for different kinds of components, they are used in AOM as follows: Anchors are used to select parts of an aspect model that should be woven into positions in the core, which are represented by hooks. Slots mark places in aspects that have to be configured with core information during the aspect weaving.

To enable the declaration of composition interface (i.e., marking model elements as variation points) in fragments, the language, in which those fragments are written, needs to be extended. To do so, Reuseware follows two approaches: 1) it offers facilities to extend metamodels and 2) it allows for the selection of original language concepts as representatives for additional composition interface concepts.

For existing (modelling) languages the second approach is more convenient, since existing tools, like model editors, do not have to be adjusted to the extended language. However, to apply this approach it needs to be defined how the composition interfaces are declared in a particular language. At the moment, an interface has to be implemented programmatically (a composition interface extractor) to identify elements of a model as parts of the composition interface.

For UML, we define a UML profile and use stereotypes to declare the composition interface. The implementation of the composition interface extractor for this profile is shown in Listing 1 (the methods that handle hooks and anchors work similar as the ones for slots and are therefore omitted). For each kind of variation point a stereotype exists in the profile. We defined, that all elements with the slot (hook, anchor) stereotype are slots (hooks, anchors) and that their name can be obtained from the name tagged value of the stereotype.

```java
public class ReusewareUML2ProfileCompositionInterfaceExtractor implements CompositionInterfaceExtractor {

    public String getSlotNodeName(EStructuralFeature node) {
        if (!(node instanceof Feature)) return null;
        Element umlElement = (Element) node;
        Stereotype vpStereotype = umlElement.
            getAppliedStereotype("reuseware::Slot");
```
if (vpStereotype == null) return null;

EObject stApplication = umlElement;
getStereotypeApplication(vpStereotype);
EAttribute nameAttr = (EAttribute) stApplication.eClass().
getEStructuralFeature("name");
return (String) stApplication.eGet(nameAttr);
}

public boolean isSlotNode(EObject node) {
if (!(node instanceof Element)) return false;

Element umlElement = (Element) node;
Stereotype vpStereotype = umlElement.
getAppliedStereotype("reuseware::Slot");
return (vpStereotype != null);
}

Listing 1 – The composition interface extractor for UML profiles

Now we can tackle the example by applying stereotypes to the advice and core models. In the advice model (cf. Figure 10) we mark all operations and associations as anchors, because we need to address them when we want to add them to the core model. Not displayed in the Figure is that all operations belonging to one class are grouped using a common name for the anchors (the groups are named observerOperationAnchor and subjectOperationAnchor). Additionally, the Observer and Subject classes are marked as slots because they need to be replaced by the core classes that become observers (respectively subjects) during the aspect weaving.

In the core model (cf. Figure 11) we add hooks to the list of operations in each class to make them extensible. We also add a hook for the content of the package, which is needed to place associations into it. Additionally all the classes are marked as anchors, because they need to be used as configuration information for the advice (to fill the slots defined at the association ends).
The example models now provide a composition interface and can be loaded into Reuseware’s graphical composition editor, to define a composition program (cf. Figure 12). The fragments (here the core and advice UML models) are displayed as boxes with circles that represent the variation points. The variation points can be connected with so-called composition operators. The set of operator connections assembles the composition program. An additional concept is that models and variation points can be grouped by queries and expressions. They are then displayed as one entity in the editor.

Since the FileSystem should act as observer, the FileSystemOperationHook is extended with the operations identified by the observerOperationAnchor. Binding the observer-ClassSlot with the FileSystemAnchor configures the advice accordingly. The anchored classes FSFile and FSFolder both are assigned the subject role which is expressed by the pointcuts FS.* (for anchors) and fs.*OperationHook (for hooks) over the composition interface of the core. The associations from the advice are added to the core model package by extending the FileSystemPackageHook. The composition could also be defined in textual form (if preferred) using Reuseware’s fragment composition language (FraCoLa).
6.2.2. Executing the Aspect Weaving: Reuseware’s Composition Engine

To execute the aspect weaving one needs to right-click on the core model in the editor and select "execute composition". Reuseware’s composition engine will then execute the weaving. Additional transformation code is not required, since all information about where which elements have to be woven is obtained from the composition interface and composition program definitions.

6.2.3. Conclusion

As a short summary, we can say that using Reuseware is a two-step process. First, a language extension has to be defined. Second, the extended language can be used. Once the first step has been completed, the extended language can be used in various scenarios. Thus, the required effort has to be judged with caution: It might cost some effort to extend a language first, but afterwards the definition of concrete aspect weavings is less complex and safer through the use of explicit composition interfaces.

Highlights of the Reuseware-based realization are the graphical composition editor and the close integration with existing model editors.

6.3. Atlas Model Weaver

The key concept of the Atlas Model Weaver tool is the definition of a core weaving metamodel that supports basic link management. The core weaving metamodel can be found at the Wiki page of AMW\(^3\). This core metamodel is extended with domain specific metamodel extensions.

6.3.1. Pointcut Definition: Defining Weaving Metamodels and Models

The key task for developing an Aspect Oriented Modelling solution is to define an AOM extension to the core weaving metamodel\(^3\). The weaving metamodel extension defines different kinds of links specific to the AOM scenario, i.e., the links specify the kinds of pointcuts and advices. A weaving model that conforms to this metamodel is created using the AMW tool. The weaving model contains the links between the core model and the advice model.

The AMW tool is implemented using the reflective API of EMF (Eclipse Modelling Framework), which means that the interface adapts to different metamodel extensions.

To be able to weave the file system model (core model) with the advice model that contains the observer pattern, we take into account three issues:

- **Where to weave**: the elements that can be woven are defined as extensions of WElementRef.
- **How to weave**: the different kinds of links are extensions of WLink.
- **How many models to weave?**: It is possible to weave several models. This is defined in the extensions to WModel.

We have used AMW in a similar application to weave a core model with different communication patterns\(^4\). To be able to weave the aspect model and the core model, we can create a weaving metamodel extension as shown in Listing 2. In this extension, we use a similar terminology of the Reuseware tool, to be able to see the similarities between our solutions.

```java
class WeavingModel extends WModel {
  -- @subsets wovenModel
  reference coreModel container : WModelRef;
  reference aspectModel container : WModelRef;
}
class AnchorLink extends WLink {
  -- @subsets end
  reference aspectElement [*] container : Slot;
  -- @subsets end
```

\(^3\) [http://wiki.eclipse.org/index.php/AMW#Core_weaving_metamodel](http://wiki.eclipse.org/index.php/AMW#Core_weaving_metamodel)

In this extension, \textit{WeavingModel} refers to the models that are woven, i.e., the \textit{coreModel} (FileSystem) and the \textit{aspectModel} (Observer Pattern). Different \textit{AnchorLinks} can be created between the elements of both models; for instance, between \textit{Operation} and \textit{Subject}. The \textit{AnchorLinks} are similar to the pointcut definition. The \textit{Hook} and \textit{Slot} are the endpoints of the links. A \textit{Hook} refers to an element of the core model, and a \textit{Slot} refers to an element of the advice model.

This extension can be loaded in the AMW tool, and the weaving links saved in a weaving model. This weaving model is created manually using the adaptive interface, which is a tree interface based on the generic EMF editor. However, this interface could be extended to support more advanced graphical visualisation, for instance, by using GMF (Graphical Modelling Framework).

One of the advantages of using a weaving model to define the weaving links is that we do not modify the initial models (core and aspect) with any information about where it is possible to compose the models.

The extension shown above supports only simple kinds of weavings. This could be extended, for instance, to support linking several elements based on wildcards. However, in this solution, we prefer creating a different link for every model element.

\subsection*{6.3.2. Interpreting the input weaving model}

The weaving model is an abstract specification of the composition process. To perform the weaving, we need to implement an ATL transformation, similar to the one shown in Listing 3. It takes as input the weaving model, the core model, and the aspect model. It produces a new composed model of the file systems and of the observer pattern.

This transformation rule copies all the classes of the core into the output model, and adds the methods from the Observer pattern that is connected by \textit{Anchor} links.

\begin{verbatim}
rule EFS {
  from
  com : Core!EClass in Core
to
  out : Core!EClass {
    eOperations <- com.eOperations-> union (com.getNewMethods),
    eSuperTypes <- com.eSuperTypes,
    "abstract" <- com."abstract"
  }
}

helper context MOF!EClass def: getNewMethods : Set(MOF!EStructuralFeature) =
  AMW!Hook.allInstancesFrom('IN')->select {
    e | MOF!EClassifier.getInstanceById('core',
      e.refImmediateComposite().
      aspectElement->first().element.ref) = self ;
}
\end{verbatim}

Listing 3 – An ATL transformation rule
6.3.3. Conclusion
To summarize, the strengths of our approach, in the AOM use case, are the following:

- We can implement an AOM metamodel extension that is used in the AMW tool. The adaptive interface supports any extension to the core metamodel.
- The specification of weaving models which are used as input to model transformations is a pattern that has been applied in different applications of model weaving.
- Such an example can be implemented using solely MDE techniques, i.e., model weaving and model transformations. It is not necessary to quit the modelling world by implementing, for instance, Java code.

6.4. Epsilon Platform
Epsilon is a component of the Eclipse GMT project that provides infrastructure for implementing task-specific model management languages. Atop this infrastructure several model management languages have been developed, two of which (ECL, EMF) are demonstrated in the context of this case study.

In this implementation, Ecore, instead of UML 2.0, is used to express the Core and Aspect models. The reason is that at the time we faced technical problems creating UML 2.0 models programmatically with Epsilon.

6.4.1. Pointcut Definition: Defining Link Models and Correspondences
For pointcut definitions we create 1) a link metamodel for defining observer-subject pairs, 2) an instance of this model to define the pair from the example, and 3) a correspondence definition between core and advice model elements using ECL (Epsilon Comparison Language).

The first step of the implementation process is to enable users to define pairs of observer-subject classes in the Core model. To achieve that, we define the following ObserverLink metamodel using Emfatic (a textual syntax for Ecore). The ObserverLink class is used to capture links between observers and subjects while the ObserverLinkModel class acts as a container or ObserverLinks.

```java
package ObserverLink;

class ObserverLinkModel {
    val ObserverLink[*] observerLinks;
}

class ObserverLink {
    ref EClass observer;
    ref EClass subject;
}
```

Listing 4 – The ObserverLink metamodel

Then, we create an instance of this metamodel (FileSystemObserverLink.model) and use ModeLink to define (using drag-n-drop) the pairs of observer-subject in our FileSystem model (FileSystem observers FSFolder and FileSystem observes FSFile as shown in Figure 7).
Since our aim is to merge the file system and observer pattern models, we first need to define correspondences between them. To do this, we use the following ECL (Epsilon Comparison Language) program (CoreWithObserverPattern.ecl).

```
rule ClassWithObserver
  match c : Core!EClass
  with o : ObserverPattern!EClass {
    guard : o.name = 'Observer'
    compare : ObserverLink!ObserverLink.allInstances.
    exists(l|l.observer = c)
  }
rule ClassWithSubject
  match c : Core!EClass
  with s : ObserverPattern!EClass {
    guard : s.name = 'Subject'
    compare : ObserverLink!ObserverLink.allInstances.
    exists(l|l.subject = c)
  }
```

Listing 5 – ECL definition of the correspondences between core and advice model

In this program (cf. Listing 5) we use all three models we have defined so far: we compare the core with the observer pattern models using the ObserverLink model. When this ECL program is executed, it creates the following correspondences in an internal memory-based match-trace: FileSystem->Observer, FSFolder->Subject, FSFile->Subject. Now we have all the information we need to merge the two models.
6.4.2. Executing the Aspect Weaving: Merging the File System and Observer Pattern Models

Having identified the correspondences between the file system and the observer pattern models in the previous step, we now exploit this information to merge the two models. We achieve merging with the following EML program:

```eml
abstract rule ClassWithOperationsContributingClass
   merge c : Core!EClass
   with s : ObserverPattern!EClass
   into t : Target!EClass {
      for (sop in s.eOperations) {
         var op : new Target!EOperation;
         op.name := sop.name;
         t.eOperations.add(op);
         for (sopp in sop.eParameters) {
            var p : new Target!EParameter;
            p.name := sopp.name;
            p.eType ::= sopp.eType;
            op.eParameters.add(p);
         }
      }
   }

auto rule ClassWithObserver
   merge c : Core!EClass
   with o : ObserverPattern!EClass
   into t : Target!EClass
   extends ClassWithOperationsContributingClass {
      guard : o.name = 'Observer'
      for (s in ObserverLink!ObserverLink.allInstances.
               select(l|l.observer = c).collect(l|l.subject).flatten()) {
         var tref : Target!EReference;
         var sref : Target!EReference;
         tref := t.createOneToManyRef
               ('subjects' + s.name, s.equivalent());
         sref := s.equivalent().
               CreateOneToManyRef
               ('observer' + c.name, c.equivalent());
         sref.eOpposite := tref;
      }
   }

auto rule ClassWithSubject
   merge c : Core!EClass
   with s : ObserverPattern!EClass
   into t : Target!EClass
   extends ClassWithOperationsContributingClass {
      guard : o.name = 'Subject'
   }

operation Target!EClass createOneToManyRef
   (name : String, type : Target!EClass) : Target!EReference {
      var ref := new Target!EReference;
      ref.name := name;
      ref.eType := type;
   }
```
Listing 6 – EML program for merging advice and observer models

By inspecting the EML program from Listing 6, we see that only the cases of interest have been explicitly addressed. For example, there is no explicit code for specifying how unrelated artefacts, such as packages, attributes or features such as visibility, scope etc. should be merged. This is achieved by using strategies. Strategies in EML are pluggable exogenous transformations that implement trivial tasks such as deep copying and structural merging so that developers don't have to specify them using EML.

Nevertheless, developers can extend the strategy-defined behaviour by using rules marked as auto, as we have done in this example. Such rules execute the logic defined in the strategy first, and then execute their body. For the specific example we have used the LeftAndMergedCommonEmfMetamodel merging strategy and the CommonEmfMetamodel transformation strategy for transforming instances of the core model into the target model.

The EML program defines three merge rules. The first one (ClassWithOperationsContributingClass) merges two classes by creating an exact copy of the left class and copying only the operations of the right class. This is an abstract rule, which means that it can only be executed if it is extended by another rule.

The second rule (ClassWithObserver) merges a class from the Core model with the Observer class from the observer pattern model. It extends the ClassWithOperationsContributingClass rule and so it creates an exact copy of the core class in the target model, enhanced with the operations of the Observer class. Additionally, in its body it creates the references to any related subjects defined in the ObserverLink model.

Finally, the third rule (ClassWithSubject) merges a class from the core model with the Subject class from the observer pattern model. Similar to the previous rule, it extends the ClassWithOperationsContributingClass rule and thus it creates a copy of the core class into the target model, decorated with the operations of the Subject class.

Having defined the comparison and merging operations, in the final step we integrated them into a seamless process using the Epsilon ANT tasks. To achieve that we define four tasks (targets in terms of ANT):

```xml
<?xml version="1.0"?>
<project default="main">
    <target name="main"
        depends="loadModels, compare, merge, disposeModels">
    </target>

    <target name="loadModels">
        <epsilon.loadModel name="Core" type="EMF"
            ...
        </epsilon.loadModel>

        <epsilon.loadModel name="ObserverPattern" type="EMF"
            ...
        </epsilon.loadModel>

        <epsilon.loadModel name="ObserverLink" type="EMF">
```
**Listing 7 – ANT script for executing the aspect weaving with Epsilon**

The `loadModels` task loads all the necessary models from their respective files. The `compare` task invokes the ECL comparison program and exports its match trace into the workflow context. Then, the `merge` task invokes the EML merging program and defines that merging should be performed on the correspondences identified in the match-trace produced by the previous comparison step (matchtrace="eclMatchTrace"). Finally, the `disposeModels` task unloads all the models loaded in the context of the workflow.

6.4.3. **Conclusion**

In this section we have provided a detailed overview of an implementation of the AOM case study using tools and languages from the Epsilon GMT component. The implementation presented covers all the steps of the process and delivers exactly the same results as the original Reuseware.
implementation\textsuperscript{5}.

Although aspect weaving was not the primary target of ECL and EML, this case study demonstrated that in combination, they address this problem in an integrated and efficient manner. In our view, this is mainly due to the fact that ECL and EML (like every other Epsilon-based language) can manage an arbitrary number of diverse models concurrently and this enables users to employ weaving models, to capture additional relationships between model elements. Also, important is the fact that they are both based on EOL, the imperative features of which can be used to address aspects for which ECL and EML were not originally designed.

The source code of the implementation using Epsilon can be accessed via CVS at http://dev.eclipse.org/viewcvs/indextech.cgi/org.eclipse.gmt/epsilon/examples/AOM/.

6.5. Evaluation

The previous sections demonstrated how different technologies were used for Aspect-Oriented Modelling and for executing aspect weaving. The survey was done on one example (observer design pattern) written in one modelling language as example (UML class diagrams). In this section we analyze and compare the three different realizations of the example.

6.5.1. Realization of AOM Concepts

The example realizations demonstrated which artefacts have to be created using the different tools. Table 1 and the reminder of this section summarize these artefacts and relate them to the AOM concepts introduced in Section 5.1. Afterwards, we compare the tools wrt. to their applicability for AOM and its underlying concepts.

<table>
<thead>
<tr>
<th>Core Definition</th>
<th>AMW</th>
<th>Epsilon</th>
<th>Reuseware</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advice Definition</td>
<td>- Arbitrary model</td>
<td>- Arbitrary model</td>
<td>- Language extension</td>
</tr>
<tr>
<td>- Models with composition interfaces based on the same metamodel</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Joinpoint Definition</td>
<td>- Weaving metamodel</td>
<td>- Link metamodel</td>
<td>- Inside core and advice models</td>
</tr>
<tr>
<td>- Weaving model</td>
<td>- Link model</td>
<td>- WCL definition</td>
<td></td>
</tr>
<tr>
<td>Pointcut Definition</td>
<td>- ATL transformations for each specific case</td>
<td>- EML program for each specific case</td>
<td>- Composition program</td>
</tr>
<tr>
<td>Execution</td>
<td>- EML program for each specific case</td>
<td>- none (Generic engine handles every case)</td>
<td></td>
</tr>
</tbody>
</table>

Table 1 – Overview of the three different AOM realizations

Core and Advice Definition

All three realizations used the same models as base for core and advice definition. Both were defined in the same modelling language (UML). This restriction, however, is only required by the Reuseware approach. The other two technologies can combine cores and advices based on different metamodels directly, if a useful semantic for such an aspect weaving exists.

Joinpoint and Pointcut Definition

All three solutions apply a kind of linking model that links elements from the advice model to joinpoints in the core model as pointcut definition. In AMW and Epsilon, joinpoints are not explicitly defined and any element in the core can act as such. In Reuseware, joinpoints are explicitly defined in the core and advice models. Consequently, AMW and Epsilon propose a kind of clear-box AOM approach while the Reuseware paradigm led to a black-box AOM approach (cf. clear-box and black-box AOP in \cite{11}).

Aspect Weaving Execution

In every approach, the execution is defined in terms of model transformations that adhere to the

\textsuperscript{5}http://reuseware.sourceforge.net/index.php?title=Aspect_Oriented_Modeling_(example)

D3.1.b Model Weaving
AOM principles. In Reuseware, the transformation is hard-coded into the tool, but generic to work with arbitrary model elements addressable through a composition interface. In AMW and Epsilon, the transformations are specific to the utilized model elements (which are UML Classes, Operations, and Associations in the example).

6.5.2. Comparison

Based on this observation we compare the technologies in the AOM context. First we compare AMW and Epsilon, because the approaches follow a similar clear-box AOM approach. Then we relate them both to the Reuseware solution.

AMW and Epsilon both place no restrictions on the core and advice models. This means that no additional effort is required from the core or advice model developer. The pointcut-definition techniques of these two technologies are also quite similar: The developer has to create a specific metamodel for every new language with which AOM should be used. The pointcuts are then defined as instances of these metamodels. The main difference lays in the definition of transformations for aspect weaving execution. AMW supports the user by automatically generating (parts of) the ATL transformation for selected cases. Epsilon comes with a rich set of model managing languages that ease the development of certain types of transformations. Consequently, which technology is better suited here depends on the model elements involved in the weaving and their complexity. We cannot make a detailed analysis here, since we only studied one example. In any case, new transformations have to be designed for every modelling language AOM should be applied to.

Language extension is Reuseware’s analogous to link metamodel definitions. This has to be performed for any new language with which AOM should be utilized. The Reuseware approach requires additional effort from the core and advice model developer who needs to define explicitly the joinpoints in the models. The benefit from this is that a common metamodel for pointcut definitions (composition programs) is utilized. This (together with the language extension) takes away the effort of defining several metamodels for different modelling languages and enables the development of a rich tooling based on the composition program metamodel. Most important, the approach comes with one generic transformation that works independent of modelling languages. Consequently, no new transformations need to be developed.

6.6. Conclusion

This survey showed that all reviewed approaches have advantages and disadvantages in the AOM context. Reuseware differs from the other approaches by restricting itself to AOM concepts. This makes it less powerful when seen in a general model management context, but more efficient for tackling AOM requirements. Especially the need for defining complex transformations is avoided.

The AOM prototype requirements described in Section 5.2 were not yet identified at the moment of tool evaluation. However, we can say that none of the requirements is fulfilled by any of the tools presented here at the moment. As a consequence, most of the prototype will be a novel implementation that will be built on top of EMF, which is chosen as the modelling environment by offering support for tackling prototype requirements 2, 3 and 4. Nevertheless, the Reuseware concepts are a promising candidate for providing an implementation of a language independent AOM framework with an easy-to-use and advanced pointcut definition tool (requirement 1 and 5). This will be shown in examples from case study providers in Section 8.

Therefore, parts of the Reuseware framework, and the experimental Reuseware AOM implementations done so far, will be reused as much as possible in the prototype realization. We also discovered that additional concepts required for AOM can be applied in other areas of composition and will therefore be integrated into the core of the Reuseware framework, which is continuously developed in the REWERSE network. Consequently, ongoing exchange between related research in MODELPLEX and REWERSE is ensured from which both projects, and not at least the AOM prototype, will profit.
The requirements also show that AOM and other model transformation can and have to be used in combination. Therefore, AMW and Epsilon, which are evolved tools in this area have to be smoothly integrated with the AOM prototype.

The next section will propose an architecture for the prototype including a description of 1) extended Reuseware concepts for AOM, 2) the integration with AMW and Epsilon as transformation tools, and 3) the artefacts that need to be newly developed.

7. The AOM Prototype

The AOM approach, which is presented in this section, is based on requirements derived from the motivations for AOM described in Section 5.2. In this section, we summarise these requirements and describe in detail how they can be fulfilled. We conclude with an overview of the prototype architecture. Note that the concepts and tools presented here are extensions of the ones used in the tool evaluation in Section 6.2. Thus, some definitions might vary.

The prototype implementation will be based on the Reuseware Composition Framework already used in Section 6.2. However, large parts will be newly developed extensions to the existing code to meet the AOM prototype requirements.

7.1. AOM Prototype Requirements

The following five requirements for the AOM prototype were identified in Section 5.2:

1) The tool has to support the user in defining pointcut definitions while providing easy to use editors and similar means. Pointcuts for advices and cores defined in arbitrary modelling languages have to be supported.

2) The tool has to incorporate a synchronization mechanism that reflects changes on composed models back to the original core and advice models.

3) The tool has to run in an environment in which it can communicate with other modelling tools (primarily editors) at runtime to react on changes on the models and activate the synchronization mechanism when needed. If integration cannot be automated, the tool should provide easy means (e.g., integration Wizards) to allow the modeller to do such integrations without consulting an expert of the AOM tool.

4) The tool has to be connected to a model repository to register and discover reusable advice models.

5) Pointcut descriptions should be reusable on different abstraction levels in a model-driven process.

In the following we introduce the concepts for the AOM prototype that meets these requirements.

7.2. Basic Concepts

As concluded in Section 6.5 we tackle requirements 1 and 5, by using and extending concepts from our previous work including Invasive Software Composition [11] and Reuseware [12]. The earlier proposes a basic model for components and composition interfaces based on the notion of Fragments and Variation Points. The latter describes how these concepts can be automatically introduced into arbitrary languages. We use these concepts and extend them to fit into the AOM context. This is briefly described in the following and exemplified in the Section 8 on scenarios derived from the SAP and Telefonica case studies.

7.2.1. Model Fragments

A model fragment is a model that is defined with the intention to be combined with other model fragments. On the conceptual side, this means that the model is aware that it only describes a part of a system, and consequently offers information about how it can be combined with user model fragments—which is its composition interface.
On a more technical level, this means that the model fragment can be incomplete wrt. the syntax and semantics of the modelling language. However, it is aware of its incompleteness and reflects this into its composition interface.

A model fragment can define a core model or an advice model. It can even be both. Whether a fragment is a core or an advice depends on the usage of its composition interface in the current composition. Therefore, a model fragment can act as a core in one composition to be extended with additional advices and then act as an advice itself in another composition where it extends another fragment.

7.2.2. Composition Interfaces

Figure 14 shows the metamodel of the composition interfaces of model fragments.

The composition interface (of a fragment) is defined through a set of composition roles. They offer the concrete information about how Model Fragments can be combined. A composition role is linked to elements inside the fragment realization called variation points. Variation points can be accessed for composition through their composition role. We distinguish the following four types of variation points:

1) **Root Element**: An element that is the root element of the model fragment’s content graph.
2) **Hook Element**: An element that can be replaced with another model fragment, identified by its Root element, during composition.
3) **Anchor Element**: An element inside a model fragment that is accessible during composition to refer to it by other elements.
4) **Slot Element**: A placeholder target for references that are routed to an Anchor element during composition.

Using Hook or Slot elements instead of concrete model elements allows the developer to define the mentioned incomplete fragments. Since these elements always belong to the composition interface, they will be replaced during composition and a complete model as composition result is guaranteed.
A composition role is characterized by the variation points it contains. We distinguish between three types of composition roles:

1) **Configuring Role**: A role that contains only Slots and Anchors. It is used to configure the fragment (by re-routing references) and not to extend it (with additional model elements).

2) **Contributing Role**: A role that contains Roots (and no Hooks). It offers content (i.e., model elements) as extension to another fragment. It may contain Slots or Anchors.

3) **Receiving Role**: A role that contains Hooks (and no Roots). It is used to extend the fragment with additional model elements. It may contain Slots or Anchors.

Roles can be grouped into role groups. The grouping indicates that roles have to be used together in one composition step explained in the next section.

### 7.2.3. Composition Programs

A composition program describes a concrete composition of fragments. The metamodel for composition programs is shown in Figure 15.

**Figure 15 – The composition program metamodel**

Composition programs consist of a set of composition collaborations that in turn consist of a set of composition role links. Prallael to the types of composition roles, we distinguish two types of composition links:

1) **Configurations**: A link that connects two configuration roles

2) **Contributions**: A link that connects a contributing to a receiving role.

Every composition collaboration is executed as an individual composition step. The links and roles involved determine which fragments are cores and which are advices. For each fragment involved in a collaboration it needs to be deterministically decidable whether it acts as core or advice. Otherwise the collaboration is erroneous. Whether a fragment is involved as core or as advice in a composition collaboration is determined as follows:

- A fragment is involved in a composition collaboration if at least one role of its composition interface is addressed by a link contained in the collaboration.
• A fragment acts as a core model (in the context of a collaboration) if at least one of the receiving roles of its composition interface is addressed by a contribution link.

• A fragment acts as an advice model (in the context of a collaboration) if at least one of the contributing roles of its composition interface is addressed by a contribution link.

• A composition collaboration is invalid if receiving and contributing roles are addressed alike or if neither receiving nor contributing roles are addressed on the same fragment (i.e., only configuration roles are addressed).

Role links contain a set of bind links and extend links that relate a Slot to an Anchor (respectively a Hook to a Root). These links are derived by matching types (i.e., metaclasses) and names of the variation points. That is, a Slot (Hook) can be bound to an Anchor (Root) if they have the same type (or a common super type). If a matching is not successful, the roles cannot be connected, because their types are incompatible.

7.2.4. Language Extension

A model fragment developer requires means to define variation points and assign them to composition roles. Arbitrary modelling languages do not provide such means. The language needs to be extended with constructs for the declaration of variation points. We identified two useful approaches for achieving this:

1) Extending the metamodel of the language to introduce new constructs.

2) Using an escape mechanism (annotation / comments) of the language or naming conventions to identify elements as variation points.

The first approach is the cleaner one, because it formally adds the notion of variation points to the language. The required extension can be automated following the formalism presented in [13]. The metamodel from Figure 14 is then integrated into the language. This approach should be chosen, if a language (most likely a DSL) is newly developed and tools do not yet exist. Here, the language extension can be performed before tooling is implemented and the additional features can be taken into account when tools, like editors, are designed. The drawback of the approach is that it breaks tool support in the case of existing languages (e.g., UML).

Thus, for existing languages, the second approach can be applied. Here, the extension has to be achieved by means of the existing language. For example, in the case of UML, a UML Profile can be applied. This integration is not as easily automatable as the metamodel extension. However, it can be tool supported to a certain degree (for instance, naming conventions is a method that could be used in arbitrary language and as such be generally supported).

Because of the usefulness of both approaches we support both in our prototype. The first can be followed in DSLs that are developed in MODELPLEX, the second is useful for existing (standardized) languages like UML, SysML and others.

7.2.5. Multiple Fragment Realizations

The composition interface metamodel (cf. Figure 14) and the composition program metamodel (cf. Figure 15) are independent of the modelling languages in which fragments are defined. Composition roles connect to the internals of the fragments through variation points (which is a language-independent concept). If the set of variation points that connect to a role is varied during the development of a model, the composition interface stays the same, since it hides the details behind composition roles.

We can make further use of this property, if we allow several (alternative) sets of variation points connected to a Role. Since there is no restriction to which model the variation points belong, and in which language such model is defined, the sets can belong to different models (for instance, describing the same system on different abstraction levels and/or in a different language).
We therefore extend our notion of model fragments and distinguish between **fragments** (characterized by the composition interface) and **fragment realizations** (different models that have the same composition interface). A fragment can have multiple fragment realizations.

The idea of multiple fragment realizations is applied to tackle requirement 5 and integrate AOM into an MDE process. In the next section we examine how this fits into the AOM prototype architecture.

### 7.3. AOM Tool Architecture

While the last section introduced the basic AOM concepts that will be implemented in the prototype, this section gives an architectural overview of the prototype and clarifies which artefacts and user-interfaces will be provided.

#### 7.3.1. Involved Models and Metamodels

Figure 16 gives an overview of the metamodels and models used and processed by the AOM prototype. On the right hand side, the metamodels and models specific to AOM are visualized. On the left hand side, arbitrary modelling languages and fragments and their exchange with other MDE tools is displayed.

Each modelling language that is used for fragment definition (here L1 and L2 as examples) has a specific reuse extension for variation point definition. This extension can be either a real metamodel extension or a definition of how variation points are definable by original language constructs (cf. Section 7.2.4).

![Figure 16 – AOM Prototype Architecture](image)

Fragment realizations are instances of extended languages. The part of a fragment realization that consists of variation points is an instance of the extended part. The remainder of the fragment only depends on the original metamodel.
The reuse extension of a language depends on the common composition interface metamodel (cf. Figure 14) of which a fragment (interface) is an instance of. Several fragment realizations that describe the same fragment (cf. Section 7.2.5) can be transformed into each other using arbitrary technologies (E.g., AMW, Epsilon, ATL). These transformations have to take the connection between variation points and composition interface into account.

Composition programs—which are instances of the composition program metamodel (cf. Figure 15)—refer to fragments and can be executed by the Reuseware Composition Engine. The execution of a composition leads to the creation of a new model fragment (which is the composition of other model fragments). The Composition Engine creates a synchronization model in which relations from a composed fragment realization to the original realizations are kept. The Composition Engine uses this model to observe changes in either of the involved realizations and synchronize them.

7.3.2. Prototype Components

The AOM Prototype will consist of four major components. They all will be implemented on top of EMF and use EMF functionality for model management, exchange, and manipulation. The user interfaces and editors will be integrated into the Eclipse Platform.

Language Extension Facility

The first step when a new modelling language should be used with the AOM prototype is to define the reuse extension. As mentioned, two different approaches are supported (cf. Section 7.2.4). We provide a Language Extension Wizard (cf. Figure 17) in which the user can select where and how variation points can be defined in the language at hand. Manual metamodel extension or grammatical extensions of the prototype will not be required.

The wizard loads the indicated metamodel and extends it by applying the Reuseware formalism for metamodel extension [13]. If the second extension approach is chosen, the wizard can be used to define which kinds of annotation mark variation points. The wizard then creates transformation rules that can identify marked elements in a fragment and derive the composition interface from them.

![Image of Language Extension Wizard](image1)

**Figure 17 – Language Extension Wizard: Introduce variation point declarations into a language**

Composition Program Editor

The most used component will be the composition program editor. Based on the composition program metamodel (cf. Figure 15), it gives the developer the possibility to define compositions of model fragments written in arbitrary languages.

D3.1.b Model Weaving
Figure 18 shows a composition program in the first version of the editor. The boxes represent different fragments loaded into the editor. The circles represent the composition roles defined by the fragments—receiving roles (.drawRect), contributing roles (drawCircle), and configuring roles (drawSquare).

The developer can link roles with contribution and configuration role links. He can define composition collaborations (blue circles) and link them to role links (blue arrows).

**Figure 18 – The composition program editor**

The editor component tackles prototype requirements 1 and 5.

**Synchronizing Composition Engine**

The composition engine that executes composition programs is almost invisible to the developer. An important property is the mentioned synchronization mechanism (prototype requirement 2). This will run continuously during model development. It is hidden to the developer and does automatic synchronization whenever possible. In cases where synchronization is not completely automatic, it will request user feedback (e.g., to decide if changes of a fragment inside a composition should reflect on all fragments or not).

**Fragment Library System**

To support easy fragment reuse (prototype requirement 4), a fragment library system will be developed. This system will build on top of the EMF resources management, but adds some additional functionality for browsing fragment databases and querying them based on composition interface information. It has to be evaluated how this relates to WP2 developments.

The next section will leverage understanding of concepts and tools introduced above by applying them on concrete examples taken from the case studies.

8. **Concrete Application in MODELPLEX**

In this section we exemplify the concepts described in Section 7 on two examples: The first one based on the SAP case study, the second one based on the Telefonica case study. Both example were developed in collaboration with the case study providers. The first example demonstrates the usage of UML as an existing language for defining model fragments and shows the flexibility wrt. obliviousness and quantification of our approach by defining different composition programs. The second example uses metamodel extension on a DSL momentarily under development in Task 3.3. It shows how several model fragment realizations and transformations between them integrates our approach into the larger scale of model-based development and configuration.
8.1. SAP Case Study

In this section we examine applications of AOM on scenarios motivated by the SAP case study. As a base we use a toy model (an UML activity diagram) of a sales process provided by SAP.

We first create a language extension for UML needed to handle activity diagrams with the AOM prototype. Then we look at two scenarios: applying AOM for 1) composite application development and 2) modelling and quickly exchanging design alternatives for performance improvement.

8.1.1. Reuse Language Extension for UML

In Section 6.2 we have already handled UML class diagrams with Reuseware. There, we have defined a UML profile of the reuse concepts. We follow the same approach here.

The profile consists of stereotypes corresponding to the four types of variation points: Hook, Root, Slot, and Anchor (cf. Figure 14). A variation point has three properties: Name, Role Name (the name of the composition role it belongs to), and Group Name (the name of the group the role belongs to). The profile defines that the stereotypes can all be applied to elements of type Activity Node and Activity Edge (both defined in the UML metamodel).

8.1.2. User-friendly AOM: Developing Composite Applications

AOM can be applied to develop SAP composite applications, which are compositions of platform processes, defined by the SAP system, and extensions provided by a client.

Figure 19 – Book order process activity (a core model)

The order processing activity modelled in Figure 19 can be regarded as a platform process. The process contains a checking activity (green) that determines whether or not certain data (here customer data) is consistent. The designer of the order processing system wants to allow a client to insert additional checks in parallel to the customer data check. For this, the client should not have

---

Note that the profile might allow other UML element types to be annotated (e.g., Classes, Operations, or Associations like in Section 6.2). However, making restrictions guides the fragment developer who, in this usage of UML, only needs to annotate Activity Nodes and Edges.

D3.1.b Model Weaving
to know about the structure of order processing and how additional checks are embedded into the activity—he should be able to model his checks independently.

![Figure 20 – Properties of an annotated Anchor stereotype](image)

To allow such extensibility, variation points (red) are defined. The start point (Fork Node) and the two end points (Merge Node for failed check; Join Node for successful checks) are marked as Anchors. They are named OUTPort, INPortNo, and INPortYes respectively and are all assigned to the composition role CheckActivity (cf. Figure 20). These Anchors are needed to connect to the actions of additional checking activities. Additionally, Hooks are defined (ActionHook and ControlFlowHook) to make the content of the activity extendible with new actions and control flows.

![Figure 21 – Credit card check activity with Slots and Roots (an advice model)](image)

![Figure 22 – Properties of an annotated Slot stereotype](image)
A client can now define its own checking activities, for example a credit card (cf. Figure 21) and a debit card (cf. Figure 23) check. He only needs to follow the convention that a checking activity has to start from an OUTPort Slot and end (dependent on its outcome) in an INPortNo or INPortYes Slot which all belong to the same composition role (cf. Figure 22). The internal structure of a check activity itself is left to him as the differences between the credit and debit card checks demonstrate.

A composition program that adds the two client checking activities to the order processing activity looks fairly simple (cf. Figure 24). Only one composition role link needs to be defined for each check activity. This is due to the fact that the order process developer already defined the pointcut by grouping all variation points into one composition role. The client thus does not have to bother how the information defined in his check activities is distributed over the core activity model.

The composition engine can now be used to execute the composition and generate a view on the composed activity. This can be useful for an expert to analyze how the client's extension influences the overall process. The generated view is shown in Figure 25. The elements that originate from different advice models are coloured. In a future version of the prototype the synchronization mechanism will make this view editable. The expert could then make corrections on the client models if mistakes are made which where not obvious when modelling a check activity out of context.

Another usage of the composition program is to use it as an actual definition of a composite application. A scenario is that the system code is generated from the client (advice) models, while an implementation for the platform process (represented by the core model) already exists. If the variation points are still identifiable in the code, the composition program can be used to generate glue code and connect the two implementations.
Figure 24 – A composition program for activities

Figure 25 – A view on the composed order activity
8.1.3. Fine-grained AOM: Evaluating and Improving System Performance

The last section demonstrated how clients could easily define process extensions in simple composition programs by shifting the pointcut definition into the core model. In this section, we will modify the variation point definitions in the core model to shift the pointcut definitions into the composition program. We show how this can be used to switch between design alternatives, for instance for performance comparison.

In this application, we again use the model from Figure 19 with the following modifications: The customer data check activity (green) is extracted into a separate advice model (similar to the credit card check in Figure 21). The Anchor annotations are varied as such, that each Anchor goes into a different composition role \((\text{OUTPort}, \text{INPortNo}, \text{INPortYes})\) as role name—cf. Figure 26—to make them addressable individually. The Hooks also go into a separate composition role called \(\text{ProcessContent}\). All composition roles are grouped into the \(\text{CheckActivites}\) role group (cf. Figure 26).

The composition interfaces of the advice models are modified in a similar fashion: instead of naming each Slot, each Slot goes into a separate role group. All Roots go into the \(\text{ProcessDefinition}\) role.

Figure 26 – Each Anchor now has a separate composition role

Figure 27 – Composing check activities parallel

The composition interfaces of the advice models are modified in a similar fashion: instead of naming each Slot, each Slot goes into a separate role group. All Roots go into the \(\text{ProcessDefinition}\) role.
Figure 27 shows a composition program that composes the customer data and credit card check activity as parallel checks into the order processing core. It is more complex than the program from Figure 24. This stems from the fact that it has to define explicitly which anchor has to be connected to which slot—the pointcut definition was shifted from the core into the composition program.

While the composition program became more complex, it also became more flexible. A possible variation of the modelled process is to perform the checks not in parallel, but in sequence. This could improve performance if short checks that tend to fail are performed first. We can make such a change simply by altering the composition program (and without manipulating the models themselves). In Figure 28, the customer data and credit card check are connected in sequence (OUT-PortYes of customer data check to INPort of credit card check). A small connector fragment consisting of one empty action has to be used between them. Note that the whole composition now has to be regarded as one composition step (one composition collaboration), because there is a dependency between the two advice models. The result of execution of the sequential check composition is shown in Figure 29.

Instead of using them as views, the composed models can be further processed. For instance, in Task 4.2 a transformation from UML Activity Diagrams to AnyLogic simulation models was defined. Changing the composition program can now quickly generate the different design alternatives, which can be transformed to and analysed (wrt. performance) by AnyLogic. The different results can then be compared.

![SequentialChecks.fcdi](image)

**Figure 28 – Composing check activities sequentially**
8.1.4. Conclusion and Outlook

This section has shown two very distinct applications of AOM on the same modelling language (and even on the same models) for different useful purposes. This demonstrates the usefulness of a generic easily adjustable AOM tool based on fundamental concepts. We have identified such concepts and provided a proof-of-concept by realizing and executing the examples from this section in the current version of the prototype.

In the next month we will continue our work in close collaboration with SAP. In particular the already close collaboration in Task 4.2, which can profit from the application of AOM as demonstrated, will be extended. This will help to find consistencies between WP3 and WP4. In particular we will investigate how the planned WP4 Verification and Validation workbench can profit from the development in Activity 3.1.b.

8.2. Telefonica Case Study

This section examines an application of AOM for model-based configuration. It is demonstrated with an example motivated by the Telefonica case study.

In this example, we will use DSLs instead of general-purpose modelling languages (like UML). Concretely, we utilize a DSL for network configuration (based on CIM and under development in Task 3.3) and a configuration file language of an SNMP monitoring tool.

8.2.1. Reuse Language Extension for a DSL

For demonstration purposes, we will use simplified toy languages. Figure 30 shows the metamodel of the network configuration modelling language. The language here is limited to modems and routers, which can be connected through connections points and have a configuration attached that can be accessed through SNMP.

If we model large and complex topologies, several modem configurations will be equal to a certain degree, because the modems are of the same type and their connection to the topology follows a common scheme. Consequently it would be a good feature to define configurations once and distribute them over the model.

For this we extend the DSL with variation point definitions. Concretely we introduce Hooks and Roots for SNMP configurations. The prototype provides a language extension wizard for this task.
(cf. Figure 17). The extended metamodel is shown in Figure 31. Note that the extension of the language is based on the composition interface model (cf. Figure 14) through inheritance.
8.2.2. **Enabling Model-based Configuration through Language-Independent Pointcut Definitions**

An editor that is developed on base of the extended language can directly support the definition of Hooks and composition roles. We can define a network topology as a core model with Hooks in Figure 32. In the model the router has concrete configuration attached, while the modems declare a Hook instead (open circle). The names at the Hooks determine the composition role to which they belong. The advice model in Figure 33 contains only a configuration for the modems.

![Figure 32 – A network topology with Hooks (a core model)](image)

The composition program in Figure 34 defines the distribution of the configuration over the topology core model. To address a set of modem configuration roles together, a query over their composition interfaces is applied. Queries are an additional feature to merge roles using patterns. In the displayed program all roles that match the pattern $M^*_{Config}$ (which are the both model configuration roles $M1_{Config}$ and $M2_{Config}$).

![Figure 33 – A configuration (an advice model)](image)

![Figure 34 – Composition program for model configurations](image)

Executing the composition gives us a view on the system visualized in Figure 35.

![Figure 35 – Composed network topology](image)
One could now transform the composed model into a configuration file for an SNMP tool that can then automatically configure the network. This is fine if our configuration is complete. However, it might be of benefit if we are able to add additional information to the configuration files directly. This is in particular necessary if the modelled configuration (cf. Figure 33) is incomplete.

In the generated configuration file, the configuration for the modems is already distributed, which makes additional editing difficult. To avoid this, we could, instead of weaving the configuration into the model of the system, transform the core and advice models individually into configuration file fragments, which are only woven before the system is actually configured.

Listing 8 (core) and Listing 9 (advice) show configuration file fragments that result from transforming the core and advice model individually. To handle such file fragments, the configuration file language was also extended. The Reuseware Composition Framework includes facilities for such extensions of textual languages that can be applied here. These fragments have the same composition interface as the model. Note that each role, because of the differences in the modelling and the configuration language, now consists of three Hooks (respectively Roots), which are TYPE, VAR, and VALUE. Anyway, the roles themselves do not change and the composition program from Figure 34 can be used to weave the configuration file fragments (composition result in Listing 10).

```
Listing 8 – Configuration file with Hooks (core)

nmpset 192.168.0.1 public system.syscontact.0 Octetstring " 'Mary Smith' "
nmpset 192.168.0.20 public <<VAR::M1Config>> <<TYPE::M1Config>> <<VALUE::M1Config>>
nmpset 192.168.0.90 public <<VAR::M2Config>> <<TYPE::M2Config>> <<VALUE::M2Config>>
```

```
Listing 9 – A configuration fragment with Roots (advice)

DEF CONTRIBUTING ROLE Config1 {
    VAR::system.syscontact.0
    TYPE::Octetstring
    VALUE::" 'James Johnson' "
}
```

```
Listing 10 – Composed configuration file

nmpset 192.168.0.1 public system.syscontact.0 Octetstring " 'Mary Smith' "
nmpset 192.168.0.20 public system.syscontact.0 Octetstring " 'James Johnson' "
nmpset 192.168.0.90 public system.syscontact.0 Octetstring " 'James Johnson' "
```

**8.2.3. Conclusion**

In this section we briefly discussed an application of AOM where the reuse of composition programs over different abstraction layers is helpful. We will extend the investigations and the collaboration with Telefonica in the upcoming month to identify the more concrete need behind their case for AOM.

**9. Conclusion and Outlook**

In this document we have derived requirements for the AOM prototype and related them to the case study provider requirements. We presented an architecture for the prototype that can realize the required functionality. A first version of the prototype has been implemented and demonstrated on example scenarios from SAP and Telefonica. This evaluation on the examples showed the usefulness of the approach and also relations to other activities in MODELPLEX.
In its current state, the prototype contains the Language Extension Wizard and the Composition Program Editor. The composition engine can compose models, but not yet synchronize them. However, a synchronization engine has been implemented and will be adjusted to the needs of the composition engine in the next three months. A basic fragment management system has been realized. This will be extended according to needs that arise when the other components of the prototype are functional and studies with larger model repositories can be exercised.

The next iteration of the deliverable will be a first stable prototype realizing the core functionality of the described architecture. The collaborations with the case study providers will continue to identify their most important needs to guide the prototype’s further development.

10. References


[5] MODELPLEX Project (Contract n° 034081)

D3.1.b Model Weaving

46/46