Workpackage 3: Model Engineering

Deliverable D3.1.b.V6: Model Weaving
**Contract Number:** 034081  
**Project Acronym:** MODELPLEX  
**Title:** MODELling solution for comPLEX software systems

**Deliverable No:** D3.1.b (v6)  
**Due Date:** 01/2010  
**Delivery Date:** 01/2010

**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>04/01/10</td>
<td>Jendrik Johannes, TUD</td>
<td></td>
<td></td>
<td>Initial Version</td>
</tr>
<tr>
<td>0.2</td>
<td>08/01/10</td>
<td>Jendrik Johannes, TUD</td>
<td>Mirko Seifert, TUD</td>
<td></td>
<td>Sections 1–4</td>
</tr>
<tr>
<td>0.3</td>
<td>11/01/10</td>
<td>Jendrik Johannes, TUD</td>
<td>Matthias Schmidt, TUD</td>
<td></td>
<td>Section 5</td>
</tr>
<tr>
<td>0.3</td>
<td>11/01/10</td>
<td>Mirko Seifert, TUD</td>
<td>Jendrik Johannes, TUD</td>
<td></td>
<td>Section 6.3</td>
</tr>
<tr>
<td>0.4</td>
<td>12/01/10</td>
<td>Jendrik Johannes, TUD</td>
<td>Matthias Schmidt, TUD</td>
<td></td>
<td>Sections 6.1 and 6.2</td>
</tr>
<tr>
<td>0.5</td>
<td>13/01/10</td>
<td>Matthias Schmidt, TUD</td>
<td>Jendrik Johannes, TUD</td>
<td></td>
<td>Sections 4.3.3, 5.5 and 6.1.4 extended</td>
</tr>
<tr>
<td>0.6</td>
<td>13/01/10</td>
<td>Mirko Seifert, TUD</td>
<td>Jendrik Johannes, TUD</td>
<td></td>
<td>Section 6.3 extended</td>
</tr>
<tr>
<td>0.9</td>
<td>31/01/10</td>
<td>Jendrik Johannes, TUD</td>
<td></td>
<td></td>
<td>1&lt;sup&gt;st&lt;/sup&gt; Internal Reviewer: Dimitrios Kolovos, YORK</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2&lt;sup&gt;nd&lt;/sup&gt; Internal Reviewer: Mathias Fritzsche, SAP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Updates corresponding to reviews received</td>
</tr>
<tr>
<td>1.0</td>
<td>01/02/10</td>
<td>Final Deliverable</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Table of Content**

1. Executive Summary ................................................................. 7
2. Introduction .............................................................................. 7
3. Glossary of Abbreviations ...................................................... 7
4. The Reuseware Composition Framework .................................. 8
   4.1. Background: Invasive Software Composition ...................... 8
   4.2. Development Process .......................................................... 9
   4.3. Architecture ......................................................................... 10
       4.3.1. Composition Core Runtime (CoCoNut) ......................... 11
       4.3.2. Diagram Composition .................................................. 11
       4.3.3. Fragment Repository and Browser ............................... 12
       4.3.4. Round-trip Support ...................................................... 12
       4.3.5. Support for Textual Modelling Languages: EMFText .... 13
   4.4. Metamodels and Languages .............................................. 14
       4.4.1. Fragment Metamodel .................................................. 15
       4.4.2. Composition Program Metamodel and Composition Language 16
       4.4.3. Composition System Metamodel and Language .............. 19
       4.4.4. Reuse Extension Metamodel and Language .................... 20
       4.4.5. Extension Point for Model Transformation Engines ....... 25
5. Reuseware User Guide .............................................................. 25
   5.1. Installation ............................................................................ 26
       5.1.1. MODELPLEX Platform ............................................... 26
       5.1.2. Eclipse Galileo .............................................................. 26
       5.1.3. Rational Software Architect ......................................... 26
   5.2. Getting Started ..................................................................... 27
   5.3. Developing Composition Systems ....................................... 28
   5.4. Defining Model Fragments and Composition Programs ....... 29
   5.5. Classifying Fragments and Browsing the Repository ......... 31
6. Applications in MODELPLEX Case Studies ............................... 35
   6.1. Télefonica Case Study ......................................................... 35
       6.1.1. Scenario ...................................................................... 35
       6.1.2. Composition Language Customization ....................... 37
       6.1.3. Developing additional CIM DSLs ................................. 38
       6.1.4. Browsing for CIM Fragments ....................................... 38
       6.1.5. Conclusion ................................................................. 40
   6.2. SAP Case Study .................................................................... 41
       6.2.1. Scenario ..................................................................... 41
       6.2.2. A Composition System for UML Activity Diagrams .... 43
6.2.3. Conclusion................................................................................................................................. 44
6.3. Thales Case Study............................................................................................................................................ 45
  6.3.1. Brief Case Study Description............................................................................................................ 45
  6.3.2. Instantiating the Reuseware framework for the Case Study................................................................. 45
  6.3.3. Composition Example......................................................................................................................... 47
  6.3.4. Round-trip Support .................................................................................................................................... 48
  6.3.5. Conclusion.............................................................................................................................................. 49
7. Conclusion ......................................................................................................................................................... 49
List of Figures

Figure 1 – A composition systems consist of a composition technique, a component model and a composition language which interact .......................................................... 9

Figure 2 – Invasive composition systems share a common composition technique and have common foundations for component models and composition languages ........................................... 9

Figure 3 – Reuseware Development Tool Chain .................................................................................................................. 10

Figure 4 – Reuseware Metamodels ........................................................................................................................................ 14

Figure 5 – Fragment Metamodel .......................................................................................................................................... 15

Figure 6 – Composition Program Metamodel ...................................................................................................................... 17

Figure 7 – Generic graphical syntax of the generic composition language ........................................................................ 18

Figure 8 – A composition program with customized syntax .................................................................................................. 18

Figure 9 – Composition System Metamodel ........................................................................................................................ 20

Figure 10 – Reuse Extension Metamodel (Component Model Specification Part) ........................................................................ 22

Figure 11 – Reuse Extension Metamodel (Composition Language Specification Part) ........................................................................ 24

Figure 12 – Installing Reuseware through the update manager in Eclipse Galileo ............................................................. 26

Figure 13 – Fragment Store activation button in Eclipse toolbar ............................................................................................. 27

Figure 14 – Fragment Repository View .................................................................................................................................. 28

Figure 15 – Eclipse New File Wizard with Reuseware category ............................................................................................ 28

Figure 16 – A reuse extension file opened in the editor for the reuse extension language ................................................................. 29

Figure 17 – Configuration dialogue for Fragment File Extensions ........................................................................................... 30

Figure 18 – Creating a new composition program .................................................................................................................. 31

Figure 19 – Classifying CIM fragments ................................................................................................................................ 32

Figure 20 – Selecting facet values for the protocol facet of the EthernetPort fragment ................................................................. 32

Figure 21 – Search process using the Fragment Browser ............................................................................................................. 33

Figure 22 – Reusing model fragments in a Reuseware Composition Program ................................................................................. 34

Figure 23 – A CIM Level 0 model decomposed into fragments ........................................................................................................ 36

Figure 24 – A composition program composing CIM Level 0 model fragments ............................................................................. 36

Figure 25 – A CIM Level 1 model: A composition program with customised syntax ........................................................................... 38

Figure 26 – CIM specific facets .................................................................................................................................................... 39

Figure 27 – An activity diagram modelling an order process ........................................................................................................ 41

Figure 28 – An activity diagram for credit card checks as business process extension ................................................................. 41

Figure 29 – Business process composition program .................................................................................................................. 43

Figure 30 – Composed business process model .............................................................................................................................. 44

Figure 31 – Viewpoint synchronisation in the Thales case study ................................................................................................. 45

Figure 32 – Reuseware applied to the Thales case study .......................................................................................................................... 46

Figure 33 – UML model ......................................................................................................................................................... 47

Figure 34 – Security RIO model ............................................................................................................................................. 47

Figure 35 – Derived integrated security model ................................................................................................................................. 48

Figure 36 – Overall Round-trip approach ................................................................................................................................. 48
List of Listings

Listing 1 – Schematic Composition System Specification .............................................................. 20
Listing 2 – Schematic Component Model Specification ................................................................. 23
Listing 3 – Schematic Composition Language Specification ......................................................... 25
Listing 4 – Composition system specification for business process composition ...................... 42
Listing 5 – Reuse extension for UML activity diagrams ................................................................. 42
1. Executive Summary

This report is the sixth and final version of deliverable D3.1.b “Model Weaving”. It is the second full report (the first full report was version two). The other four versions were prototype deliverables with accompanying short reports.

In this document we summarise the work of Activity 3.1.b that concentrated on developing model weaving and composition methods and implementing them in the Reuseware Composition Framework. We explain, how the Reuseware Composition Framework was developed, investigate its components and architecture and explain its composition and weaving concepts. By this, we show how the requirements for Reuseware, formulated in version 2 of this deliverable [1], are met in the implementation. Furthermore, we include a user and developer guide into this report and provide the details about the usage of Reuseware in the MODELPLEX case studies of Télefonica, SAP and Thales [2].

2. Introduction

Model weaving and composition allows system developers to separate cross-cutting concerns (also known as 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.

The weaving and composition technique developed in MODELPLEX and implemented in the Reuseware Composition Framework allows for a flexible exploitation of these advantages. This is due to the fact that the developed technique is language-independent and can quickly be tailored to the needs of arbitrary modelling (and also programming) languages. It can therefore be applied in the development of complex systems where different (domain-specific) languages are involved.

As part of the Reuseware Composition Framework, we developed specification languages that support developers to quickly develop composition systems for arbitrary modelling languages. Such composition systems introduce new composition and weaving techniques into the chosen modelling languages. In MODELPLEX, this was applied to develop composition systems for UML (Thales and SAP case), BPMN (SAP case), the Télefonica network DSL [3] and the Thales security DSL [4].

The remainder of this document is structured as follows. Section 3 contains a glossary of abbreviations. Section 4 explains details of the Reuseware Composition Framework and Section 5 describes how the Framework can be installed and used. Section 6 shows the application of Reuseware in the different MODELPLEX case studies and Section 7 concludes the document.

3. Glossary of Abbreviations

- CIM – Common Information Model
- DSL – Domain Specific Language
- EMF – Eclipse Modeling Framework
- EMOF – Essential Meta Object Facility
- Ecore – EMOF implementation (Metamodelling Language of EMF)
- GMF – Graphical Modeling Framework
- OCL – Object Constraint Language
- OMG – Object Management Group
- RIO – Risk, Impact, Opportunity
- TUD – Technische Universität Dresden
- UML – Unified Modeling Language
- WP – Work package
4. The Reuseware Composition Framework

In the course of MODELPLEX, the Reuseware Composition Framework was developed at TUD. The development was driven by the following requirements that were identified in version two of this deliverable [1, 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 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 first repeat and summarise the conceptual background for Reuseware in Section 4.1. We then explain in Section 4.2 which tools were used to develop Reuseware and how modelling and code generation tools were used to do model-driven development of Reuseware itself. We then describe the architecture and components of the Reuseware in Section 4.3. To obtain this final architecture, several iterations of the prototype were developed and evaluated on the MODELPLEX case studies. Finally, we give more details about the metamodels and specification languages that were developed for instantiating and using Reuseware in Section 4.4. Throughout this section we clarify how Reuseware meets the requirements from above (Requirements 1–5).

4.1. Background: Invasive Software Composition

The composition and weaving concepts that were developed in MODELPLEX as basis for Reuseware are extensions of the Invasive Software Composition (ISC) [5] idea. The basic idea of ISC is to provide a foundation for a variety of invasive composition systems that can compose fragment components written in arbitrary languages.

We classify composition systems according to [5] as a triple of composition technique, component model and composition language (cf. Figure 1). The set of invasive composition systems is a subset of all possible composition systems that could exist. They share a common composition technique—invasive merging of components—and have common foundation for their component models—which are all fragment component models—and their composition languages (cf. Figure 2). These common foundations allow a quick specification of new invasive composition systems that, with the proper tool support, can be put into practical application with little effort. A contribution of the work in Activity 3.1.b of MODELPLEX is to further explore these foundations of ISC systems and make them fit to model-driven technologies and standards.

Specification languages and tool support to quickly define invasive composition systems for arbitrary languages were initially introduced as universal ISC in [6], which was to a large degree a result of the European FP6 NoE REVERSE [7]. There, only textual, grammar-based languages were treated, which also means that the structures that were composed were abstract syntax trees. In MODELPLEX we wanted to apply similar concepts to modelling languages and models. Models however do have a graph structure and metamodels, which define modelling languages, support the specification of such graph structures. Thus, we leveraged the grammar-based approach of [6] to a metamodel-based approach that supports the composition of graphs. This is one of the major
conceptual contributions of our work. In the following we explore the Reuseware Composition Framework that implements these ideas to make them usable in the MODELPLEX case studies.

![Composition Language](image1)

**Figure 1** – A composition systems consist of a *composition technique*, a *component model* and a *composition language* which interact

![Composition Language](image2)

**Figure 2** – Invasive composition systems share a common *composition technique* and have common foundations for *component models* and *composition languages*

### 4.2. Development Process

Since it was decided in the beginning of MODELPLEX to build tools as plug-ins of the Eclipse Platform and connecting them with the Eclipse Modelling Framework (EMF) [8], Reuseware was developed as a set of Eclipse plug-ins that utilise and interact with the EMF.

Reuseware is a language-independent (Requirements 1 and 4) model weaving and composition tool. This means, that arbitrary modelling languages can be plugged into the tool by providing a metamodel for that language. For this, we required a metamodeling language. We chose Ecore—an implementation of the OMG's EMOF standard [9]—that is part of the EMF, because of its standard conformance and code generation facilities that integrate nicely into Eclipse. Ecore was also used to develop domain specific languages in MODELPLEX in the Télefonica and Thales case studies [3,4].

To create, modify and process models of a certain modelling language, a syntax description for the language is needed. Syntax can be graphical or textual. For graphical syntax, we made use of the Graphical Modeling Framework (GMF) [10] of Eclipse. For processing textual models, we developed EMFText as one component of Reuseware (cf. Section 4.3.5).

EMF, GMF and EMFText were also used for developing Reuseware itself. That is, we modelled the tool itself as much as possible instead of hand-coding it. For this, the modelling tool Fujaba [11] was utilized in addition for modelling composition semantics which can not be modelled natively in EMF because Ecore does not include behaviour modelling support. Fujaba provides the *story diagram* formalism [12], which allows the graphical modelling of graph rewriting rules. It comes with

---

D3.1.b.V6 Model Weaving 9/51
an EMF compatible code generation [13] that can generate code from these rules as dynamic semantics for EMF models. We used this to model the composition semantics of Reuseware.

Figure 3 shows the setup of the Reuseware development tool chain. The underlying metamodels of Reuseware (cf. Sections 4.3.1 and 4.4) are defined in Fujaba together with model composition semantics in form of metaclass operations that are defined using story diagrams (upper left). In EMFText and GMF, we defined textual and graphical syntax for these metamodels (middle left). The Fujaba code generation is then used to generate Ecore versions of the metamodels and Java code for the model composition semantics (upper right). EMFText and GMF code generators are used to generate model (de)serialization code and model editors (middle right). In the end, additional resource management and UI code was manually developed in Java on top of the generated code (bottom left). Additional details about this process and all (meta)models and specifications that are input to the code generations can be found online at http://resuseware.org/index.php/Metamodel.

4.3. Architecture

Reuseware consists of a core (called CoCoNut) that implements the composition technique and a set of additional components that address specific functionality that are useful for certain use cases but are not required for the core functionality. Furthermore, a set of extension points allows for customized extensions of Reuseware and the integration of Reuseware with other MODELPLEX technologies such as Epsilon [14] or tools used by MODELPLEX partners (such as Rational Software Architect (RSA) [15] which is used by Thales).
In the following, we give a brief overview of the Reuseware core in Section 4.3.1. The different additional components are described in Sections 4.3.2–4.3.5. The Reuseware core contains the metamodels and specification languages which we explore in more detail in Section 4.4.

4.3.1. Composition Core Runtime (CoCoNut)

It is important to understand that we distinguish two user roles in Reuseware: composition system developers and composition system users. A composition system implements a certain component and composition methodology. Module systems or aspect systems are examples of composition systems. Composition systems can usually only handle components (e.g., modules or aspects) that are written in a specific modelling language. Reuseware can be used by composition system developers to create new composition systems for existing or newly developed modelling languages. Consequently, composition system developers instantiate the Reuseware Composition Framework with new composition system. These composition systems are then used by composition system users to define reusable model components and composition programs (i.e., compositions of components).

In MODELPLEX, TUD played the role of the composition system developer who developed different composition systems for different languages (e.g., UML, Télefonica DSL, Thales DSL) and different composition system users, who, in the case of MODELPLEX, are the case study providers Télefonica, SAP and Thales.

CoCoNut offers tooling for the composition system developer to define composition systems using textual modelling languages which are described in more detail in Section 4.4. Furthermore, it offers basic tooling for the composition system user that includes a basic fragment repository and a graphical editor for composition programs (cf. Section 4.4.2). This tooling seamlessly integrates with model editors in Eclipse (Requirement 3) that are used to create model fragments and view composed models.

More details about the metamodel and languages that make up most of Reuseware’s core can be found in Section 4.4. The end user tooling of the core and its user interface is described in Section 5.3 (composition system developer tooling) and Section 5.4 (composition system user tooling). Additional information can be found in the following Reuseware-related MODELPLEX publications:


4.3.2. Diagram Composition

The Reuseware core is able to compose model fragments. However, a model often has a diagram associated with it. The diagram holds additional layout information not present in the model. For D3.1.b.V6 Model Weaving
instance, in a UML class diagram, the information about the classes (name, operations, etc.) is stored in the model, but the information where the class boxes are located in the diagram and how the associations between the classes are routed is stored in the diagram. This is information that is not of interest for the computer that processes the model (e.g., for code generation) but for the developer who created the model and inspects or modifies it.

The goal of the diagram composition component is to preserve layout information during composition. This involves merging of the layout information of the composed fragments and adjustments to this information to remove overlaps. Since there is currently no standard format for layout information, this component was designed to be extendable to support different formats.

We implemented support for all formats that were used in the MODELPLEX case studies where Reuseware was used. These are GMF, TOPCASED [16] and Rational Software Architect (RSA) [15]. As mentioned, GMF is commonly used for graphical syntax in Eclipse. It was used in both the Téléfonica and Thales DSLs [3,4]. The open-source TOPCASED UML tools were used in the SAP case study. RSA, a commercial EMF-based UML tool, was used by Thales in their case study. The diagram composition component was thus used in all applications of Reuseware in the MODELPLEX case studies.

The diagram composition component does not provide additional UI tooling but works in the background together with the composition algorithm implemented in the Reuseware core. The results of the diagram composition are however visible in the graphical model editors (e.g., TOPCASED UML editor, Téléfonica DSL editor) that are used to explore composed models.

More details about the diagram composition, its layout merging and adjusting concepts and its extension possibilities can be found in the following Reuseware-related MODELPLEX publication:


### 4.3.3. Fragment Repository and Browser

With this component we tackled Requirement 4. It introduces a repository and browser based on the ideas of faceted classification and browsing [17,18]. A faceted classification is based on a set of predefined facets that focus on different aspects of a model fragment. Classifying a fragment is done by selecting some of these facets and choosing one value per facet that describes the fragment the best. A set of these classifications allows an intuitive and explorative search for model fragments in huge libraries.

This component consists of three major features. First, classifying model fragments to prepare them for later reuse. Second, browsing a set of classified fragments. Third, retrieving fragments for reuse in a Reuseware composition program. These features are best described together with the additional UI tooling they provide for the composition system user. This is done in Section 5.5. This component was used in the Téléfonica case study as they stated in MODELPLEX Requirement 244 [19, p. 39]:

"Other important features for the [Téléfonica] DSL Tool would be: Library creation – to allow easy modelling in high levels of abstraction based on a pre-created library of lower-level components, like network devices, etc."

More details about the facet-based fragment repository and browsing can be found in the following diploma thesis that was written in the MODELPLEX context (a publication based on the thesis is currently under review):

4.3.4. Round-trip Support

As formulated in Requirement 2 (see version two of this deliverable [1] for more details), we desired a synchronization mechanism. This mechanism is realized in the round-trip component. It allows composition system users to modify composed models and reflect changes back to the original model fragments.

This synchronization mechanism allows it to use composed models as editable viewpoints and thus synchronize these viewpoints. This was a requirement in the Thales case study as stated in MODELPLEX Requirement 179 [19, p. 29]:

"…several models will be built for a system, at different abstraction levels (e.g., logical vs. physical, PIM vs. PSM). The abstraction/decomposition support requested here addresses the support of the analysis/design or navigation process within one system model."

Furthermore, Requirement 179 requests [2, p. 29]:

"- Need to manage impact on inter-model relationships
- Need for tool mechanisms for editing, navigating, change mgt., etc."

Consequently, we used Reuseware together with this round-trip component in the Thales case study for viewpoint synchronization.

The process of synchronization cannot be fully automated in a general fashion. Rather, specific strategies need to be provided for a full automation. This can be done via an extension point provided by this component. We implement such a case specific extension for the Thales case study scenario (this extension is not public available). More details can be found in Section 6.3.

More details about the round-trip extension for Reuseware can be found in the following Reuseware-related MODELPLEX publication:


4.3.5. Support for Textual Modelling Languages: EMFText

Two of the modelling languages of Reuseware—which will be discussed in Section 4.4—have textual syntax. Thus, Reuseware always contained a part to process textual models. Since it turned out that this functionality is useful in general to define textual syntax for modelling languages, we separated this part into a stand-alone component we called EMFText (because it can be used to define textual representations for EMF models).

Now, this component can also be used to instantiate Reuseware for textual languages—also programming languages. We showed this by realizing the Java Model Parser and Printer (JaMoPP) with EMFText. JaMoPP allows us to treat Java as a modelling language, which means that Java programs can be processed directly by modelling tools. In the case of Reuseware, this means that composition systems can also be defined for Java programs in a similar way as for models. And since (Java) code is also an abstraction level in model-driven development, this tackles Requirement 5 as well.

EMFText and JaMoPP also attracted a large user community independent of Reuseware in a short time. The tools are also used and further developed in other research projects at TUD already.

More details about EMFText and JaMoPP can be found in the following MODELPLEX publications:

4.4. Metamodels and Languages

The Reuseware core is built around four metamodels shown in Figure 4 (middle). Each metamodel is modelled as a class diagram in Fujaba. Each of these class diagrams is then translated to an Ecore model by Fujaba's EMF code generation (cf. Section 4.1). Instances of the metamodel are either derived by Reuseware or have a concrete syntax (i.e., we provide a modelling language with editor based on the metamodel) that can be used by a composition system developer or a composition system user directly.

A composition system developer may utilise two dedicated languages for composition system development—one language to define the concepts of a CompositionSystem and one language to specify where these concepts are found in a modelling language that is defined by an Ecore metamodel. We call such a specification a ReuseExtension for a given language. Both, CompositionSystem and ReuseExtension language, have a textual syntax defined with EMFText. The ReuseExtension language embeds OCL as expression language. These two metamodels are only used for specification and do not have operations that define composition semantics.

A composition system user works with two kinds of artefacts: Model Fragments (the components in Reuseware) and CompositionPrograms (specifications for compositions of fragments). A fragment has a composition interface through which model elements are accessed (or modified) during a composition. Concepts for composition interfaces are modelled in the Fragment metamodel. Instances of that metamodel are created by interpreting ReuseExtension specifications on arbitrary EMF models. For this, operations exist in the Fragment metamodel that are defined by story diagrams (cf. Section 4.1).

Composition programs can again be created by interpreting ReuseExtension models but also manually by a composition system user. It depends on whether the composition system developer specified a dedicated composition language for the composition system or decided to use the generic composition language of Reuseware. This composition language is defined in the CompositionProgram metamodel and has a graphical syntax defined with GMF. In any case, parts of a composition program can be derived and updated automatically. This is specified in operations in the CompositionProgram metamodel.

The details about the metamodels are described in the following subsections. We can however only give an overview here. More detail and the complete metamodels (including the story diagrams defining the operations) can be found online at http://reuseware.org/index.php/Metamodel.
4.4.1. Fragment Metamodel

Each model that is treated by Reuseware is a model fragment. A model fragment has a composition interface that consists of a set of ports. Each port is linked to a set of model elements inside the model that are modified or accessed during composition. All elements that are not connected to a port are hidden and cannot be modified during composition.

Figure 5 shows the fragment metamodel which defined the concepts for composition interfaces of model fragments. To be usable with arbitrary modelling languages, we refer to two metaclasses from the Ecore metamodel. EObject stands for an arbitrary model element and EStructuralFeature identifies a reference between model elements.

To both model element and reference, so-called variability types can be assigned. An AddressablePoint represents such a variability type. Furthermore, we distinguish ReferencePoints (places in a model that can be accessed during composition) and VariationPoints (places in a model that can be modified during composition). These are further categorized into Prototype, Hook, Anchor, Slot, ValuePrototype and ValueHook.
Pairs of reference and variation points can be combined in a composition as follows:

- A model element that is typed as **Prototype** can be bound to a **Hook**. This means that the prototype element and its children are copied and put into the place were the hook is.

- A model element that is typed as **Anchor** can be bound to a **Slot**. This means that references pointing at the slot element will point at the anchor element after composition. In contrast to hook/prototype bindings, no elements are copied and added to a model.

- Attributes (of model elements) can be typed as **ValuePrototype** or **ValueHook**. Binding a value prototype to a value hook means that the attribute value is copied. One can also replace only parts of an attribute value that is typed as value hook (**beginIndex** and **endIndex** properties).

For each combination of variation and reference points also the types of the model elements (i.e. metaclasses) have to match. That is, they have to be equal or one has to be a subclass of the other. The metaclass is determined via the **eClass** reference from **EObject** to **EClass** of the Ecore metamodel (not shown). This ensures that a composed model (i.e., a model that is the result of an invasive composition) is correct wrt. its metamodel.

**AddressablePoints** are grouped in **Ports**. We distinguish between **HeterogeneousPorts** and **HomogeneousPorts**. **HeterogeneousPorts** allow the grouping of model elements with different types. **HomogeneousPorts** allow further grouping of similar **HeterogeneousPorts**.

A set of ports then makes up a **CompositionInterface** of a **Fragment**. The grouping into ports allows us to address groups of model elements that are scattered over a model fragment. This allows for complex and cross-cutting weaving operations.

### 4.4.2. Composition Program Metamodel and Composition Language

The composition program metamodel shown in Figure 6 defines the concepts for defining compositions of model fragments in so-called composition programs. Together with a graphical syntax defined in GMF, it defines a composition language that can be utilised for composing model fragments defined in arbitrary modelling languages.

The metamodel consists of four basic composition language concepts **FragmentInstance**, **PortInstance**, **CompositionLink** and **Setting**. Additionally, **CompositionProgram** is introduced as container for **FragmentInstances** and **CompositionLinks**.

A **FragmentInstance** represents a model fragment in a composition. In contrast to a Fragment (cf. Figure 5), a **FragmentInstance** appears in the context of a composition program, which essentially means that it is linked to other **FragmentInstances**, different instances of the same fragment can have different links in different or even in the same composition programs (i.e., different contexts). Different fragment instances in one composition program are distinguished through unique names inside the program (**name** attribute of **FragmentInstance**).

The connection between a fragment and its instance is established via the **ID** attribute of **FragmentInstance** and its **fFragment()** operation. We use the prefix **f** for all operations and references that lead from a composition program concept (e.g., **FragmentInstance**) to a fragment concept (e.g., **Fragment**). **fFragment()** asks the repository for the fragment with the ID specified in the **ID** attribute. If the fragment exists, the current fragment instance is valid. Otherwise an error is raised which can be reported to the developer.

The remaining attributes of **FragmentInstance** are **target**, **targetID** and **reference**. Setting a fragment instance as **target** means that the composed copy of the fragment that is obtained after execution of the composition is stored into the repository. The **targetID** is then used as the ID for this newly composed fragment. Setting a fragment instances to **reference** enforces that its fragment will not be copied during composition and that only cross-references to it will be established.
Figure 6 – Composition Program Metamodel

If a fragment instance is valid, PortInstances can automatically be created using the update() operation. This operation essentially creates one port instance per Port (cf. Figure 5) that is available on the Fragment returned by fFragment(). The port instance is linked to the port via its name.

To define compositions of model fragments we introduce the concept CompositionLinks. A composition link links two PortInstances (source and target). Once a link is created and connected, the match() operation is performed. It relies on the type and variability type matching of variation and reference points. match() determines if the link is valid.

The last composition concept is Setting. A Setting holds a property/value pair and can be used to set a value hook directly rather than linking it to a value prototype. Consequently, a port instance may have multiple settings. The property of the value identifies the value hook to be set by its name. The value specifies the value to be set. Thus a setting can be executed as attribute composition.

Generic Composition Language

Based on the metamodel, we defined a generic graphical composition language for model fragments. The graphical syntax was specified as a GMF model [10] on the basis of which an editor was implemented as part of Reuseware. We enriched this implementation with possibilities for syntax customisation.

Figure 7 shows all variants of the graphical syntax elements of our composition language. Fragment instances are depicted as boxes with round corners. If the box has a grey background, the fragment instance is set to target. If the box has a dashed outline, the fragment instance is set to reference. If the fragment instance is invalid (i.e., if the fragment that is identified by the ID attribute does not exist), the box has a grey outline and is marked with a warning symbol.
Figure 7 – Generic graphical syntax of the generic composition language

Port instances are visualised as circles attached to the fragment instance boxes. If a port instance is receiving (i.e., contains hooks), the circle has a solid border but is not filled. If a port instance is contributing (i.e., contains prototypes), the circle is filled. If a port instance is configuring (i.e., contains neither hooks nor prototypes), the circle has a dashed outline. Invalid ports instances (i.e., the port that is identified by the name attribute does not exist) have a grey dashed outline. Invalid port instances are only shown if they are still linked otherwise they are removed automatically.

A composition link is depicted by a solid black line. If a link connects a prototype with a hook, the composition direction is visualised with an arrow. If a link is invalid, it is marked with a warning symbol. A link is invalid if the connected port instances are invalid or if the types of the model elements behind the two connected ports do not match.

Settings can be modified in a tabular properties view, which is shown individually for each fragment instance when it is selected in the editor (shown in the bottom of Figure 7). Here, the settings that exist are listed grouped by the port instances they belong to.

Figure 8 – A composition program with customized syntax
Customized Composition Language

Although the generic composition language is not (domain) specific for one composition system, the syntax—boxes for components and lines to connect them—is very intuitive for many composition systems. The user experience can be improved by customising the graphical syntax with dedicated icons for certain fragments or types of fragments.

This customisation does not change the core of the language—that is, the metamodel (cf. Figure 6) does not change, but only the graphical syntax. This can be done dynamically with the proper tool support. We extended the editor in Reuseware to allow the assignment of dedicated icons to fragments. Figure 8 shows an example of a composition program with customized icons.

4.4.3. Composition System Metamodel and Language

The composition system specification (CSYS) language of Reuseware is used by composition system developers to define the basic concepts of a composition system as a base for the component model and the composition language. The metamodel for CSYS is shown in Figure 9. Since it contains only specification concepts, no operations for semantic interpretations are provided in this metamodel. It has semantic impact on the operations found in the fragment (cf. Figure 5) and composition program (cf. Figure 6) metamodels. The CSYS metamodel forms the base of a specification language for composition systems. These specifications are the base for reuse extensions—that introduce new composition concepts into existing languages—discussed in the next section.

A CompositionSystem consists of a set of FragmentRoles and CompositionAssociations, which can be Configurations or Contributions. Furthermore, a FragmentRoles contains a set of PortTypes, which can be StaticPortTypes or DynamicPortTypes. A CompositionAssociation points at two PortTypes and their FragmentRoles as ends. Each FragmentRole, CompositionAssociation and PortType has a unique name within the CompositionSystem. These metaclasses are explained in the following.

FragmentRoles: We observed that many composition systems have different types of components. For example, a module system has module components and program components, an aspect system has advice components and core components. We can also think of more domain-specific component types as we discovered in the Télefonica case study (cf. Section 6.1). Such a component type is a role a model plays when it is used as a component (i.e., as model fragment). Thus, the concept component type is called FragmentRole in Reuseware.

PortTypes: To become more specific about how fragment roles relate, we can make some statements about the ports a fragment's interface would provide if the fragment plays a certain fragment role. Although ports are created dynamically depending on the addressable points available in a fragment, for certain ports we know that they always exist (and if they do not something is wrong with the fragment) and for others we know at least that they—or that multiple of their kind—may exist. Consequently, each fragment role has PortTypes, which specify this.

CompositionAssociations: Given the knowledge about the port types of fragment roles, we can also add knowledge about allowed composition links. That is, we can say if links between ports that are of a certain port type are allowed and if they should be contributing (i.e., bind prototypes with hooks) or not. Following the UML terminology, a type of a link is an association. Thus, we call this concept, which types composition links, CompositionAssociation.
Composition System Specification Language

For the metamodel we defined a textual syntax with EMFText. Listing 1 shows a schematic instance of the metamodel. In Line 1, an ID is given under which the specification is stored in the repository. Lines 2–5 show the definition of a fragment role with the name fragmentRoleName. It consists of static (Line 3) and dynamic (Line 4) port type definitions. A contributing association is defined in Lines 6–8 and a configuring association in Lines 9–11. These definitions state which two port types defined above can be connected.

```
1: compositionsystem ID {
2:   fragment role fragmentRoleName {
3:     static port portName;*
4:     dynamic port portName;*
5:   }*
6:   contributing association associationName {
7:     fragmentRoleName.portName --> fragmentRoleName.portName
8:   }*
9:   configuring association associationName {
10:    fragmentRoleName.portName --> fragmentRoleName.portName
11:   }*
12: }
```

Listing 1 – Schematic Composition System Specification

4.4.4. Reuse Extension Metamodel and Language

Figure 10 and Figure 11 show the ReuseExtension metamodel which, together with an EMFText-based syntax, defines the reuse extension language of Reuseware. A reuse extension (REX) specification binds a composition system to a concrete modelling language (e.g. UML) by defining how reuse concepts can be found in that language. The reuse extension language thereby consists of two parts. The first part (cf. Figure 10) is used for component model specification and the second for composition language specification (cf. Figure 11).

Specifying a component model means that we define which elements in a model (that is an instance of the metamodel for which the component model is specified) have which variability types and are grouped into which ports on the composition interface (cf. Section 4.4.1). The metaclasses
for component model specification are therefore closely aligned with the variability types of the Fragment metamodel (cf. Figure 5).

Specifying a composition language means that composition semantics are given to an existing modelling language (e.g., UML). This language can then be used to define composition programs, instead of using the generic graphical language shown in Section 4.4.2. Defining a new composition language hides more internal details of Reuseware to the composition system user and allows for additional abstraction (we did this for the Thales case study; Section 6.3). Reusing the generic composition language of Section 4.4.2 on the other hand saves the composition system developer work and allows reuse of the graphical composition editor (this was done in the Telefónica and SAP case studies). However, in the Telefónica case we also experimented with individual composition languages.

In general, the reuse extension language has to allow us to specify the conditions under which a model element has a certain variability type (or composition semantics). For this, we required an expression language that allows to define such rules on the basis of Ecore metamodels. Naturally, OCL is a candidate for this. It allows us to express conditions and queries on Ecore models and the EMF MDT implementation of the OCL [20] can be directly used in an EMF based tool. Therefore, we decided to base the reuse extension specification language on OCL, which was also a reason to use textual syntax for this language (since OCL itself has textual syntax).

The similar upper parts of Figure 10 and Figure 11 show the part of REX that is common to component model and composition language specification. Each ReuseExtension as a reference (csCompositionSystem) pointing to the CompositionSystem (cf. Figure 9) it binds. Through this, a REX specification is linked to a CSYS specification. The metaclasses FragmentRoleBinding, PortTypeBinding and CompositionAssociationBinding relate parts of a REX specification to FragmentRoles PortTypes and CompositionAssociations respectively. Furthermore, the metaclasses EPackage, EClass and EStructuralFeature are imported from Ecore. Through them the metamodels, metaclasses and features of the modelling language for which the REX specification is written are identified.

Component Model Specification

For component model specification, the following metaclasses are used (Figure 10). The metaclasses FragmentRole2FragmentBinding and PortType2PortBinding (with its subclasses PortType2HeterogeneousPortBinding and PortType2HomogeneousPort-Binding) are used to assign sets of AddressablePointDerivationRules to port types. An AddressablePointDerivationRules specifies if (and if, which kind) of addressable point is created for which model element when a composition interface for a model fragment is constructed (cf. Figure 5). To determine if a rule can be applied to a certain model element, theRuleContext has to be known. The context is specified through the EClass that a model element has to instantiate (eBound-Class reference) and, in the case the rule should be applied to a reference, through an EStructuralFeature of the EClass (eBoundFeature) in addition. The context can be narrowed down by the isExpression that has to be fulfilled by a node. The context of a ReuseExtension concerns the root element(s) of a model fragment and determines whether the specification is applicable for that model fragment at all. This basically defines the language that is extended by the specification.
To specify variability types for model elements AddressablePointDerivationRules are utilised. For each subtype of AddressablePoint a corresponding derivation rule type exists (cf. Figure 5). The context specifies to which elements a rule shall be applied. If a certain element has a variability type, additional information to identify the element on the composition interface has to be derived. The portNameExpression, mergePortNameExpression and pointNameExpression are used for this. Respectively, they derive the name of the heterogeneous port, the homogeneous port and the addressable point itself. Consequently, all these expression have to return a value that is or can be converted into a String.

In addition, value prototypes need to extract information as String. Thus, a valueExpression has to be given for a ValuePrototypeDerivationRule, which is expected to return a String or a value that can be converted into a String. For value hooks we might require the definition of a range inside an attribute that shall be manipulated. This can be specified using the beginIndexExpression and endIndexExpression that are expected to return positive integer values that identify a position in the attribute that is to be manipulated. These expressions are not required if the complete attribute should be modified.

The textual syntax for component model specification is shown on a schematic instance in Listing 2. Lines 1–4 specify a ComponentModelSpecification instance and its properties. In Line 1, an ID is assigned to the component model under which it is placed in the repository. Next, in Line 2, the namespace of the model fragments to which this component model may apply is specified—multiple apply statements are allowed. In order to refer to EClasses in the rest of the specification, a set of EPackages is identified in Line 3 by the EPackages’ nsURIs. The EClasses contained in the first EPackage might be addressed by their name only in the following. A class from any of the EPackages can be addressed using the EPackages’ nsPrefixes in the :: prefix nota-

Figure 10 – Reuse Extension Metamodel (Component Model Specification Part)
tion (e.g., \texttt{uml::Model}). Finally, Line 4 defines the EClass (i.e., the context) a root node has to instantiate to make the REX applicable to the corresponding model fragment—optionally followed by a condition formulated in OCL that the root node has to fulfill.

Following this, the fragment roles (Line 6) and port types (Line 7 and 19) are identified for which addressable point derivation rules should be defined. Lines 4–13 show a blueprint specification of one of the \texttt{AddressablePointDerivationRules}. Which rule type is defined concretely is determined by the \texttt{variabilityType} which is either \texttt{hook}, \texttt{prototype}, \texttt{slot}, \texttt{anchor}, \texttt{value hook} or \texttt{value prototype}. In Line 5, \texttt{eClass} is the name of one \texttt{EClass} defined in one of the \texttt{EPackages} named above. This \texttt{EClass} has to be instantiated by a model element to apply the rule to it. Optionally, a condition can be given in OCL that the element has to fulfill in addition. If an element is of correct type and fulfills the given condition, the specified expressions can be evaluated to derive the details about the variability type of the current model element.

```plaintext
1: componentmodel ID
2: implements csysID
3: apply namespace.** +
4: packages <package> +
5: rootclass rootelementEClass if $OCLExpression$? {
6:   fragment role fragmentRoleName if $OCLExpression$? {
7:     port portName {
8:       eClass.feature? is variabilityType if $OCLExpression$? {
9:         port merge expr = $OCLExpression$ ?
10:        port expr = $OCLExpression$ ?
11:       point expr = $OCLExpression$ ?
12:       value expr = $OCLExpression$ ?
13:     }
14:   }
15:   homo port portName {
16:   }
17: } +
18: } +
19: homo port portName {
20: }
21: }
22: } +
23: }
```

Listing 2 – Schematic Component Model Specification

Composition Language Specification

The lower part of Figure 11 shows the composition language specification part of REX. The upper part is similar as in Figure 10, since composition language specifications are bound to CSYS specifications in a similar manner as component model specifications. The concrete subtypes of these binding metaclasses however differ, to bind fragment roles and port types to composition language concepts rather than component model concepts.

These subclasses are \texttt{CompositionLanguageSpecification}, \texttt{FragmentRole2FragmentInstanceBinding}, \texttt{PortType2SettingBinding} and \texttt{CompositionAssociation2CompositionLinkBinding}. These metaclasses are derivation rules themselves. A \texttt{CompositionLanguageSpecification} has the additional \texttt{idExpression} that is used to compute the ID of the composition program that is modified when the specification is applied. Several applications might modify the same composition program by computing the same ID. Each \texttt{CompositionLanguageSpecification} consists of a set of \texttt{FragmentRole2FragmentInstanceBindings} and \texttt{CompositionAssociation2CompositionLinkBindings}. The former ensures that a fragment instance with the name computed by the \texttt{nameExpression}, the \texttt{ufi} computed by the \texttt{ufiExpression} and the target \texttt{ufi} computed by the \texttt{targetUfiExpression} exists in the corresponding composition program. The latter ensures that the port identified by the names computed through the \texttt{fragmentInstance1NameExpression} and the \texttt{portInstance1NameExpression} is linked to the port identified by the \texttt{fragmentInstance2NameExpression} and the \texttt{portInstance2NameExpression}.
In addition, a \texttt{FragmentRole2FragmentInstanceBindings} can contain \texttt{PortType2SettingBindings} to derived settings. This is done through \texttt{propertyExpression} and \texttt{valueExpression} pairs that derive concrete property/value pairs for the port with the corresponding type when executed in the context of the current \texttt{FragmentRole2FragmentInstanceBinding}.

![Diagram of the Reuse Extension Metamodel](image)

**Figure 11 – Reuse Extension Metamodel (Composition LanguageSpecification Part)**

We explain the syntax for composition language specification on a schematic instance shown in Listing 3. The header in Lines 1–4 is similar as in the component model specification syntax (cf. Listing 2). In Line 5 we see the definition of the unique composition program ID (\texttt{ucpi}) that identifies a composition program. The expression has to return a String that can be used as ID. For this we introduced an ID metaclass with a set of ID computing operations (e.g., \texttt{appendSegment()}, \texttt{removeSegment}(), etc.) that are helpful when computing IDs in OCL expressions.

Lines 6–15 show the binding of a fragment role to a fragment instance. The \texttt{fragmentRoleName} identifies the role in the implemented composition system. The \texttt{reference} keyword determines if the fragment instance should be a reference (i.e., not copied but only referenced). The \texttt{eClass} specifies the \texttt{EClass} for which instances the rule applies. The optional \texttt{if} expression can enforce further constrains on those instances. The \texttt{fragment}, \texttt{ufi}, and \texttt{target ufi} expressions (Lines 8–10) compute the \texttt{name}, \texttt{ufi} and \texttt{target ufi} of the fragment instance respectively. The \texttt{target ufi} is optional and only if it is set, the target flag of the fragment instance is set to \texttt{true}. The \texttt{fragment expression} has to return a string and both \texttt{ufi} expressions an ID. Settings of a port with the name \texttt{portName} are derived by using a pair of OCL expressions (Line 12).

D3.1.b.V6 Model Weaving
Association bindings are used to derive composition links as shown in Lines 17–25. Again, eClass and its optional if condition is used to identify the elements that trigger the derivation. Then, four rules, which are all expected to return a string, are used to identify the two fragments and the ports on these fragments that should be linked.

```plaintext
Listing 3 – Schematic Composition Language Specification
```

### 4.4.5. Extension Point for Model Transformation Engines

The creation of an instance of the Fragment metamodel (cf. Figure 5) or Composition Program metamodel (cf. Figure 6), which is performed by interpreting component model or composition language specifications (cf. Section 4.4.4), can be regarded as a model transformation. The source metamodel is then the metamodel of the language in which model fragments are defined. The target metamodel is either the Fragment or the Composition Program metamodel.

If a developer is familiar with a certain model transformation language, it might be easier for him to define a component model or a composition language using this transformation language instead of the reuse extension language of Reuseware. However, in these cases a deeper knowledge of the Fragment or Composition Program metamodel is required as when using the reuse extension language. To support the usage of arbitrary model transformation languages for this purpose we provided two extension points in Reuseware.

- `org.reuseware.coconut.fragment.fragmentUpdateOperation`
- `org.reuseware.coconut.fragment.derivedCompositionProgramUpdateOperation`

Implementing a simple Java interface, developers of composition engines can connect their engine with Reuseware. YORK developed such a connector plugin for the Epsilon engine developed in MODELPLEX [14]. We experimented with this in the Télefonica case study (cf. Section 6.1.3).

### 5. Reuseware User Guide

This section describes how Reuseware is installed (Section 5.1) and how its Eclipse-based UI tooling is used (Sections 5.2–5.5).
5.1. Installation
Reuseware can be obtained as part of the MODELPLEX platform or can be independently installed in Eclipse Galileo or IBM’s Rational Software Architect.

5.1.1. MODELPLEX Platform
Reuseware (and all its optional components) is part of the MODELPLEX Platform that can be downloaded and used instantly (on Windows systems):


5.1.2. Eclipse Galileo
Since the Reuseware Composition Framework consists of a set of Eclipse plug-ins it can be installed in any Eclipse Platform installation\(^1\) of version 3.5 (Galileo) or higher. The framework can be installed within the Eclipse platform using the update manager (Help > Install New Software…).

To install the framework use the following URL in the Work with field:

- http://reuseware.org/update

All components are found in the Reuseware Composition Framework category (cf. Figure 12) and can be individually installed. Press Next > and follow the installation instructions. Missing dependencies will be automatically installed from the Galileo update site.

![Figure 12 – Installing Reuseware through the update manager in Eclipse Galileo](image)

5.1.3. Rational Software Architect
Reuseware can also be installed in IBM’s Eclipse-based Rational Software Architect (RSA)\(^2\). For this use the update manager (Help > Install New Software…).

---

\(^1\) http://www.eclipse.org/downloads

D3.1.b.V6 Model Weaving
• Add the Reuseware update site: http://reuseware.org/update

• Ensure that only the Reuseware update site (see above) and the Eclipse 3.4 update-site (http://download.eclipse.org/eclipse/updates/3.4) are activated (deactivate others via Manage Sites...)

• Install only the following components:
  o Reuseware Composition Framework
    ▪ Reuseware Core (CoCoNut)
    ▪ Reuseware Feature: GMF Diagram Composition
    ▪ Reuseware Feature: Round-trip Support Composition
    ▪ Reuseware Feature: GMF Diagram Composition

5.2. Getting Started

This section gives an overview of the UI tooling of the Reuseware core (cf. Section 4.3.1). The user interface of Reuseware is integrated into the Eclipse platform and works together with modelling editors installed in the current Eclipse platform. The Reuseware UI core provides the following two basic features: Fragment Stores and the Fragment Repository View.

Fragment Stores are folders in the Eclipse workspace that contain model fragments. One can mark any folder in any kind of project as a fragment store, by selecting the folder and pressing the Activate Fragment Store button in the toolbar shown in Figure 13.

![Figure 13 – Fragment Store activation button in Eclipse toolbar](http://www-01.ibm.com/software/awdtools/architect/swarchitect/)

Fragments that are registered in a store are available for reuse in composition programs. Each fragment has a Unique Fragment Identifier (UFI). The UFI is determined by the position of the fragment in the store.

Fragment stores can also contain composition programs (*.fc files), composition system definitions (*.csys files) and reuse extensions (*.rex files). How such files are created is described in Sections 5.3 and 5.4.

The Fragment Repository View (cf. Figure 14) lets you inspect which fragments are available in your system. Open the view in Eclipse through Window > Show Views > Other... > Reuseware > Fragment Repository. From the view you can 1) directly open a fragment by double-clicking it 2) select fragments you want to reuse in a composition program by pressing the + button.

Another way to inspect and search the repository is the fragment browser component that can be installed in addition and is described in Section 5.5.

---

2 http://www-01.ibm.com/software/awdtools/architect/swarchitect/
5.3. Developing Composition Systems

New composition system definition (cf. Section 4.4.3) or new reuse extensions (cf. Section 4.4.4) can be created using the New File Wizard of Eclipse. Selecting the corresponding specification type in the Reuseware category as shown in Figure 15 does this. The file needs to be created in a folder that belongs to a Fragment Store.

The new file is opened in a specific editor (generated with EMFText) that supports features like highlighting and code completion as shown in Figure 16. The syntax for the specification languages is described in Sections 4.4.3 and 4.4.4. As soon as the file is saved, the fragments and composition programs in the repository that are influenced by the specification are updated.
5.4. Defining Model Fragments and Composition Programs

The composition system user can utilise the following UI tooling of the Reuseware core for creating model fragments and composition programs.

Model Fragments

Model fragments can be defined using any Eclipse model editor for creating EMF models. This can for instance be UML models created with TOPCASED [16], Telefonica or Thales DSL models created in the corresponding GMF model editors, textual models created with EMFText-based editors or also Java code that is loaded as model with JaMoPP (cf. Section 4.3.5).

To make a model available as model fragment it has to be placed in a folder inside a fragment store and the file extension of the model file has to be registered with Reuseware. This is done via Preferences... > Fragment File Extensions as shown in Figure 17. If the model comes with a separate diagram file (e.g., in the Telefonica DSL, the model with the extension *.cim has a corresponding diagram with the extension *.cim_diagram), a mapping from model file extension to diagram file extension needs to be registered as seen in Figure 17. Only then, the diagram composition component (cf. Section 4.3.2) can work correctly for these models and diagrams.
Composition Programs

A composition program can also be created via the Eclipse New File Wizard (cf. Figure 15). Customized layouts can be activated using the template drop-down box on the wizard page shown in Figure 18. A composition program also has to be placed into a folder in a Fragment Store. A new composition program will open in the graphical editor introduced in Section 4.4.2.

The syntax of the elements of Composition Programs is explained in Section 4.4.2. A fragment instance is created by selecting a fragment in the Fragment Repository View (cf. Section 5.2) and pressing the + button when the composition program editor is opened. The fragment instance is displayed as a box with circles attached. One can open fragments directly from a composition program by double-clicking a fragment instance.

At least one of the fragment instances in a composition program has to be set as target. This means that the composition will extend this fragment with others. In the properties of a target fragment, one can define the UFI of the result (targetUFI). Target fragments are displayed in grey.

Composition links are created via the palette. Use composition links to connect Ports (the circles at the fragment instances) to describe compositions. Only ports of matching type can be linked. If the composition link has a warning symbol, the linking is not possible. Some links turn into arrows describing a direction of composition. This depends on the type of ports you are linking. The direction indicates that one fragment is extended with another.
5.5. Classifying Fragments and Browsing the Repository

If the faceted classification and browsing component (cf. Section 4.3.3) is installed, the following UI tooling is available for fragment classification and repository browsing.

Fragment Classification

Figure 19 gives an overview of the fragment classification tooling. As one example of a model fragment, Figure 19 (b) shows a model that is modelled in the Téléfonica DSL. This DSL is based on the CIM standard [21] and allows the modelling of network devices and networks in a great level of detail. By putting the model into a Reuseware Fragment Store, it becomes a model fragment.

The model fragment is then classified as follows (cf. Figure 19): As mentioned in section 4.3.3 predefined facets are needed. In this example, the project ‘library_facets’ (a) contains facets to classify CIM model fragments. After marking a folder as Fragment Store (a), the fragment shown in the Téléfonica DSL editor (b) can be classified using the Fragment Description View (c). This view allows to edit free-text attributes (d) and to select facets for faceted classification (e). After selecting a facet an additional view is presented (cf. Figure 20) showing the values grouped in this facet. Here, one has to select the values that describe the fragment best. This value will then be stored and form one part of the faceted classification. Furthermore, buttons on the right (f) allow to edit or delete selected facet-value-pairs or to clear the whole list. After a number of fragments were classified, a separate Eclipse Perspective can be used to browse the repository (g).
Fragment Browsing

The Fragment Library provides a Fragment Browser that allows faceted browsing on the base of facetized classifications (cf. Figure 21). Using facet widgets on the left and right, one can select a facet value (a) to filter upon this criterion. The Search Content View (b) presents fragments that meet this criterion and selecting new values makes this list shrink (called zoom-in step). One can hit the Reset button (c) to deselect a value and perform a zoom-out step. Facet-value-pairs not presented in widgets are listed in the Selected Facets View (d) and the ✗ button (e) can be used to delete this criterion and perform the corresponding zoom-out. Furthermore, the restart button (f) allows starting the browsing from scratch and the reuse button (g) gives the opportunity to reuse a fragment selected in the list. However, this button cannot be used in this perspective and is therefore described in the following.
Figure 21 – Search process using the Fragment Browser

Fragment Reuse

Fragments that are the result of a search can directly be reused in a Reuseware composition program (cf. Section 5.4). As the browser’s Search Content view (a) can be opened in different perspectives it can be presented while editing a composition program (b). To reuse a fragment, simply select it in the Search Content view and hit the reuse button (c). The fragment will then appear in the composition program (d) and is ready to be connected with other fragments in the program.
Figure 22 – Reusing model fragments in a Reuseware Composition Program
6. Applications in MODELPLEX Case Studies

In this section we summarise the results of applying Reuseware in the MODELPLEX case studies. We have utilised Reuseware in three case studies in MODELPLEX: Telefonica, SAP and Thales. The utilisation in the Telefonica case study (Section 6.1), started in the middle of the project. It is the most intensive usage of Reuseware in MODELPLEX and exploits the Reuseware core (cf. Section 4.3.1), diagram composition component (cf. Section 4.3.2) and repository component (cf. Section 4.3.3). The SAP case study related work (Section 6.2) was performed and continuously updated in the first phase of the project and helped significantly in initially developing the concepts behind Reuseware. The work on the Thales case (Section 6.3) was performed in the last year of the project and showed that the concepts of the Reuseware core are usable in another scenarios. Additionally, we experimented with the composition language specification feature (cf. Section 4.4.3) and the round-trip component (cf. Section 4.3.4), which are the latest additions that were not applied in the other case studies before.

6.1. Telefonica Case Study

The part of the Telefonica case study that concerns Reuseware is based on the Common Information Model (CIM) standard [21]. Based on this standard, the Telefonica DSL—also called CIM DSL—was developed by Telefonica and Xactium in MODELPLEX [3].

With Reuseware, we developed a composition system that allows for the reuse of CIM model fragments with different at different levels of abstraction. The abstraction level of a CIM fragment hereby depends on the degree of complexity that is hidden behind the composition interface of a CIM fragment. That is, the more model elements are hidden (i.e., the composition system user is not aware) the higher the abstraction. Normal CIM models that expose all details are on Level 0.

6.1.1. Scenario

The CIM Level 0 models are modelled using the CIM DSL editor developed by Telefonica and Xactium [3]. Such models describe network configurations in all details. The TID case study proposes (Requirement 224 in [19]) to abstract the modelling to higher levels (Level 1, Level 2, etc.) with more abstract concepts. On Level 0, for instance, an EthernetPort, is a single model element, while on Level 1, a Router (containing several EthernetPorts) is a single model element.

Experimenting with Level 0 models and Reuseware, we discovered that we could express concepts of higher abstraction levels by Level 0 models. That is, a collection of Level 0 model elements can represent one Level 1 element. For example, a Router (on Level 1) can be described by a Level 0 model fragment containing several EthernetPorts and other elements.

Consider the example in Figure 23, modelled in the Level 0 DSL. The three boxes indicate that the model can be decomposed into three fragments: wan.cim, router.cim and lan.cim. These model fragments represent Level 1 concepts.

Using the mentioned fragments and an appropriate composition system, we can define a composition program that constructs the original Level 0 model (cf. Figure 23). All the composition system specifications can be found online.³

The composition program is shown in Figure 24. It combines all three fragments (wan.cim, router.cim, lan.cim) into an empty core Level 0 model (empty.cim) using contribution composition links⁴. Additionally, configuration composition links⁵ define relations between fragments. Furthermore, settings (cf. Section 4.4.2) are used to set attributes of model elements directly in the composition program (not shown in the figure). Here, some attributes of certain elements inside the fragments are set from the outside.

³ http://reuseware.org/index.php/CIM_DSL_Extension
⁴ contribution composition link (cf. Section 4.4.2): copies elements
⁵ configuration composition link (cf. Section 4.4.2): changes references between elements

D3.1.b.V6 Model Weaving
Figure 23 – A CIM Level 0 model decomposed into fragments

Figure 24 – A composition program composing CIM Level 0 model fragments
The composition program can be seen as a Level 1 model: The fragments (except the empty.cim) represent single Level 1 elements. The composition links represent relations between them. The value definitions represent attribute settings of the elements. In comparison to a model expressed in a conventional modelling language, there is no specific syntax (i.e., different visualizations – shapes, icons, etc. – for different elements) and the empty core and the links to it are shown as well as some unused ports.

However, the following was desired:

1. Define different syntax for different fragments in a composition program.
2. Hide uninteresting fragments, ports and composition links.
3. Automate the definition of hidden fragments and composition links.

These requirements were realized in the syntax customization feature (cf. Section 4.4.3) that was then used in the Téléfonica case study.

6.1.2. Composition Language Customization

To implement the above requirements, we defined extension points in Reuseware that allows us to plug-in syntax customisations with the following features:

1. Composition Program Designs enable specification of custom figures for fragments, ports and composition links. For each figure, a condition over the fragment's interface is given (e.g., does port X exist) to define for which fragments the figure should be used (cf. issue 1 above). It is also possible to define that an element is hidden (cf. issue 2 above).

2. Composition Program Templates define fragments that will be automatically created in a composition program. A template can be selected when a composition program is created with the Creation Wizard (cf. Figure 18). When additional fragments are added to such composition programs, Reuseware automatically connect ports of the newly added with matching ports of the hidden fragments (cf. issue 3 above).

We defined a customization for CIM abstractions. Basically, we defined different figures and icons for the different fragments through a Composition Program Design extension. Additionally, we defined a template that adds, but hides, the empty core fragment to each composition program based on the template.

Using the customisation, the composition program from Figure 24 looks as shown in Figure 25. There are several advantages in defining CIM abstractions with composition programs over defining them with a separate DSL for each abstraction level:

- Reuseware’s tooling, in particular the composition program editor and fragment repository, can be directly reused to define and manage the model fragments. No new tools need to be developed from scratch.
- The concepts of other abstraction level) can intuitively be defined using concepts of Level 0. This makes it easier and more flexible for domain experts (in this case Téléfonica) to define new abstractions themselves.
- The semantics of new abstraction levels are given through the composition. Reuseware can compose abstract models to one monolithic Level 0 models without additional effort. Through this, all tooling (e.g., model transformations and code generators) developed for the CIM Level 0 DSL in MODELPLEX can be applied for all CIM models, regardless of their abstraction level.
6.1.3. Developing additional CIM DSLs

The original idea in the Télefonica DSL development was to develop a set of DSLs—one for each abstraction level. Using Reuseware, we were able to tackle the abstraction issue with the use of composition systems and programs instead of completely new DSLs as described above. Still, the concern was expressed that the customisation of the composition language might not give enough flexibility for the design of additional abstraction levels.

The main blocking factor for starting the development of additional DSLs was the cost of developing graphical DSLs itself, which was experienced to be higher than expected in the case of the Level 0 DSL [3]. Thus, YORK and TUD explored how Epsilon [14] and Reuseware could be combined to ease the development of new graphical DSLs that are abstractions of an existing DSL.

YORK extended Epsilon with the EuGeNia component [25] that allows a quick specification of graphical syntax with significantly less effort than doing it with GMF alone (which was done for the Level 0 DSL in [3]). EuGeNia uses metamodel annotations for this. Together, we connected Epsilon and Reuseware (cf. Section 4.4.5). Then, we developed a metamodel annotation approach to define composition languages using Epsilon. This approach allows the full specification of new DSLs with graphical syntax and composition language semantics only by annotating the metamodels. The approach was published in [23].

6.1.4. Browsing for CIM Fragments

To evaluate the usefulness of the fragment library feature of Reuseware in the Télefonica case study, we performed an evaluation in collaboration with Télefonica. This evaluation included the following three steps. First, we defined domain specific CIM facets to allow the classification of CIM-fragments. Second, we used the facets to classify a set of CIM fragments specified by Télefonica. Third, we collected feedback from Télefonica with the following results.

Domain Facet Definition

To identify domain-specific facets to classify CIM model fragments, we performed a domain analysis and found a set of six facets (cf. Figure 26). In contrast to general software component facets, they allow a classification specific for the telecommunication domain. A telecommunication expert can then use our integrated tooling to classify CIM models using the domain facets (cf. Section 5.5). Thus, he works in the terminology of his domain and does not need any knowledge about, for example, source code components or their classification. Other telecommunication experts can then use these domain-specific facets with our faceted browser to browse a library of CIM model fragments (cf. Section 5.5).
Téléfonica CIM experts recognized facets as a useful approach for classifying fragments in general. However, they see a potential weakness of the approach in the fact that facet developers bear a huge responsibility. First, they restrict facets and facet values that are the base for all later classifications and browsings. Second, they need to clarify the meaning of facets and values and should take the fragment developers’ perspective into account. These aspects underline that it is crucial for both, classification and browsing to have well defined facets available.

**Fragment Classification**

Faceted classification with its restrictive character appears to be an adequate method for structuring a fragment library for the experts at Téléfonica. If the facets are well defined, they make sure even a big number of fragment developers will not create anomalies such as synonyms, antonyms or plural forms. In addition to that, providing domain knowledge as facets and values simplifies work especially in a huge domain such as telecommunication, since fragment developers and users do not have to remember all these concepts on their own.

An automated classification appears to be a very useful approach for practical use. Rather than classifying huge sets of fragments by hand, the domain experts, in the roles of composition system users, want to use as much automation as possible. Therefore, rules must be specified that cover important aspects of the domain. In cooperation with Téléfonica we identified opportunities in the telecommunication area to specify such rules for automation.

One particular application of automation rules was identified as an interesting alternative to using the specific classification tool. In the case of CIM fragments, adding notes to a graphical fragment diagram was a common method used by the CIM experts. These notes contained information that could be extracted and translated into facet values. Téléfonica’s domain experts saw this as a useful feature, since it gives them the opportunity to define facet values directly in their models. All in all, the domain experts saw the automated classification support as a critical feature for broad industrial acceptance of faceted classification.

**Fragment Browsing**

Faceted browsing was received by the CIM experts as an intuitive and user-friendly method to search in a huge library of fragments. They acknowledged that step-wise searching in a faceted browser supports fragment users that think in the problem space rather than in the solution space. As transferring ideas between both worlds is a major challenge in finding the right fragment for reuse, presenting facets and values does help. The domain experts, who used the composition environment for CIM without the faceted library beforehand, stressed the importance of integrating the library system into the composition environment. For them it was very important that a discovered

<table>
<thead>
<tr>
<th>Facet</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIM-Schema</td>
<td>Uses CIM specific terms to classify the fragment.</td>
<td>CIM-Core, CIM-User, CIM-Interop</td>
</tr>
<tr>
<td>Connection</td>
<td>Names the main connection used by the fragment.</td>
<td>Ethernet, Wifi, Bluetooth</td>
</tr>
<tr>
<td>Device</td>
<td>Describes which sort of device is used by the fragment.</td>
<td>Hub, Router, Modem</td>
</tr>
<tr>
<td>Element Type</td>
<td>Distinguishes between conceptual and real life fragments.</td>
<td>Logical, Physical</td>
</tr>
<tr>
<td>Protocol</td>
<td>Names the main protocol used by the fragment.</td>
<td>IP, DHCP, IPX, SSH, Telnet</td>
</tr>
<tr>
<td>Structure</td>
<td>Gives a hint about the fragment’s inner structure.</td>
<td>SingleConcept, MultiConcept</td>
</tr>
</tbody>
</table>

**Figure 26 – CIM specific facets**

Téléfonica CIM experts recognized facets as a useful approach for classifying fragments in general. However, they see a potential weakness of the approach in the fact that facet developers bear a huge responsibility. First, they restrict facets and facet values that are the base for all later classifications and browsings. Second, they need to clarify the meaning of facets and values and should take the fragment developers’ perspective into account. These aspects underline that it is crucial for both, classification and browsing to have well defined facets available.

**Fragment Classification**

Faceted classification with its restrictive character appears to be an adequate method for structuring a fragment library for the experts at Téléfonica. If the facets are well defined, they make sure even a big number of fragment developers will not create anomalies such as synonyms, antonyms or plural forms. In addition to that, providing domain knowledge as facets and values simplifies work especially in a huge domain such as telecommunication, since fragment developers and users do not have to remember all these concepts on their own.

An automated classification appears to be a very useful approach for practical use. Rather than classifying huge sets of fragments by hand, the domain experts, in the roles of composition system users, want to use as much automation as possible. Therefore, rules must be specified that cover important aspects of the domain. In cooperation with Téléfonica we identified opportunities in the telecommunication area to specify such rules for automation.

One particular application of automation rules was identified as an interesting alternative to using the specific classification tool. In the case of CIM fragments, adding notes to a graphical fragment diagram was a common method used by the CIM experts. These notes contained information that could be extracted and translated into facet values. Téléfonica’s domain experts saw this as a useful feature, since it gives them the opportunity to define facet values directly in their models. All in all, the domain experts saw the automated classification support as a critical feature for broad industrial acceptance of faceted classification.

**Fragment Browsing**

Faceted browsing was received by the CIM experts as an intuitive and user-friendly method to search in a huge library of fragments. They acknowledged that step-wise searching in a faceted browser supports fragment users that think in the problem space rather than in the solution space. As transferring ideas between both worlds is a major challenge in finding the right fragment for reuse, presenting facets and values does help. The domain experts, who used the composition environment for CIM without the faceted library beforehand, stressed the importance of integrating the library system into the composition environment. For them it was very important that a discovered
fragment was directly reusable from the search result view of the fragment browser (cf. Section 5.5).

Nevertheless, the experts missed some features while testing the browser during the case study. The browser always constructs a query using logical and concatenation of all selected facet values. The domain experts encountered cases, where there was a need to express that a facet value should not be set or where or concatenations would be desirable. They suggested that the browser could be improved in the way that the fragment user selects a facet value that should not be met by the desired fragments or other configuration facilities, in order to influence the construction of the actual queries based on the selected facet values. Ultimately, the domain experts also suggested that for complex searches a SQL-like query language over the facet data would be helpful for experienced users. Nevertheless, the faceted browsing process has proven to be intuitive as it supports the user in various ways.

The overall results of the evaluation are positive. In particular, it confirms the following points:

- Faceted classification and browsing are promising methods to structure and explore libraries of domain-specific fragments.
- Automatic rule-based classification is important for usability and acceptance of the library system.
- The integration of the library system with fragment development and composition environment is of high importance.

6.1.5. Conclusion

This section gave an overview of the utilisation of Reuseware in the Télefonica case study. It showed how Reuseware is used to reduce complexity in a large DSL at different levels of abstraction. The results of the collaboration with Télefonica from the perspective of Task 3.1 are very positive. The works allowed us to evaluate Reuseware with a work with a real complex language (CIM). It drove the stabilisation of the tools and the development of additional components such as the diagram composition and the fragment browser. The publication [23] at MODELS, one of the leading modelling conferences, is also a result of the collaboration.
6.2. SAP Case Study

In this section shows an application of Reuseware ob business process models from the SAP case study. The models are defined as UML activity diagrams. In SAP, similar process modelling languages such as BPMN [22] are also applied for business process modelling. Because of the language independent nature of Reuseware, the composition concept developed for UML activities can be transferred to theses languages easily.

6.2.1. Scenario

Although UML activity diagrams can be modularised into partitions in single models, reusing and combining parts of activities modelled separately is not well-supported by UML itself. For the SAP case, we would like to define general processes with activity diagrams and keep them extensible with specific activities for concrete application use-cases. Such extensions are used in SAP to develop so called SAP composite applications.

As an example, we look at the order processing activity modelled in Figure 27. The process contains a checking activity (the `CustomerDataCheck` action together with the decision node below) that determines whether certain data (here customer data) is consistent. We want to keep the order processing activity extensible such that additional checks can be inserted in parallel to the customer data check.

To perform the extension, a developer should not need to know anything about the ordering process, but that check activities can be inserted. What this developer needs to know is that a check activity has to have one incoming control flow (from the `checkFork` node) and two outgoing flows (to the `checkMerge` and `checkJoin` nodes). With this knowledge, the developer can design additional check activities—for instance, the one from Figure 28 that determines the customer’s credit card liquidity.
Such extensibility can be realised by thinking about models as components. Treating the ordering process model (cf. Figure 27) as a model fragment, almost the whole activity should be encapsulated. Only the checkFork, checkMerge, and checkJoin nodes (grey boxes), to which the incoming and outgoing flows of additional checks can connect, should be reflected in the composition interface. Looking at the credit card check (cf. Figure 28) as a model component, we can again hide the internal activity. We only think of the initial (InitialNodeCREDIT) and final nodes (FINISH and CHANCEL) as open spots in the model, which need to be manipulated through the composition interface.

By defining a reuse extension for UML activity diagrams, as done in the next section, we provided the extensibility and variability described above.

Listing 4 – Composition system specification for business process composition

Listing 5 – Reuse extension for UML activity diagrams
6.2.2. A Composition System for UML Activity Diagrams

In the case of business process extension we wanted to give the composition system user a high amount of flexibility. Thus, we developed a small UML profile that can be applied to activity diagrams by composition system users to mark variation or reference points directly. Stereotype applications do then correspond to the declarations of addressable points as defined in the reuse extension below.

The profile is tailored for the activity diagram scenario and simplistic: it defines two stereotypes Anchor and Slot that both define the tagged values portName, groupName, and pointName from and can be applied to activity nodes.

Listing 4 shows the composition system specification and Listing 5 the reuse extension in the corresponding editors. Therefore, in Listing 5, the interpretation of the profile as component model is specified. In Line 4, it is defined that this is a specification for the UML languages. In Lines 8 and 9, we define that each activity offers two implicit extension points (hookd): one for the list of nodes (Activity.node) and one for the list of edges (Activity.edge) of an Activity. We group them together into the static port ContentsHook (Lines 8–11). In Lines 22 and 23, all elements contained in an activity (i.e., its nodes and edges) are identified as prototypes and associated with the static port Contents. At last, Lines 13–16 and Lines 26–29 define that nodes with a Slot (or Anchor) stereotype are treated as slots (or anchors respectively). The properties of the addressable point are derived from the tagged values of the stereotype application (OCL expressions in Lines 14, 15, 27 and 28).

We are now ready to prepare the order processing model from Figure 27 and the credit card check model from Figure 28 for composition. The order processing model now implicitly offers a receiving port ContentsHook. Additionally, the checkFork, checkJoin, and checkMerge nodes should be addressable to connect additional check activities to them. We do that by applying stereotypes to these nodes and setting the tagged values portName and pointName.

The credit card check model now implicitly exports its content—that is its two actions and all control flows—to the contributing port Contents. To connect the edges correctly to the nodes of the order processing model later, we apply the slot stereotype on the initial and final nodes of the credit card check model. We add them to the CreditCardCheck group and give them the same point names.

We now load the fragments into the composition program editor that displays their composition interface as shown in Figure 29. In the composition program, we link the contributing port Contents with the receiving port ContentsHook and the two ports CreditCardCheck and CheckActivities. When we execute the composition program, we obtain a composed model as shown in Figure 30.

Figure 29 – Business process composition program
6.2.3. Conclusion

In this section we showed how business processes, as modelled in the SAP case study, are extended using a composition system developed with Reuseware. The compactness of the composition system specifications (cf. Listing 4 and Listing 5) indicates how quickly composition systems can be developed and modified. Therefore, Reuseware is explicitly applicable for quick prototyping when exploring new composition paradigms for different kinds of (modelling) languages.

Furthermore, the only language depended part of the specification is the reuse extension (cf. Listing 5). The same composition system could be bound to another business process language with similar concepts as UML activity diagrams (e.g. BPMN). To transfer the same composition system to another language, it is sufficient to exchange the UML specifics (metaclasses and OCL expressions) with the corresponding concepts in the other language’s metamodel. This is a good foundation for further case studies in this area.
6.3. Thales Case Study

In this section a description of the application of Reuseware within the Thales case study can be found. Most importantly, the new features that were implemented in Reuseware to execute the case study, namely the Round-trip extension (cf. Section 4.3.4) is described in detail. We used this feature for viewpoint synchronization addressing parts of Requirement 179 [2]. Concretely, Requirement 179 states [2, p. 29]:

“...several models will be built for a system, at different abstraction levels (e.g., logical vs. physical, PIM vs. PSM). The abstraction/decomposition support requested here addresses the support of the analysis/design or navigation process within one system model.”

Furthermore, Requirement 179 requests [2, p. 29]:

- Need to manage impact on inter-model relationships
- Need for tool mechanisms for editing, navigating, change mgt., etc.

The verification that Reuseware meets these requirements was done using the Thales case study.

6.3.1. Brief Case Study Description

In the Thales case study, a system model consists of different models, representing different viewpoints, defined in different modelling languages. Thus, the requirement manifests in the goal to integrate different viewpoints on a single system and allow editing of each viewpoint separately without compromising the consistency of the overall system model. More precisely, four viewpoints are present in the Thales case study (i.e., the capability analysis model, the capability configuration models, the system services model and the security model). The first three viewpoints are specified in UML, which ensures some consistency based on the fact that all views use the same language. However, the last viewpoint is expressed in a DSL, namely the Thales Security DSL [4]. We used the round-trip feature to ensure consistency between the different viewpoints modelled in different languages. The overall use case is schematically depicted again in Figure 31.

6.3.2. Instantiating the Reuseware framework for the Case Study

To apply Reuseware to the Thales case study the framework needed to be instantiated. For example, the inter-model relation between the UML models and the DSL models needed to be formalised. In addition, several basic artefacts were created to allow the composition of UML models and Security DSL models. An overview of the complete round-trip scenario including the additional artefacts, addressed in the Thales case study, is shown in Figure 32.
Figure 32 – Reuseware applied to the Thales case study

Starting from the UML system models (shown on the left) two composition programs are derived. The first one uses Security DSL core model fragments to compose a Security DSL core model that represents the system core, which is modelled in UML, in the Security DSL (upper part). The second composition program enriches the Security DSL core model with security information (lower part). The transformations that perform these two derivation steps as well as the Security DSL core model fragments were manually created to instantiate Reuseware for the Thales scenario. That is, the transformations and Security DSL core model fragments are deployed such that Reuseware can work with them. This procedure needs to be taken only once.

The purpose of the core model fragments and the derived composition programs is to be able to compose the parts of the security models, which can be determined from the UML models and the parts, which are not present there. The latter are stored separately in a model called Security RIO\(^6\) model. The derived composition program weaves the core model fragments to assemble the structure of the system, which can be extracted from the UML models and the additional security information from the RIO models to obtain an integrated security model (shown in the right part of Figure 32).

It must be noted here, that the Thales case study employs two compositions. The first one (upper left in Figure 32) composes basic building blocks of the security DSL (the core model fragments) according to the system models defined in UML (UML acts as composition language here). The second one (lower right in Figure 32) composes the core security model (i.e., the structure of the system relevant to security analysis defined in UML) and the security RIO model (i.e., additional information needed for security analysis which is not present in the UML models).

\(^6\)Risk, Impact, Objective

D3.1.b.V6 Model Weaving
It is important to realize here that the first composition technically does not compose UML models and security models, because both are expressed in different languages, namely the UML and the Security DSL. What is actually composed by Reuseware is a translated version of the UML models and the core security fragments. Invasive Software Composition as the formal grounding of Reuseware allows composition of arbitrary fragments, but they must be expressed in the same language. Note, that this is not a restriction of the Reuseware tool, but rather a conceptual limitation that applies to all model composition tools. Model integration across multiple languages requires language integration beforehand. For the Thales case study, where the UML and the Security DSL needed to be handled, both languages were defined independently. Thus, even though there is a relation between the two, no formal definition of this relation (e.g., a transformation) is available.

The second composition is homogeneous with regard to the languages of the involved fragments. The implications of the difference between the two types of compositions will be discussed shortly, when the limitations of Reuseware’s Round-trip feature are discussed.

### 6.3.3. Composition Example

After instantiating Reuseware for the Thales case study (i.e., after writing the transformations and compositions) users can derive security models from the UML models and enrich them with additional information. An extract from an example UML models is shown in Figure 33.

The additional security information is stored in separate models that conform to the Security DSL metamodel. Again, an example is shown in Figure 34.
An example of an integrated security model, which is the result of the model composition that is performed by Reuseware is shown in Figure 35.

Here, some elements stem from the UML models (e.g., the exchanged data elements and the security elements, for instance the AODBInstance element shown in Figure 33). Other elements (e.g., specific data about risks, see risk instances, risk impact and risk opportunity in Figure 35) are composed by Reuseware. Thus, at this stage it was possible to compose the data from the UML models and the information found in the Security to models to obtain an integrated model. However, the final goal of the case study was not reached yet as the requirements state that the composed model (or view) needs to be changed by the modellers. To support this demand, Reuseware was extended with a new feature called Round-trip support. The conceptual work on this feature was done earlier and can be found in [24].

### 6.3.4. Round-trip Support

As one of the goals of the Thales case study was to support editing of viewpoints, Reuseware was extended with support for round-trip engineering. The overall approach is depicted in Figure 36. More details are published in [24].
During model composition (i.e., while weaving models) additional trace information is captured by the Reuseware engine. This information, which may also be called trace information, associates model elements in the source fragments (i.e., either elements in the core security fragments or in the ROI security model) with their respective copies in target fragments (i.e., the integrated security model). This association between elements allows the editing of composed models by propagating changes to the respective source models. Each change that is made needs to be translated to a respective change in a source fragment. In general this translation is not always fully automatic as pointed out in [24]. However, in the Thales case study the nature of the composition (i.e., the concrete composition programs used) allow for fully automatic Round-trip.

Consequently, in the Thales case study, the integrated security model can therefore be changed and each change is automatically propagated to the RIO model. When the UML models are changed, the integrated view (i.e., the integrated security model) is recomposed preserving the contents of the RIO model. Thus, all viewpoints of a system (i.e., the UML and the security models) can be edited without compromising consistency.

6.3.5. Conclusion

With the development of the Round-trip extension for Reuseware it was possible to both underpin the theoretical work of [24] and to implements viewpoint synchronisation as requested in the Thales Use Case. We have shown that Reuseware can compose model fragments specified in the Thales Security DSL. Furthermore, the resulting integrated models can be edited and changes are propagated back to the original fragment. As found in [24], this Round-trip functionality does need manual decision making in general. But, the Thales Case Study showed, that the specification of default behaviour can be sufficient for some application. In this particular case, no manual decisions are needed at all. This is quite a strong point as it reveals that the weakness of the Round-trip technology, namely the need for user decisions, can be easily overcome in some real-world applications.

7. Conclusion

This Deliverable summarised the work performed on model weaving in WP3, Task 3.1, Activity 3.1.b of the MODELPLEX project. As main result of this activity, the Reuseware Composition Framework and the composition and weaving concepts implemented in it, were described. Reuseware was used in three of the four case studies defined by MODELPLEX partners in WP1. There it forms the foundation to addresses different requirement (e.g., abstraction, reuse, extension and synchronisation) concerning different modelling languages (e.g., UML, different process modelling languages and different DSLs). This shows the genericity and portability of the composition approach and its implementation in Reuseware. Furthermore, the numerous publications that were accepted at international conferences, workshops and journals confirm the scientific importance of the work.

In the future, development of the Reuseware Composition Framework will be continued in the Software Technology group at TUD. The tool is already published on the website www.reuseware.org, where it will be updated and maintained. It is planned to perform more case studies in future research and PhD projects at TUD. The results of the work performed in this Activity, which could only be presented in a certain level of detail in this report and the cited publications, will be published in all details as part of a PhD thesis at TUD this summer.
References


D3.1.b.V6 Model Weaving