Aspect-Oriented Behavior Modeling In Practice

Matthias Schöttle
matthias [at] mattsch.com

MASTER OF SCIENCE

September 2012

School of Computer Science
Software Engineering Laboratory
McGill University
Montréal, Canada

and

Department of Computer Science
Karlsruhe University of Applied Sciences
Karlsruhe, Germany

Supervisors:

Prof. Jörg Kienzle, McGill University
Prof. Peter A. Henning, Karlsruhe University of Applied Sciences

Matthias Schöttle: *Aspect-Oriented Behavior Modeling In Practice,* Master's thesis in Computer Science, © September 2012

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Karlsruhe, Germany, September 2012

Matthias Schöttle

To my wife Stéphanie and my parents,
for their wonderful love and support.

Abstract

Aspect-Oriented Programming (AOP) addresses the separation of cross-cutting concerns from the business logic on the source code level. Aspect-Oriented Modeling (AOM) allows to do this on a higher level of abstraction where cross-cutting concerns are addressed during earlier phases of the software development process.

Reusable Aspect Models (RAM) is an aspect-oriented multi-view modeling approach that allows detailed design of a software system. Notations similar to UML class, sequence and state diagrams are used to describe the structure and behavior of a reusable aspect. Previously a meta-model and a weaver was defined for the structural view (class diagram) to be used in a tool for RAM. The definition of message views (similar to Sequence Diagrams) and weaving of message views has been only done in theory so far.

In this thesis we present the transformation of message views defined in theory into practice to be usable in the RAM tool. The message views and their features are evaluated for feasibility and adjustments are made where necessary to obtain a well-defined meta-model at the end. The weaving of message views is formalized and a general weaving process defined that entails all views. The weaver for message views was then implemented. Finally, the multitouch-enabled tool TouchRAM is extended with support for visualization and weaving of message views. Furthermore, ideas are presented on how to offer streamlined editing of message views and how the overall architecture of TouchRAM can be improved to increase the code quality and maintainability.

Kurzfassung

Aspekt-orientierte Programmierung (AOP) ermöglicht die Trennung von sogenannten querschnittlichen Belangen (*cross-cutting concerns*) und der Geschäftslogik auf Quelltext-Ebene. Bei der aspekt-orientierten Modellierung (AOM) wird diese Methodik bereits auf einer abstrakteren Ebene angewendet, indem querschnittliche Belange in früheren Phasen der Softwareentwicklung berücksichtigt werden.

Reusable Aspect Models (RAM) ist ein aspekt-orientierter Ansatz, welcher detailliertes Design eines Softwaresystems ermöglicht. Die Struktur und das Verhalten eines wiederverwendbaren Aspekts werden mit Hilfe von Diagrammnotationen beschrieben, welche auf den UML Klassen- und Sequenzdiagrammen sowie Zustandsautomaten basieren. Bisher wurde ein Meta-Modell und ein Weaver für die Struktur von Aspekten definiert. Die Definition von *message views* (basierend auf Sequenzdiagrammen) und das Weben (*weaving*) dieser in ein Gesamtsystem ist bisher lediglich theoretisch erfolgt.

Diese Arbeit stellt die Transformation von *message views* aus der Theorie in die Praxis vor. Die theoretische Definition wird dabei evaluiert und bei Bedarf Anpassungen vorgenommen, um ein wohldefiniertes Meta-Modell zu erhalten. Das Weben der *message views* wird formalisiert und ein Gesamtprozess für das Weben definiert, welcher das Weben der Struktur beinhaltet. Darauf folgend wird der Weaver implementiert. Anschließend wird die Multitouch-Anwendung TouchRAM erweitert, um *message views* visualisieren und weben zu können. Des Weiteren werden Ideen zum effizienten Editieren von *message views* vorgestellt sowie dargelegt, wie die Architektur der Anwendung verbessert werden kann, um die Qualität und Wartbarkeit zu erhöhen.

Acknowledgements

I am grateful to Prof. Kienzle for allowing me to come to McGill in Montréal and all his time and effort while supervising my thesis there. I am also grateful to Prof. Henning at my home university in Karlsruhe for his immediate and invaluable support for my thesis. Both of you made it possible for me to conduct my thesis abroad. Thank you very much.

I am thankful to my friend Harald for reading my thesis and helping me to spot mistakes and improve the understandability. I would also like to thank Engin, Omar, Valentin and Wisam from the lab at McGill University for the great cooperation and all the interesting discussions we had. It was great working with you, thanks.

Contents

1 Introduction **1**

2 Background **4**
2.1 Aspect-Orientation 4
2.1.1 Aspect-Oriented Programming 5
2.1.2 Aspect-Oriented Modeling 5
2.2 Reusable Aspect Models 6
2.2.1 Example 7

3 Meta-Model for Message Views **14**
3.1 Overview of UML Sequence Diagram Meta-Model 15
3.2 Features of Message Views 17
3.3 Current Meta-Model of RAM 20
3.3.1 Overview 20
3.3.2 Structural View 21
3.4 Definition of Message View Meta-Model 23
3.4.1 General structure 23
3.4.2 Interaction 25
3.4.3 Local properties 25
3.4.4 Lifeline 27
3.4.5 Message 29
3.4.6 InteractionFragments 30
3.4.7 The Complete Message View Meta-Model 31
3.4.8 Visualizing Message Views 31
3.5 Open Features 33
3.5.1 Message View Features 33
3.5.2 RAM Features 34
3.6 Facilitating future meta-model changes 37

4 Message View Weaving **39**
4.1 Weaving Structural Views 39
4.2 Message View Weaving Requirements 41
4.3 Formalizing Weaving 44

4.3.1 Copying Message Views 46
4.3.2 Weaving 49
4.4 Related Work 54

5 Message View Support For TouchRAM 57
5.1 Background 57
5.1.1 Eclipse Modeling Framework 57
5.1.2 Kermeta 58
5.1.3 Multitouch for Java (MT4j) 58
5.2 Overview of TouchRAM 59
5.3 Meta-Model in EMF 59
5.4 Integration of Message View Weaver 61
5.4.1 Implementation Details 62
5.5 Displaying Message Views in TouchRAM 65
5.5.1 Using adapters in TouchRAM 67
5.6 Streamlined Message View Editing 70
5.7 Related Work 71

6 Conclusions And Future Work 73

Bibliography 75

Chapter 1

Introduction

When designing software systems, the most common software development methodologies are conventional object-oriented techniques. Software systems are grouped into modules according to their functionality. Each module encapsulates a certain functionality of the system. Requirements for a system that affect every module (e.g., error handling, logging, authentication, transactions) cannot be assigned to a specific module. These requirements are called *cross-cutting concerns* as they cross-cut the whole system. Their code is scattered across every module, which makes it more difficult to maintain the software system, reusability is reduced and no clear separation from the business logic is possible. Cross-cutting concerns are often referred to as *aspects*.

Aspect-Oriented Software Development (AOSD) is an emerging methodology that aims at providing methods to separate cross-cutting concerns. The goal is to increase maintainability and reusability. Therefore, cross-cutting concerns can be defined at a central place and specified where in the software system they will be applied. When the final application is created, the concerns will be woven into the application at the appropriate places. While Aspect-Oriented Programming (AOP) provides this technique at the source code level, Aspect-Oriented Modeling (AOM) aims at defining aspects at a higher level of abstraction, which allows a consistent use throughout the software development process. AOM techniques often make use of the Unified Modeling Language (UML). To fully exploit what AOM approaches offer, it can be used in the context of Model-Driven Engineering (MDE), where models are the primary artifacts. Using the features of both methodologies, it allows to retrieve an executable model at the end by using model transformations and generators.

With Reusable Aspect Models (RAM) such an aspect-oriented approach has been developed in the past several years at the Software Engineering Laboratory (SEL) of McGill University in Montreal, Canada. RAM is a multi-view modeling approach which describes the structure and behavior

of aspects using diagram notations similar to class, sequence and state diagrams of UML. Their respective names in RAM are *structural view*, *message view* and *state view*. One of the main goals of RAM is to provide a high level of reusability. Aspects define a certain concern of a software system and are defined as general as possible. Furthermore, aspects can reuse the functionality of other aspects. Aspects can then be composed together by weaving them to a final application at the end.

In the past, a meta-model for the structural view of aspects was defined. Furthermore, a weaver was implemented, which weaves the structure of certain aspects together to a final woven model. A graphical tool (TouchRAM) has been in development, allowing the creation and editing of structural views of RAM aspects as well as offering the modeler to weave aspects and see the woven result.

The definition and weaving of message views has been done in theory so far. Message views were created and defined using a graphical drawing tool. The goal of this thesis was to define a meta-model for message views that integrates with the already existing RAM meta-model and to verify the theoretical realization of message views for feasibility and to make adjustments where necessary. Furthermore, the weaving had to be developed and the visualization of message views integrated into TouchRAM.

This paper presents the main contributions of this thesis. These are as follows:

- **message view meta-model**: A meta-model for message views was defined and integrated into the existing meta-model.
  - Various improvements were suggested and made.
  - For issues that haven't been resolved yet, proposals on how they could be resolved have been made.
- **weaving of message views**: The theoretical algorithm of message view weaving was formalized.
  - Weaving of structural and message views was combined.
  - The weaver was prepared to be extended for state view support.
- **message view support for TouchRAM**: TouchRAM was extended to support message views.
  - The weaver was implemented.
  - A visualization of message views was implemented in the tool.
  - Message views can be woven in TouchRAM and the woven result viewed.
  - Suggestions on how the current architecture of TouchRAM can be improved and the code quality be increased were made.
  - Ideas on how TouchRAM can offer the designer fast, easy and intuitive editing capabilities for streamlined editing of message

views were presented.

The paper is structured as follows. Chapter 2 presents background information on aspect-orientation and a detailed overview of Reusable Aspect Models. Chapter 3 describes what the meta-model for message views had to support, the decision on the structure of the meta-model and what changes were proposed for message views and to the existing meta-model. Chapter 4 provides an overview on how weaving of message views works, explains the formalized weaving algorithm and shows some related work. Chapter 5 presents the implementation of the message view weaver, the implementation of the visualization of message views in TouchRAM, provides ideas on how the architecture of TouchRAM can be improved and code quality increased and presents ideas on how streamlined manipulation of message views can be integrated. The last chapter concludes this paper and presents some thoughts on future work.

Chapter 2

Background

2.1 Aspect-Orientation

Aspect-Oriented Software Development (AOSD) [5] is an emerging software development methodology which addresses the identification, specification and separate expression of *cross-cutting concerns* in the software development life cycle. In traditional software development paradigms, that for example use *Object-Orientation*, the system is decomposed into units of primary functionality which are often called *modules*. Functionality required across several or all modules, for example, logging, security, caching, authentication etc., cross-cuts the whole software system. These are referred to as *cross-cutting concerns*. When implementing the units of functionality the code of the concerns is scattered through several modules meaning that their code is spread over multiple modules. This makes it harder to maintain the system as changes to a concern have to be applied to all modules. Furthermore, it results in code tangling, meaning that the code of cross-cutting concerns gets intermixed with the business logic. This reduces the possibilities of reusing modules. Even the use of frameworks, e.g., for handling logging, is not satisfactory as the code is still scattered across all the modules.

AOSD aims to provide techniques and mechanisms to separate cross-cutting concerns from the business logic by allowing to define them separately and to express where in the system they have to be applied to. Concerns are often called *aspects*. When the final software system is created the concerns will be unified with the system by applying them at the specified places. This allows to increase the maintainability and reusability.

Aspect-oriented techniques can be applied at different levels of abstraction. It evolved from being introduced to the implementation level as *Aspect-Oriented Programming* and can now be applied at earlier phases in the software development process, like the requirements [18] or the design phase (see Section 2.2).

2.1.1 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) [10] offers the separation of cross-cutting concerns from the business logic on the source code level. It builds on existing programming paradigms like Procedural Programming or Object-Oriented Programming. Various aspect-oriented programming languages have been developed where AspectJ [9] is the most widely-known language offering aspects for Java. Most languages share common concepts on how aspects can be defined.

A *join point model* defines various well-defined places in the execution of a program where aspects can be applied to. A *join point* can for example be the execution or call of a method. Depending on the language used this can include several more join points, like reading or writing a field, initializing an object or a class or when an exception is raised.

Developers can use *pointcuts* to quantify a join point, i.e., a pointcut describes a set of join points. Additionally it is possible to express in what circumstance exactly a certain join point should be matched. For example, when implementing the *Observer* pattern a developer could specify a pointcut that should match all calls to methods that start with *set* on classes that implement the interface *Observable*.

The structure or behavior that should be added at a join point is called *advice*. The most common mechanisms on where the advice is added are before, after or around the join point. For example, AspectJ also allows to define whether the advice should be applied when a method call is made (i.e., the advice is applied to the caller of the method) or when the method is executed (i.e., the advice is applied inside the method) using different pointcuts.

An aspect encapsulates those concepts and is defined separately from the business logic allowing to be applied across the whole system depending on the defined pointcuts. Weaving is performed when the application is compiled. The aspects then get woven into the application unifying it to the final application.

2.1.2 Aspect-Oriented Modeling

Looking at a software development process when only Aspect-Oriented Programming is used during the implementation phase this creates a gap between the design and earlier phases of a software system and the implementation. The developer has to build a mental bridge between the design of the system, which can be based on object-oriented methods for example, and the aspect-oriented implementation. The process of transforming a non-aspect-oriented design is complicated and error-prone which leads to a reduction in the quality of the software system. When tracing back elements from the implementation to the design of the software system the same process has to

be applied in opposite order making traceability difficult.

This lead to the emerge of Aspect-Oriented Modeling (AOM) that allows aspect-oriented techniques to be applied at higher levels of abstraction. Furthermore, approaches can be combined to form a process [17]. Schauerhuber *et. al* [21] present an in-depth survey which provides identification of similarities and differences of several AOM approaches.

Aspect-Oriented Modeling techniques allow the identification and definition of cross-cutting concerns on a high level of abstraction without having to consider the details on how they will be implemented. The notations of AOM approaches are often based on the ideas of the *Unified Modeling Language* (UML). In general, AOM approaches can be distinguished between asymmetric and symmetric approaches [21]. Asymmetric approaches make a distinction between cross-cutting concerns and the base of the application. Symmetric approaches, however, don't have this distinction and the whole system is divided into aspects.

Combined with Model-Driven Engineering (MDE) [22] the advantages of each technique can be unified and disadvantages compensated. MDE utilizes models as the primary artifact in the software development process and aims to provide and use model transformations, verifications and checks to refine or combine them to include more and more detail. The goal is having models at the end where an executable version using generators can be created from. Ultimately this could mean that it is possible to generate a 100% code.

Instead of performing the process of weaving on the code level, AOM techniques that support weaving of models allow the composition of complex models through model weavers. Model hierarchies are used to compose a final woven model. Alternatively aspect-oriented code can be generated from aspect-oriented models [16].

2.2 Reusable Aspect Models

Reusable Aspect Models (RAM) [1, 11, 13] is an Aspect-Oriented Modeling approach that offers detailed design by providing scalable multi-view modeling. It integrates extended versions of three diagram notations from the Unified Modeling Language (UML). These are class, sequence and state diagrams. Their respective names in RAM are *structural view*, *message view* and *state view*. Aspects in RAM describe the structure and behavior of concerns. It is a symmetric approach meaning that there is no distinction between aspect and base. Every concern or functionality of a software system is described by an aspect. One of the main goals of high importance is to achieve a high level of reusability. Aspects in RAM are defined as general as possible and mechanisms are provided to reuse functionality of existing aspects. This allows high-level aspects to be decomposed into lower-level aspects and other aspects can reuse their functionality as well. Besides, as-

pects can be shared across applications. Furthermore, the functionality of existing aspects can be extended by other aspects using pointcut and advice mechanisms.

RAM aspect models define an aspect interface that clearly designates the functionality provided by the aspect. Every public operation is part of the aspect interface. When an aspect in RAM wants to reuse the functionality of existing aspects this is done by instantiating that aspect. Aspects in RAM are aware of what functionality they are extended by. Therefore, they are not oblivious as described by Filman and Friedman. In [6] they state the definition of AOP as being *quantification* and *obliviousness*. However, when building large applications with AOP, changing a pointcut can lead to unwanted effects. The results become unpredictable as too many elements could be affected by a slight change in a pointcut.

Combining several aspects might lead to conflicts when several changes in the behavior or structure are required by different aspects. For this case, RAM offers *conflict resolution aspects* where the resolution of conflicts between two aspects can be separately defined.

Recently, with TouchRAM a tool has been developed that allows to create the structural view of aspects, instantiate others, weave aspects and view the woven result. For this, a meta-model for the structural view was defined and a weaver implemented. With TouchRAM a library of existing reusable aspects is provided. The *reusable model library* offers different design patterns (e.g., *Observer*, *Singleton* etc.), utility aspects (e.g., *Named*, *Map* etc.) as well as networking, transaction and workflow aspects.

2.2.1 Example

For a better understanding of RAM we will use an example aspect to show the main features of RAM aspects. Furthermore, this example will be used throughout this paper. We will explain a small part of a stock exchange application which uses the *Observer* design pattern [7]. The *StockExchange* aspect defines a stock and its window to display it. The *Observer* design pattern is used to receive notifications about modifications of a stock and update the window. This design pattern defines an observer that is able to register itself on a subject (or observable objects). Each subject can be observed by many observers. Whenever the subject gets modified it notifies all registered observers that it was updated. This is a lower-level functionality and can be reused several times in an application. The *Observer* pattern is defined as an aspect allowing it to be used by other aspects or applications as well. The aspect *Observer*, however, depends on an even lower-level aspect called *ZeroToMany*.

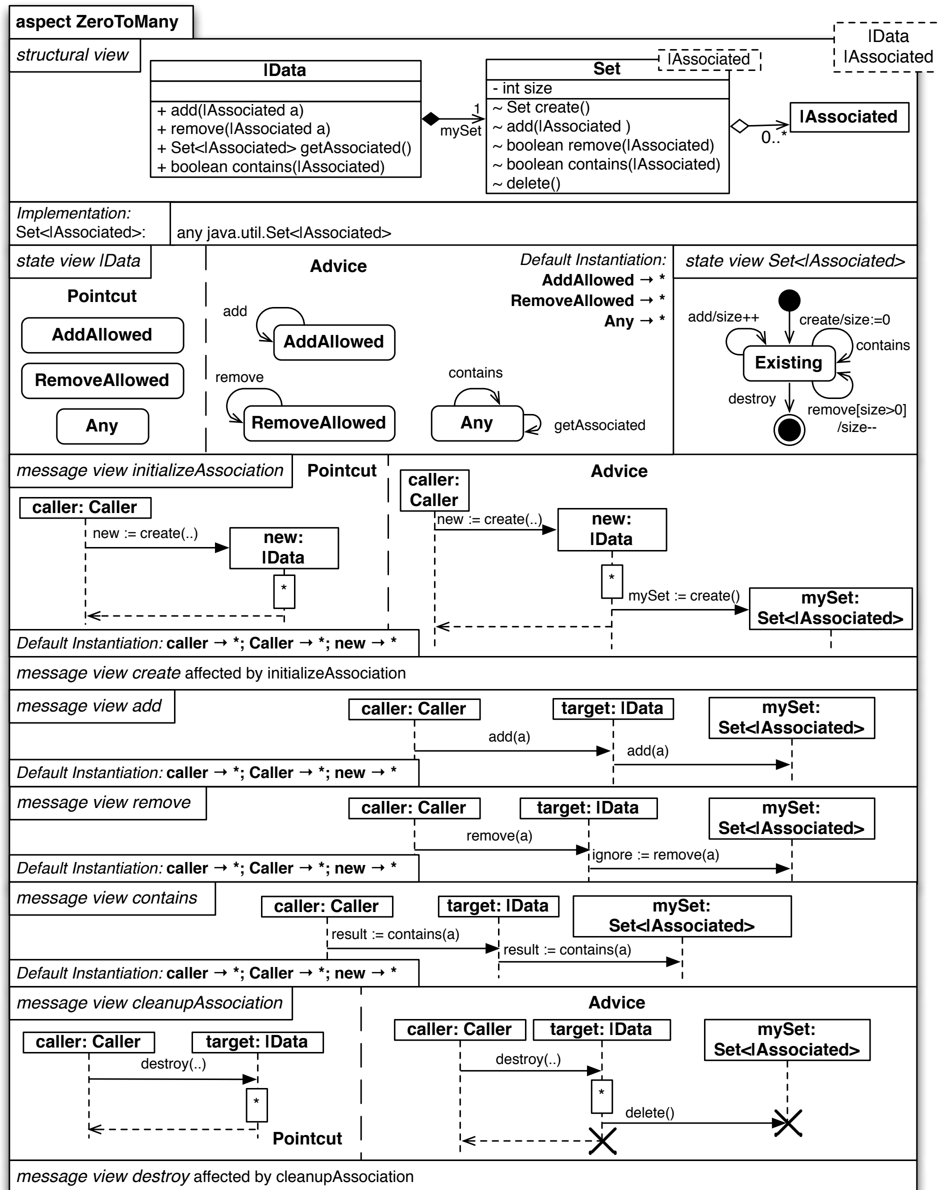

Figure 2.1: The *ZeroToMany* Aspect.

ZeroToMany

The aspect *ZeroToMany* occurs very frequently. It defines the association between two classes using composition with a multiplicity of zero to many (*0..**). The definition of this aspect is shown in Figure 2.1.

Structural View The structure of aspects is defined in a *structural view* which defines all classes and their attributes and operations. *ZeroToMany* defines a class |*Data* which uses a *Set* to link an instance of |*Data* to many instances of |*Associated*. |*Associated* is an empty class.

Classes and operations can be marked *partial* to specify that they are uncompleted. These need to be completed before being able to be used in the application. It is not possible to create instances of partial classes. Partial elements are represented by a vertical bar '|' in front of the elements name. Partial elements define the *mandatory instantiation parameters*. The *mandatory instantiation parameters* define all model elements that have to be mapped when instantiating an aspect.

Both |*Data* and |*Associated* are partial, i.e., they are incomplete as it is unclear at this point what those two classes exactly are. Any aspect that wants to instantiate this aspect has to map both classes to a class of its own structure. This allows any aspect to introduce an association between two classes with a composition. The class *Set* refers to a Java implementation of `java.util.Set`. RAM allows to reuse existing classes provided by the programming language or frameworks being used. Furthermore, |*Data* contains operations that allow the adding and removing of associated objects, to check whether a certain object is contained or the retrieval of the complete set of associated objects. These operations are all public and hence define the public interface of *ZeroToMany*. The operations of *Set* are *aspect-private* (denoted by ∼, which is package private in UML) meaning that they can only be called from classes inside this aspect. The top right corner of the structural view shows the *mandatory instantiation parameters* which in this case are |*Data* and |*Associated*.

Message Views The behavior in terms of collaboration between objects of an aspect is described using *message views*. Message views are extended sequence diagrams. Further details of message views are explained in detail in Section 3.2. For every public operation a message view has to be provided detailing the exchange of messages. Furthermore, existing behavior can be extended. In this example the constructor of |*Data* is unknown and therefore the message view is empty. However, the constructor of a class in an aspect that uses *ZeroToMany* and got mapped to |*Data* is affected by the message view *initializeAssociation*. This message view describes that whenever *create* is called (which is the *pointcut*) the behavior described in *advice* has to be executed. In this case, the behavior is added after the original behavior (represented by a box with the character '*' inside). The *Set* of |*Associated* will be created and stored in the property *mySet*. The same applies to the destructor of |*Data* where the *Set* will be deleted (see message view *cleanupAssociation* in Figure 2.1). In addition, behavior from instantiated aspects can be extended or replaced.

State View *State Views* allow the modeler to specify different states an object can be in and a protocol on what method calls it accepts in each state. State views serve as a mechanism to perform model checking or verification on whether message views comply to the defined state views. The modeled message views must conform to this protocol. For each class specified in the structural view that defines operations a state view has to be provided. One transition for each operation has to be at least contained in the state view. Due to this rule, a state diagram has to be specified for the class *Set*. State views for partial classes are different as can be seen in Figure 2.1. As it is an incomplete class it is impossible to define an initial and end state. However, it is possible to specify what states are important for this class. Therefore, for partial classes a state diagram with pointcut and advice is defined. The pointcut defines the relevant states this class needs and the advice specifies in what state calls are accepted to specific methods. Furthermore, this allows as well to extend the state views of other aspects by modifying states and transitions using pointcut and advice.

Observer

The definition of the aspect *Observer* is shown in Figure 2.2. It defines a class *|Observer* that can observe one *|Subject*. In addition to operations, that allow the start and stop of observing a subject, it defines *|update* which allows the subject to notify the observer about changes. *|Subject* defines an operation *|modify*. Both operations are partial meaning that aspects that want to use the *Observer* aspect have to specify concrete operations that modify the subject and an operation that allows the observer to update by providing mappings. The association between a *|Subject* being observed by many *|Observer*s is introduced through instantiating the *ZeroToMany* aspect.

Instantiation When instantiating an aspect, RAM currently offers the modeler two different kinds of instantiations: *depends* and *extends*. The *depends* instantiation is used for aspects providing different functionality, i.e., the modeler wants to reuse the functionality of another aspect and might want to extend it. The modeler has to provide mappings for all elements that should be exposed at the higher level. The visibility of unmapped elements is changed from *public* to *aspect-private* [1]. *Extends* can be used for aspects with similar functionality modeling the same level of abstraction, when the modeler wants to extend the functionality of an aspect. In this case the visibility of elements stays the same. Mappings are provided in the form *|Data* → *|Associated* where first the element from the lower-level aspect is given and then mapped to the element in the higher-level aspect. An element from the lower-level aspect can be mapped to many elements of the higher-level aspect.

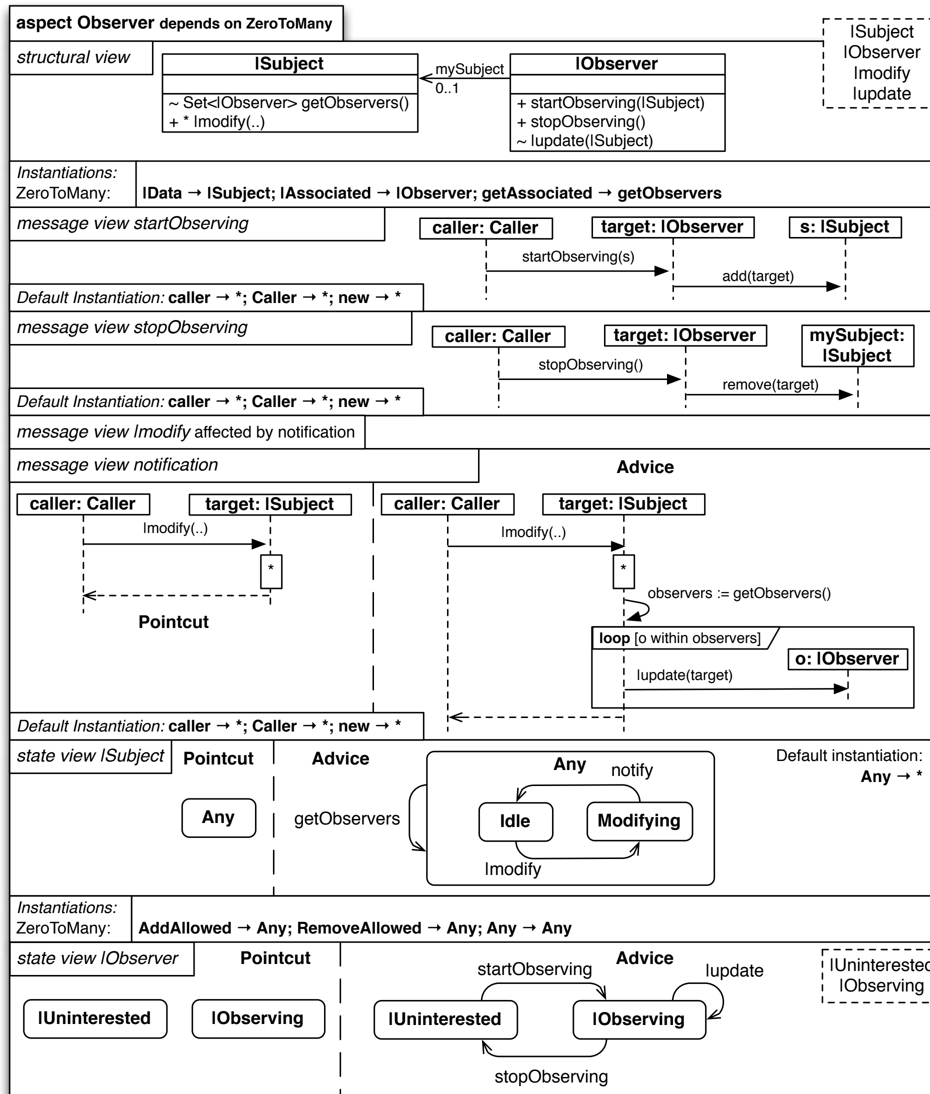

Figure 2.2: The *Observer* Aspect.

When instantiating *ZeroToMany* in *Observer*, mappings for the *mandatory instantiation parameters* *|Data* and *|Associated* have to be provided. *|Data* is mapped to *|Subject* and *|Associated* to *|Observer*. This means that *|Subject* will get a *Set* of *|Observer*s. Additionally the operation *getAssociated*, which returns the *Set*, is renamed to *getObservers* by mapping it.

In the message views it is now possible—due to the mapping—to call operations on *|Subject* that come from *|Data* of *ZeroToMany*. For example, in the message view for *startObserving* the operation *add* is called.

Furthermore, in the state views a mapping for the states of *|Data* has to

be provided. In this case, all states are mapped to the state *Any* of |*Subject*.

StockExchange

The final step in our example is to apply the *Observer* to a complete aspect, i.e., the highest level aspect in our example. The *StockExchange* aspect is depicted in Figure 2.3. It defines a *Stock* which has a name and a price and corresponding getter and setter operations. A stock windows task is to display the information of a stock. Therefore, it defines the operation *updateWindow* that can be called to request the window to update the information of the stock. In order for a *StockWindow* to be able to receive updates whenever the stock was modified, the modeler reuses *Observer* as this already existing aspect provides the required functionality.

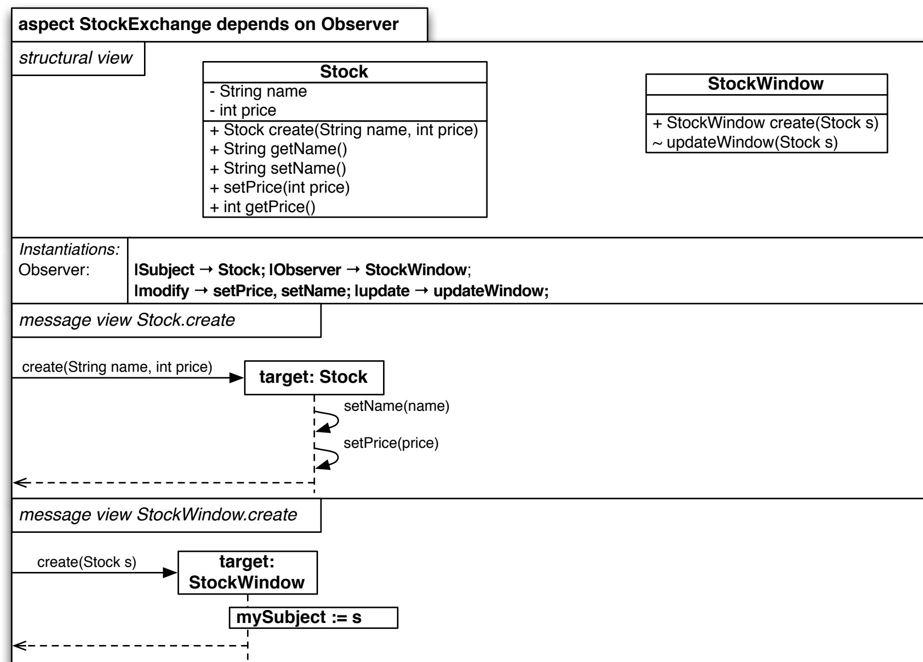

Figure 2.3: The *StockExchange* Aspect.

The modeler has to instantiate it. When instantiating, the modeler has to provide mappings for the *mandatory instantiation parameters* at least. The mappings are shown in Figure 2.3. The |*Subject* is the *Stock* and the |*Observer* is the *StockWindow*. The operations that modify stock are the setters, hence, |*modify* is mapped to *setName* and *setPrice*. For *StockWindow*, |*update* is mapped to *updateWindow* as it is the operation that should be called to request the window to update itself.

Weaving

Currently only the structural views of aspects can be woven using a tool. Weaving of the other views has only been done in theory so far. The weaver in RAM allows to weave aspect hierarchies of arbitrary depth. The designer is given the choice to either weave an aspect hierarchy completely, i.e, all aspects the highest-level aspect depends on are woven into this aspect, or to weave two specific aspects together that are directly dependent. This allows the modeler to see the result of instantiating an aspect and ensure that the result is the way it is intended to be.

For all mapped elements the weaver merges them together. Unmapped elements are copied over. Figure 2.4 shows the result of weaving *StockExchange* completely, i.e., *ZeroToMany* gets woven into *Observer* and $ZeroToMany +$ *Observer* woven into *StockExchange*. This results in an aspect that is independent.

|*Data*, |*Subject* and *Stock* got merged. Therefore, *Stock* now contains the operations and properties from |*Data* and |*Subject*. The unmapped properties were copied into *Stock*. As *getAssociated* was mapped to *getObservers* to rename it, *getObservers* retrieved the behavior of *getAssociated*. Since |*Associated* doesn't contain anything, only |*Observer* and *StockWindow* were merged. The weaving of message views was omitted for space reasons as this is explained in more detail in Chapter 4.

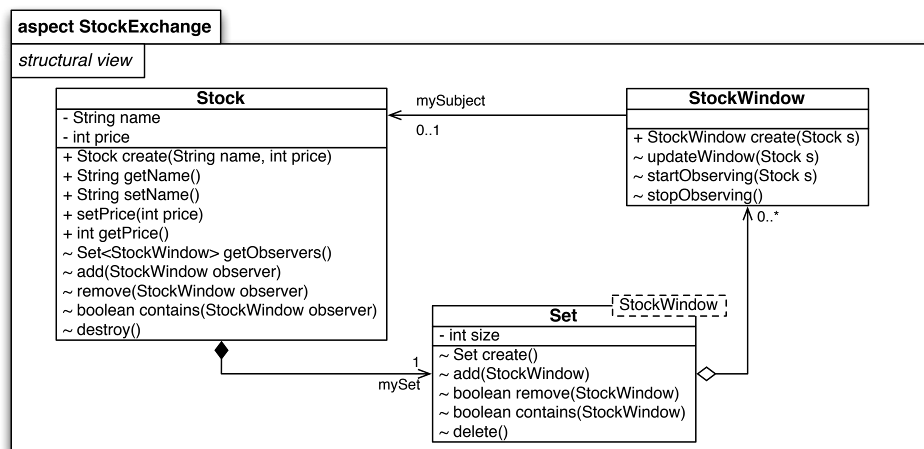

Figure 2.4: The structural view of the woven *StockExchange* aspect.

Currently only a meta-model for the structural view exists. In order to support weaving of message views a meta-model is required. We explain the definition of the message view meta-model in the following chapter.

Chapter 3

Meta-Model for Message Views

Message views are used to describe the behavior of aspects. The modeler of a RAM aspect defines how model elements inside an aspect interact with each other. The focus lies on the description of the interchange of messages between objects. Message views are based on *Sequence Diagrams* (SD) of the Unified Modeling Language (UML). SDs allow to describe message interchanges in a software system. In RAM, message views only describe interchange of messages in the form of operation calls. However, message views have a much higher level of detail. Sequence diagrams don't require to define every possible execution path in an interaction. This means that the modeler may only show the most important information that is necessary to understand the behavior of the software system or part of it. One goal of RAM is to be able to generate as much code as possible for aspects. Ultimately this would mean that 100% code of an application can be generated. However, this requires to have a much higher level of detail in message views.

One idea for the RAM tool is to offer users the ability to import their existing UML models. Furthermore, users that designed their models in UML tools before will be able to reuse them. Only missing information or restructuring would then be necessary to get them compatible with RAM. This could lead to a faster adoption by users.

Although the existing meta-model of the structural view is not exactly the same as the class diagram meta-model, it is similar as it shares the common concepts (see Section 3.3). Even though message views are mainly based on UML Sequence Diagrams this doesn't mean the same meta-model has to be used. Due to some structural differences and the higher level of detail this is not possible. However, a subset of it can be used in order to have as few differences as possible. This can help with importing and exporting existing UML models. Therefore we looked at existing UML tools and in particular what underlying meta-model they use. While often a proprietary

format is used to serialize the project which contains the models, almost all tools offer the possibility to import or export the models to a machine readable format of the UML specification serialized in XMI (XML Metadata Interchange). Some of the tools further extend the format, e.g., by adding visual information. Other tools base their models on the Eclipse UML [26] meta-model—an Eclipse project offering an EMF[1]-based implementation of the UML meta-model—or additionally offer import/export for it.

The order of occurrences of events in UML is defined by their geometrical (vertical) position. It is not sufficient to have an order that is defined in the graphical representation of the diagram and not contained in the model itself. This information is essential for weaving message views, but also for code generation. We depended the decision on whether to base our meta-model on the UML meta-model on the possibility of ordering fragments through the meta-model. In the UML meta-model, the list of fragments in an interaction is ordered. The send and receive event of a message call are fragments of an interaction which allows to retrieve the order of messages. The message views in RAM currently only make use of synchronized message calls, which makes it possible to use this information. If concurrency support will be added to RAM and asynchronous messages will be modeled, the order of events cannot be retrieved by ordering the fragments. In that case it is necessary to explicitly specify the order of messages, e.g., by defining tuples of events in the form of $< e_1, e_2 >$ where e_1 is the preceding event and e_2 the succeeding event.

3.1 Overview of UML Sequence Diagram Meta-Model

In order to be able to compare sequence diagrams and message views and understand their differences we will take a look at the meta-model of *UML Sequence Diagrams* [20] defined by the Object Management Group (OMG). It is contained in the *interaction* package of UML which also allows the description of several other diagrams for different purposes among which are *Interaction Overview Diagram*, *Communication Diagram* and *Timing Diagram*. Therefore, the meta-model is defined in a very general way in order to allow the definition of all possible diagrams. We show the elements of the meta-model most relevant to sequence diagrams. An example sequence diagram is shown in Figure 3.1 where we pointed out areas and their corresponding meta-model elements. An excerpt of the UML meta-model of the sequence diagram is depicted in Figure 3.5.

An *Interaction* is the encapsulating unit of behavior representing a sequence diagram. Interactions focus on the interchange of messages between instances of classifiers. It itself is a *Class* through inheritance, meaning that,

[1]The Eclipse Modeling Framework (EMF) is further described in Section 5.1.1.

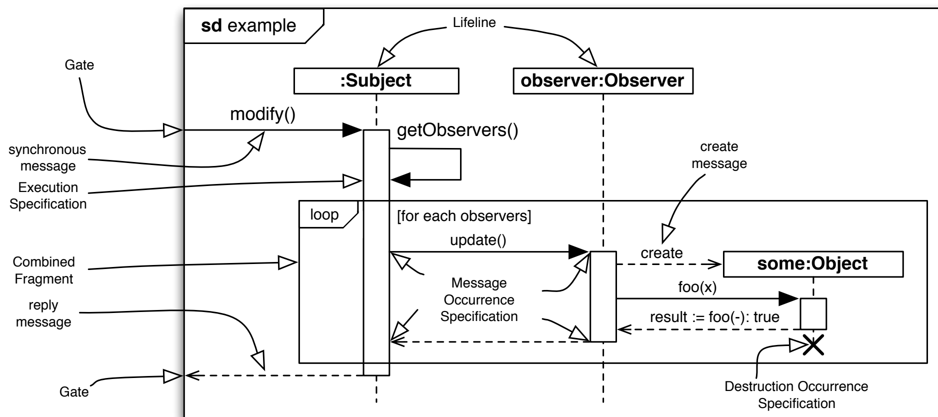

Figure 3.1: An example *UML Sequence Diagram* with corresponding meta-model elements pointed out.

for example, it has a name and owns properties. Furthermore, an interaction contains lifelines, messages, fragments (which is an ordered set) and formal gates.

*Lifeline*s represent the individual participants of an interaction. It is a *NamedElement* and references a *ConnectableElement*, which represents an instance of a classifier. For example, this could be an attribute. Furthermore, lifelines contain a list of references to fragments they are covered by.

A *Message* represents a specific communication between two lifelines. This can be the invocation of an operation, raising a signal or creation or deletion of instances. There are different kinds of communication possible: synchronized or asynchronous calls, reply, create or delete messages. Messages have a name and a defined syntax describes how the name of a message is built together. The message name contains the assignment of the return value to an attribute, the name of the operation called, arguments that are passed to the operation as well as specific return values. Using all information, a message name could look like the following: `result := contains(observer) : true`. The return value and assignment, however, are only used for reply messages. For the given example this would mean that a reply message assigns the return value `true` to the variable `result` when returning the call to `contains` with the argument `observer`. Furthermore, a message is associated to a send event and a receive event specifying the sender and receiver of the message. These *MessageEnd*s can be either a *Gate*, in case a message is coming from outside the scope of an interaction or leaving this scope, or a *MessageOccurrenceSpecification* denoting the occurrence of a message. This can also be a *DestructionOccurrenceSpecification* in case of the destruction of an instance. Both are an *OccurrenceSpecification* which in

turn is an *InteractionFragment*.

An *InteractionFragment* is the most general unit of an interaction. It is *abstract* meaning that the subtypes define what exactly that fragment is and represents. The fragments of an interaction can be the sending or receiving of a message, the execution of behavior, referring to other interactions, a constraint for the runtime, a combination of fragments allowing loops, alternations etc. Each *InteractionFragment* covers at least one lifeline. Most fragments cover exactly one, except for complex fragments that span across several lifelines. Furthermore, a fragment knows which enclosing interaction (or operand) it belongs to.

As described before, message occurrences are *InteractionFragment*s. An *ExecutionSpecification* represents the execution of behavior on a lifeline. The duration of the execution is designated by its associated start and finish occurrence. In most cases these are the receive event of a message and the send event of the reply. The visual notation is a box (thin rectangle) on a lifeline. This can be seen in Figure 3.1 when the lifeline of *Subject* receives the message *modify* and starts the execution until it is finished and returns the result. The execution of behavior can also overlap, for example, when on lifeline l_1 an execution starts by calling message m_2 of lifeline l_2. The behavior on l_2 makes a call back to l_1 using message m_3. In this case the box of the second execution overlaps the box of the first execution.

A more complex fragment is the *CombinedFragment*. It allows to specify combined behavior that is executed under certain circumstances. This includes critical, parallel and optional execution, alternatives, loops etc., which are referred to as interaction operators. A *CombinedFragment* consists of one or more *InteractionOperand*s depending on the operator being used. Each operand may have a constraint that defines under what circumstance the behavior inside the operand gets executed. The operand itself—like the *Interaction*—contains an ordered set of fragments. Nested combined fragments are therefore supported as well. The example in Figure 3.1 shows a combined fragment with the operator *loop*. It contains one operand that describes what is executed inside the loop. In this case it retrieves an element from the set and calls the *update* operation on this element.

Other fragments that are not relevant in our case but should be mentioned nevertheless are *StateInvariant*, for specifying runtime constraints, *InteractionUse*, for referring to another interaction that should be executed at that point and *Continuation*, for use in combined fragments with an *alt* operator to specify continuation of different branches.

3.2 Features of Message Views

UML Sequence diagrams can be described using a high level of abstraction. The UML specification [20] allows to provide a general overview of the

interaction, meaning that not many details have to be provided. For example, no concrete methods have to be declared, a message can just have a name describing what happens or what gets transferred.

Message views in RAM are tied to an aspect and define the behavior of the elements defined in the structural view. For each public operation of a class a message view has to be provided. Inside the message view only calls to itself or classes that are associated with this class can be made. This includes operations and classes that become available through mappings when instantiating other aspects. At the beginning, an analysis of the different features of message views had to be performed in order to be able to specify a meta-model that is sufficient and provides the level of detail that is necessary for message views. Using existing RAM models we evaluated the features of message views. A collection of main features of message views is shown in Figure 3.2.

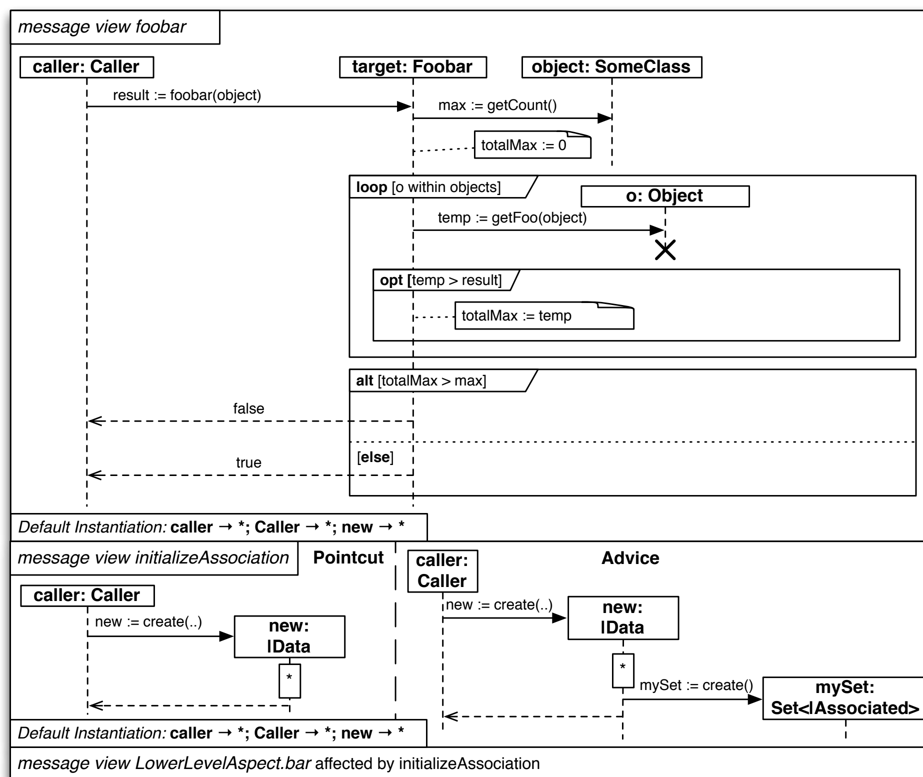

Figure 3.2: The main features of message views using the example of the `foobar` operation.

RAM distinguishes between three main kinds of message views.

Message View A *message view* (or *normal message view*) defines the behavior of a method. It highlights the interchange of messages that is executed when the described method is invoked. Additionally, a message view knows what aspects extend its functionality. We call this *affected by*, as this message view is affected by other (aspect) message views.

Aspect Message View An *aspect message view* defines behavior that extends other message views. It contains a pointcut and an advice. The *pointcut* can be the call of a method and is used to find a match in the affected message views. The *advice* describes the behavior that will be inserted at the found match point (also called *join point* in AO terminology). Additional behavior can be added before, after or around the original behavior. This is designated by a box containing the character '*' which represents the *original behavior*. The position of this *original behavior* box defines where the advice will be applied.

Message View Reference A *message view reference* allows to reference a message view coming from a lower-level aspect. An aspect that instantiates another aspect can extend the behavior of lower-level aspects. Therefore, it references such a message view and defines *aspect message views* that extend the functionality of this message view. The *message view reference* therefore just states additional *affected by* information for the referenced message view.

Using Returns In case an operation returns something, it is possible to assign the return to a property of the calling instance. Additionally, temporary variables are used to store values, e.g., the return of a message call, and to pass them as an argument to another message or to return it. Furthermore, when calling an operation, the arguments for the formal parameters can be passed. In the example message view shown in Figure 3.2 the return of the call to *getCount* and *getFoo* is stored in a temporary variable. When looping through all objects, the *getFoo* method of each object is called, passing the reference to the actual parameter *object* and the return stored in a temporary variable *temp*. While both techniques are also possible in sequence diagrams this can only be done by defining the messages' name. For message views in RAM this is not sufficient, as existing properties should be referenced.

Referencing Original Behavior The pointcut and advice both can contain an element that represents the original behavior of the advised message view. The notation of this element is a box containing the character '*' inside. Depending on the position of this box, the designer defines whether the advice gets woven in before, after or around the original behavior.

Catching Exceptions UML does not specify an interaction operator denoting a *try catch block* where exceptions can be caught and handled. A new operator is used as this is necessary in message views. The *disruptable* operator defines a combined fragment with an operand representing *try* and following operands each representing *catch*. At the end a *finally* operand can be used to define behavior that has to be executed at the end regardless of what happened.

Caller Lifeline Every message view contains a lifeline representing the caller of a message view. The first message between the *caller* and *target* serves as the beginning of the interchange of messages. In case there is a return the caller will retrieve the return from the target.

Default Instantiations Furthermore, message views have so called *default instantiations*. They define restrictions on the lifelines (or instances) and classes. For example, if there is more than one instance of a specific class, one of the instances could be specifically stated. On the caller of a pointcut this would mean that only calls from this instance would be considered. On the class this means that only calls to a certain class (i.e., the lifeline originally represents a super class) will be considered. The visual representation looks like a mapping as can be seen in Figure 3.2.

3.3 Current Meta-Model of RAM

Before the definition of the message view meta-model can be discussed, we will describe the current meta-model of RAM in order to allow an understanding of where the defined meta-model has to fit in and what existing elements can be reused or referenced. The meta-model described here evolved from a meta-model mainly focusing on the structural view to its current form as a preparation for this thesis. Furthermore, new requirements that occurred during the development of TouchRAM were integrated.

3.3.1 Overview

The unit that is modeled in RAM is an *aspect*. Therefore, this represents the root element of the meta-model which contains all other elements directly or indirectly. Generally, an aspect contains its different views. As currently only the structural view is supported, the meta-model contains only this view. Figure 3.3 depicts an overview of the current meta-model.

An *Aspect* is a *NamedElement*. Besides containing a *StructuralView* it can contain a *Layout* and many *Instantiations*. *Instantiations* describe the aspects that are instantiated and their mappings. The *type* of an instantiation describes whether it is a *depends* or *extends* instantiation. The *Mapping*

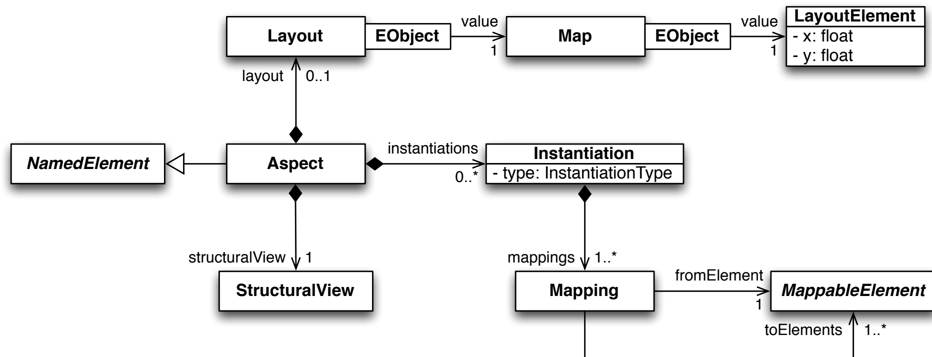

Figure 3.3: General overview of the current meta-model of RAM.

maps from an element of the instantiated aspect to one or more elements of the current aspect or another lower-level aspect that was instantiated. All model elements that should be mappable inherit from the abstract class *MappableElement*.

A *Layout* contains the layout for visualizing elements. Currently only their x and y position are saved as TouchRAM automatically layouts all elements. *Layout* contains a map. For each container element (the *key*), e.g., the structural view, a map (the *value*) is provided. That map itself contains a map from the actual element with a layout (the *key*) to its layout information *LayoutElement* (the *value*). The *key* can be any object of classes that are defined in the meta-model[2].

Furthermore, an *Aspect* contains a list of its *mandatory instantiation parameters*. This is a derived property as it can be computed from all elements of the structural view that are *partial*.

3.3.2 Structural View

The *Structural View* represents the class diagram and its basic structure is shown in Figure 3.4. It contains a list of *Classifier*s. *Classifier* is an abstract class that has a unique name among the structural view and may contain operations. The RAM meta-model distinguishes between classes an aspect introduces and classes that are reused from the programming language or a certain framework. In Section 2.2 we described how the aspect *ZeroToMany* reuses the class `java.util.Set` of Java (see Figure 2.1). An *ImplementationClass* inherits the properties of *Classifier* and additionally has an *instance class name* stating the class name of the implementation class.

An *Operation* has a name and attributes that describe the operation, i.e., whether the operation is *abstract*, *partial* or *static* and what visibility it has.

[2]In the Eclipse Modeling Framework (EMF) all classes inherit from *EObject*.

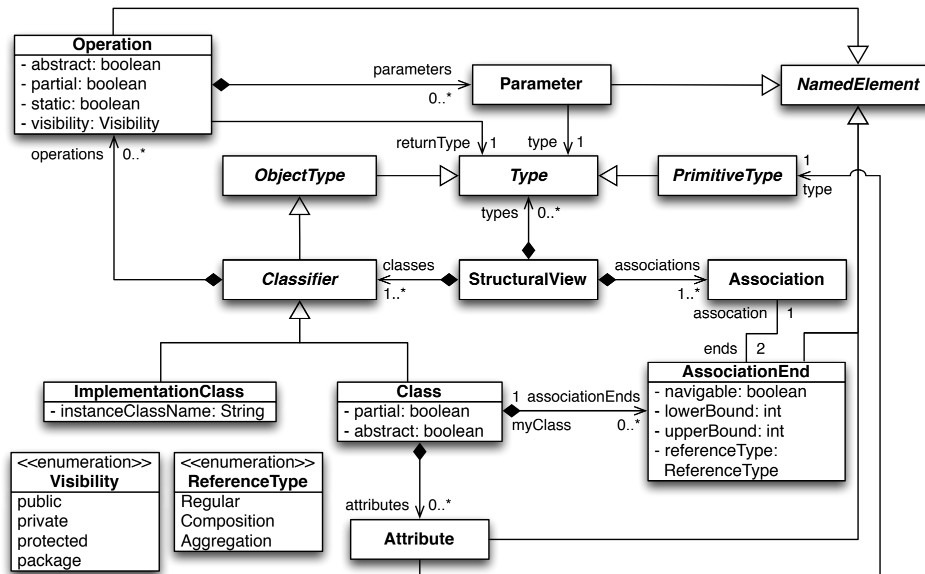

Figure 3.4: An excerpt with important elements of the structural view meta-model of RAM.

In addition, an operation has a return type and may have parameters. A *Parameter* has a name and a type.

A *Class* is a specialization of *Classifier* as well. It has attributes that allow a class to be defined as *abstract* or *partial.* A class may have one or more super types. Furthermore, it contains properties. The properties are distinguished between *Attributes* and *AssociationEnds*. This is different to the *UML Class Diagram* where a class contains properties which refers to both. An *Attribute* is a simple property of a class. It has a name and is typed, where the only possible types are primitive types. An *AssociationEnd* has a name and a lower and upper bound specifying the multiplicity of the property. A *reference type* declares the kind of aggregation of this property. This can either be *regular* for no aggregation, *aggregation* for a shared aggregation or *composition* for aggregated composition. Additionally, an *AssociationEnd* specifies whether it is navigable and it belongs to an *Association.*

Associations are directly contained by the *Structural View* and are associated with two ends. Therefore, the type of an *AssociationEnd* can be derived from the other end, i.e., the type is the containing class of the other end. Additionally, an association has a name which can be used for high level designs.

Some types are defined by the classes of the structural view. Each class is also an *ObjectType* which in turn is a *MappableElement.* Furthermore, operations are a *MappableElement.* Currently classes and operations can be mapped. The *StructuralView* allows to define certain *PrimitiveTypes* and

some special types. Among the special types are *void* and *any* which can be used as a return type of an operation. *Any* allows to specify that anything can be returned by an (partial) operation. In Figure 2.2, the operation |*modify* has such a return type. Furthermore, the following *PrimitiveType*s are currently defined: *boolean*, *int*, *char* and *String*. *PrimitiveType*s are at the same time *ImplementationClass*es as these types represent a type of the programming language being used. In Java, for example, for each primitive type a class exists allowing certain operations to be called on objects of these types.

3.4 Definition of Message View Meta-Model

The first step was to extract the required elements from the UML meta-model. As the UML meta-model is very general, allowing to use the same elements for several different diagrams, they contain numerous properties. Not all of them are required for sequence diagrams or for the message views. Figure 3.5 presents the core of the meta-model that was extracted with the necessary properties. The multiplicities of associations are mostly unrestricted. For the message view meta-model, we restricted this further to conform to the requirements. In the following sections we describe in detail the different aspects of the meta-model we defined allowing to model message views. This includes changes that had to be made in the structural view and changes that were proposed for message views. At the end we will show the complete message view meta-model.

3.4.1 General structure

An *Aspect* can contain several message views of different kinds. The three different kinds were explained in Section 3.2. For each kind, a class was introduced. As an *Aspect* can contain several different kinds, each class is a specialization of *AbstractMessageView*, which allows an *Aspect* to contain any kind and number of message views. The structure of the message views is shown in Figure 3.6.

Each message view can be affected by *AspectMessageView*s. For a *MessageViewReference* this is mandatory, as this is the purpose of specifying one. This restriction can be enforced through an additional constraint that checks for at least one *aspect message view* it is affected by.

In addition to that, a *MessageViewReference* references a *MessageView*. The referenced message view comes from an instantiated aspect.

The *MessageView* knows which operation it specifies. Besides, the message view contains the specification of the operation it specifies. However, this is not mandatory because *partial* operations usually don't have a specification. When their aspect is instantiated, a mapping will be provided for this operation, which in turn has a specification.

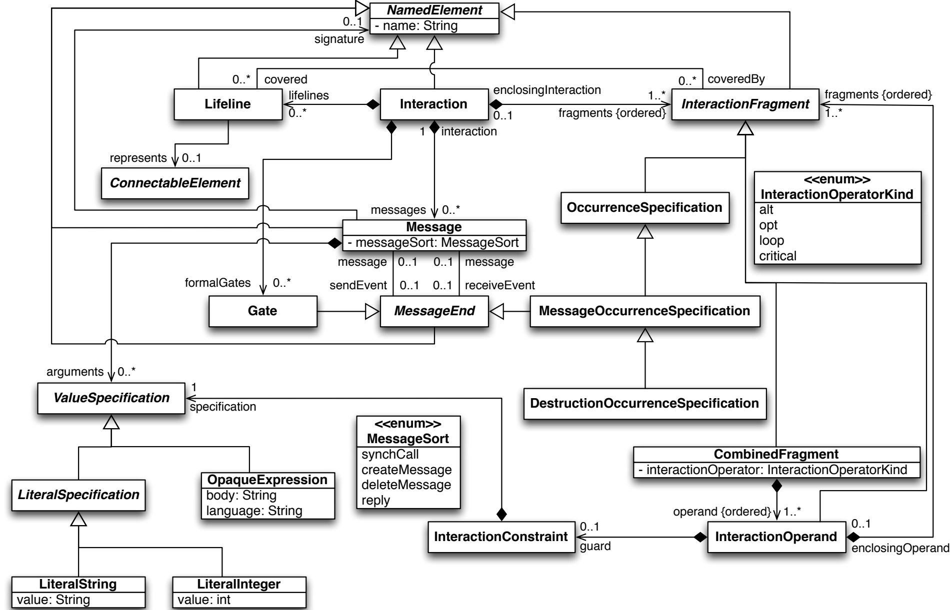

Figure 3.5: The core of the meta-model extracted from the UML meta-model for Sequence Diagrams.

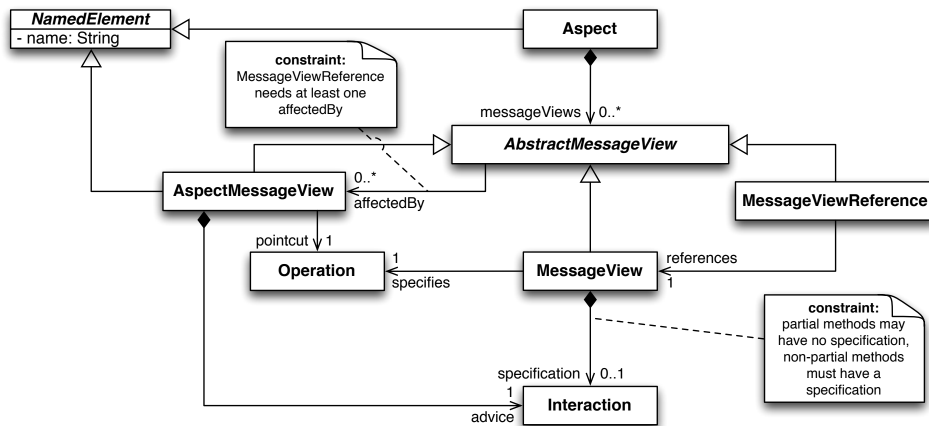

Figure 3.6: The structure of message views and their integration into an aspect.

Both *MessageView* and *MessageViewReference* don't have a name as this information can be retrieved from their referenced elements, i.e., the specified operation and the referenced message view (which in turn gets it from the specified operation).

AspectMessageViews are named describing their purpose. For example,

in the aspect *Observer* shown in Figure 2.2, the *aspect message view* that extends the functionality of the operation *modify* has the name *notification.* Its purpose is to notify every observer that is registered on the subject to receive updates. The *advice* of the *aspect message view* is—like the specification of a message view—an *Interaction* as well, having the only difference that this interaction contains the box for *original behavior*.

Pointcut

In the current graphical diagrams of message views, a pointcut is like an advice and therefore, an *Interaction* could be used as well to model the pointcut. However, in analyzing existing models of RAM aspects we noticed that all of the current models use the same (simple) pointcut kind. The example aspects in Figure 2.1 and Figure 2.2 show this call operation pointcut. The only exception is an old version of the aspect *Transaction* of the AspectOPTIMA case study [12]. It used a more complex pointcut where a message call between the start and commit of a transaction was used as the pointcut. However, in the current version of this aspect this pointcut is not used anymore.

For the current version of the meta-model we therefore decided to use a reference to the operation. A call to this operation represents the pointcut and will be matched. The pointcut can still be visualized in the same way. If other pointcuts have to be supported in future versions, the pointcut can, for example, be changed to an interaction that allows to model the pointcut.

3.4.2 Interaction

An *Interaction* is the unit encapsulating the actual behavior in the form of message interchange. For this purpose it contains *Lifeline*s, *Message*s, *Gate*s and the ordered set of *InteractionFragment*s.

An interaction, however, is not the only element that contains fragments. As described in Section 3.1 an *InteractionOperand* of a *CombinedFragment* contains fragments as well. Due to the fact that an *InteractionFragment* knows its enclosing element, in the UML meta-model a fragment has an association to the enclosing interaction as well as an association to the enclosing operand. Both have a multiplicity of *0..1* as Figure 3.5 depicts. In order to have one association to the container of an element we introduced *FragmentContainer*, which contains the ordered fragments. The *InteractionFragment* in turn knows the *FragmentContainer* it belongs to. Figure 3.7 shows the definition of *Interaction* including the changed part of the meta-model. Both *Interaction* and *InteractionOperand* are a subtype of *FragmentContainer*.

3.4.3 Local properties

Interaction itself may contain properties. This feature can be used in order to define an instance for a lifeline, e.g., which is the receiver of a call. In the UML

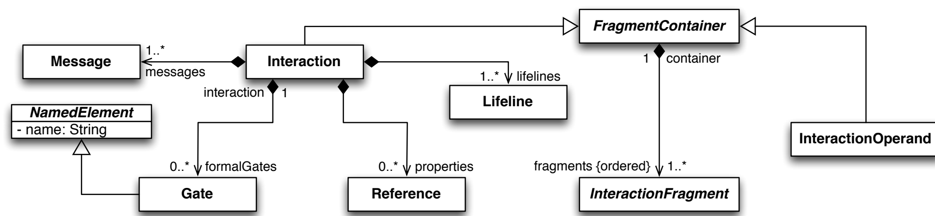

Figure 3.7: The definition of *Interaction* in the meta-model.

meta-model an interaction is a specialization of *Class* and therefore owns properties. In RAM however, the structural view meta-model distinguishes between attributes and association ends. Neither of them can be used for this case, because an *Attribute* has a primitive type and an *AssociationEnd* belongs to an association (see Section 3.3).

One of the features of message views is the possibility to use temporary variables. These can be used to temporarily store return values, reuse them in subsequent behavior, to be passed to another message or be used as the return value. An *Attribute* can be used for simple variables that have a primitive type. However, for instances of classes, the same problem as with the properties of an interaction exists.

Structural View Meta-Model Changes

In order to allow the creation of properties that have a class as its type, the structural view meta-model had to be adjusted. Different options were evaluated:

- Introduction of a *Reference* as a replacement for *Association* and *AssociationEnd*. A reference belongs to a class and has a type. One reference from class A to class B represents a uni-directional reference. A bi-directional reference is realized using two references, one from A to B and the other one in the opposite direction. Each reference knows its opposite if it exists. This alternative is based on the *Ecore* meta-model used by the Eclipse Modeling Framework (EMF).
- Introduction of *Property* as a replacement for *Attribute* and *AssociationEnd*. This is how it is realized in UML. A class owns properties, which can be used by an association. Besides, properties can be owned by the association itself, e.g., if it is a non-navigable property.
- Adding of an additional class *Reference* which allows the definition of properties. This way, only classes would be added to the meta-model. Existing classes wouldn't have to be changed, which doesn't require the change of existing instances (models) of the meta-model (meaning that existing models are still compatible).

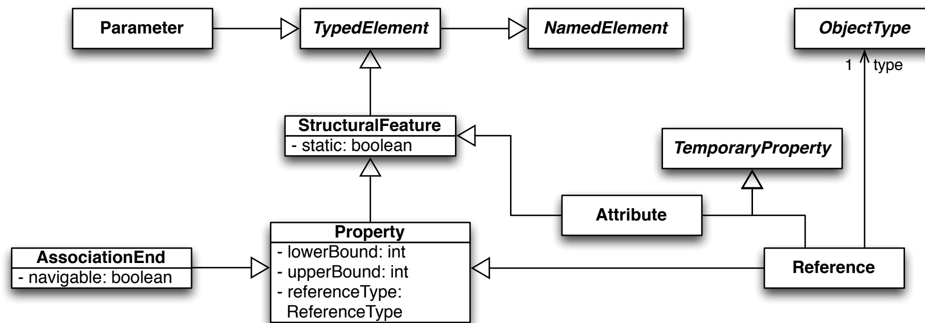

Figure 3.8: An excerpt of the structural view meta-model after the introduction of *Reference*.

On the one hand it was required to keep associations, as the possibility of defining association classes should be available in the future. Association classes are associations with additional properties. On the other hand, significant changes in an existing meta-model require the migration of existing models and the adaption of existing tools using a meta-model. On the basis of these requirements we decided to introduce an additional class *Reference*. The changes in the structural view meta-model are illustrated in Figure 3.8. As *Reference* and *AssociationEnd* share common characteristics, such as lower and upper bound and the type of the reference (i.e., the aggregation kind), a super type for both—*Property*—was introduced, which holds those properties. Additionally, a *Reference* has an association to *ObjectType*, which specifies its *type*. In this case the association cannot refer to *Type*, as this would include the types *void* and *any* as well. In order to be able to just have temporary variables that are either an *Attribute* or *Reference*, we introduced a super type *TemporaryProperty*. An *Attribute* and *Property* (meaning both *Reference* and *AssociationEnd*) are subtypes of *StructuralFeature*, used by the *Message*, which is explained in Section 3.4.5. Furthermore, a *StructuralFeature* can be static. This is not only required to specify static features of classes, but necessary for lifelines that represent a meta-class. We explain this in the following section.

3.4.4 Lifeline

As mentioned before in Section 3.4.3, temporary variables can be used in order to store values and subsequently use them. A temporary variable is usually available for a limited time. From a source code perspective this could be inside an operation body or inside a code block. In the case of a sequence diagram this could be during the execution of a message call that is visualized as a rectangular box on a lifeline (see Figure 3.1). However, the use of *ExecutionSpecification*s was omitted. Currently in RAM, only synchronous

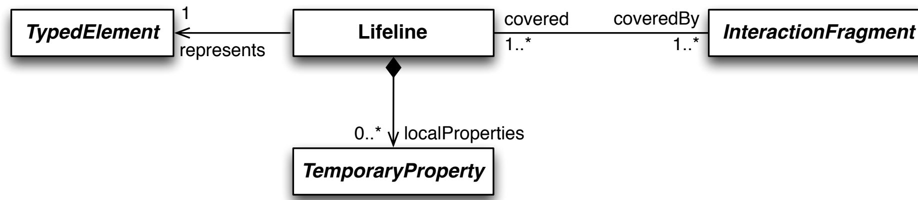

Figure 3.9: The definition of a *Lifeline* in the meta-model.

message calls are supported. The execution of a message on a particular lifeline yields from when the message call is received until the last call after receiving that call is made. Therefore, it is possible to keep the meta-model simpler. Furthermore, the visualization of message interchanges can still display execution specifications, if necessary, because it can be calculated based on the available information. If ever there is a need of integrating it in the meta-model, the meta-model can be extended at a later point.

Figure 3.9 presents the definition of a *Lifeline*. Additionally, a *Lifeline* knows which *InteractionFragment*s it is covered by and what instance it is represented by. In UML, *represents* is associated with a *ConnectableElement*, which can either be a *Property* or *Parameter*. In our case, it can be an *Attribute* (in the case an instance of a primitive type is required), a *Property* or a *Parameter*. All of those elements are either explicitly or implicitly typed. The *AssociationEnd* is implicitly typed through the containing class of the opposite end. For this purpose, the structural view meta-model was extended by a super type *TypedElement*, whose subtypes are *Parameter* and *StructuralFeature*. The part of the structural view meta-model including *TypedElement* is shown in Figure 3.8.

A lifeline can either represent an instance of a classifier or the classifier itself. The latter case allows to call static operations on the meta-class. The UML specification [20] does not specify explicitly how this is achieved. However, in UML, a feature of a classifier (e.g., a property) can be defined as static. In such a case it represents the classifier itself. The default case (non-static) represents an individual instance of a classifier. The meta-model of the structural view was extended with that information allowing a *StructuralFeature* to be declared as static, as described in Section 3.4.3. A *Reference* which is static can then be added to an *Interaction* as a property. *Lifeline*s representing this property would only be visualized with the name of the classifier and a stereotype << *metaclass* >>.

Discussion on Caller and Default Instantiations

A special case is the lifeline representing *caller* as illustrated in Figure 3.2. In order for a lifeline to represent a *caller*, a property with that type has

to exist. Consequentially a class *Caller* is required. This means that the structural view contains a class that has to be hidden from the view and the weaver has to ignore it.

The lifeline representing the caller was introduced in order to allow *default instantiations* to be defined for the caller of a message. As we described in Section 3.2, the *default instantiations* allow to restrict when a defined message view applies. For example, the caller could be restricted to a certain class or instance. This would be used for *aspect message views*, such that the pointcut not only matches if a certain operation is called. The additional restriction would enforce the advice to only be applied when the specified *default instantiations* match. In analyzing all existing RAM aspects, we noticed that the *default instantiations* are never set to any other than the default values, e.g., *caller -> *, Caller -> *, target -> **, meaning that there are no restrictions. They were used in older aspects based on an older version of RAM. A concept of RAM is to define an aspect as general as possible, as the aspect itself does not know where it will be applied or used. Thus, the *default instantiations* should be specified by the modeler on the higher level, i.e., the level where the aspect is instantiated. The modeler could restrict a particular *aspect message view* to be applied only in certain circumstances.

As currently this concept is not used and uncertain if it will be necessary, we decided to not integrate it into the meta-model. This resolves the issue with the *caller* and facilitates the ability to model an unknown sender and receiver of messages in a message view. UML uses formal gates in those cases as Figure 3.1 illustrates. Figure 3.13 highlights the updated visualization of message views.

3.4.5 Message

A *Message* defines communication between two lifelines in form of an operation call. The *messageSort* allows to specify what kind of message it is, i.e., a synchronous call, reply, create or delete message. The messages signature is defined by the referenced operation. Each message has a send and receive event defining what lifelines the message is connected between. An event can either be a *Gate* or *MessageOccurrenceSpecification* respectively *DestructionOccurrenceSpecification*. The return of an operation call can be assigned to a *StructuralFeature*, which allows referring to an existing feature of a class or a temporary property. Figure 3.10 presents the definition of these concepts in the meta-model.

In case the called operation has parameters, arguments can be defined. The UML meta-model has an ordered set of arguments where an argument is defined by a *ValueSpecification*. A *ValueSpecification* specifies a value, e.g., literals like *String* or *Integer* or more complex expressions. The order of arguments has to match the order of the operations parameters. Furthermore, the syntax definition of a messages name allows to specify parameters. Our

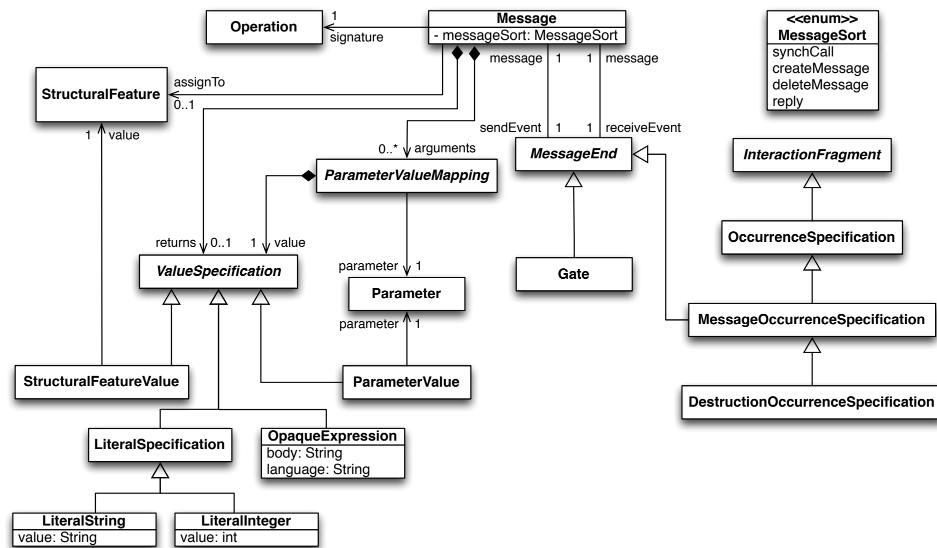

Figure 3.10: The definition of a *Message* in the meta-model.

goal was to be able to clearly state formal parameters and their actual parameters. Therefore we introduced a *ParameterValueMapping* that maps a formal parameter to an actual value as illustrated in Figure 3.10. The value is a *ValueSpecification*, however, we introduced a *ParameterValue* and *StructuralFeatureValue* allowing to refer to existing values.

When using reply messages, a modeler needs the opportunity to specify what exactly gets returned. Accordingly, a *Message* has an information regarding what it returns. This information is a *ValueSpecification* as well which allows to return properties of classes, temporary properties or specific values.

3.4.6 InteractionFragments

InteractionFragments distinguish between occurrences on a lifeline and fragments that are placed on a lifeline as Figure 3.1 depicts. As we described in Section 3.4.5, the send and receive event of a message can be occurrences.

From the UML meta-model the only other fragment that is relevant for message views are *CombinedFragment*s, as mentioned in Section 3.1. A *CombinedFragment* has an operator defining what kind it is. The current supported operators are *alt*, *opt*, *loop*, *critical* and *disruptable*. In addition, a *CombinedFragment* contains an ordered set of *InteractionOperand*s. The order is important, because *executing* one operand might lead to another operand not being *executed*. In Section 3.4.2 we explained that an *InteractionOperand* is a subtype of *FragmentContainer*, meaning that it contains an ordered set of fragments. Additionally, each operand may contain a constraint that

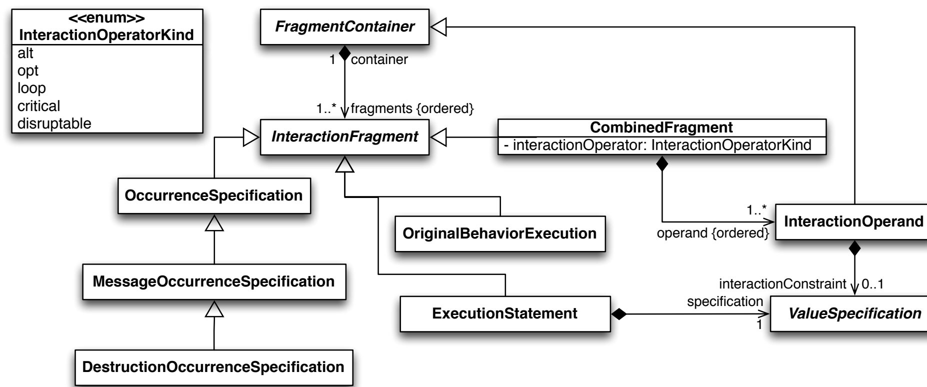

Figure 3.11: The existing *InteractionFragments* in the meta-model.

defines under what circumstances it gets used. For all operators of combined fragments—except *critical*—it is mandatory to specify a constraint. The constraint is given as a *ValueSpecification*. It can be described as an opaque expression as well, e.g., the constraint for a loop may be described in the way of the programming language: `Observer observer : observers` or `int index = 0; index < size; index++`. Figure 3.11 depicts all of the *InteractionFragments* presented in this section.

Some specific features of RAM message views—which we described in detail in Section 3.2—require their own specialization of *InteractionFragment*. The *OriginalBehaviorExecution* represents the execution of the original behavior of an operation which is visualized as a box containing a '*'. Execution of simple statements, like the initialization of a temporary variable, can be done using *ExecutionStatement*. It contains its specification as a *ValueSpecification*.

3.4.7 The Complete Message View Meta-Model

The complete message view meta-model containing all of the previously explained parts is presented in Figure 3.12. Almost all elements that we extracted from the UML meta-model were reused. The core of the UML meta-model is shown in Figure 3.5. However, additional information was added and the meta-model more restricted by making information mandatory through the multiplicity constraints.

3.4.8 Visualizing Message Views

The changes that were introduced for defining message views lead to a different visualization of message views. Furthermore, we proposed another particular change as to how message views are visualized. Previously, the

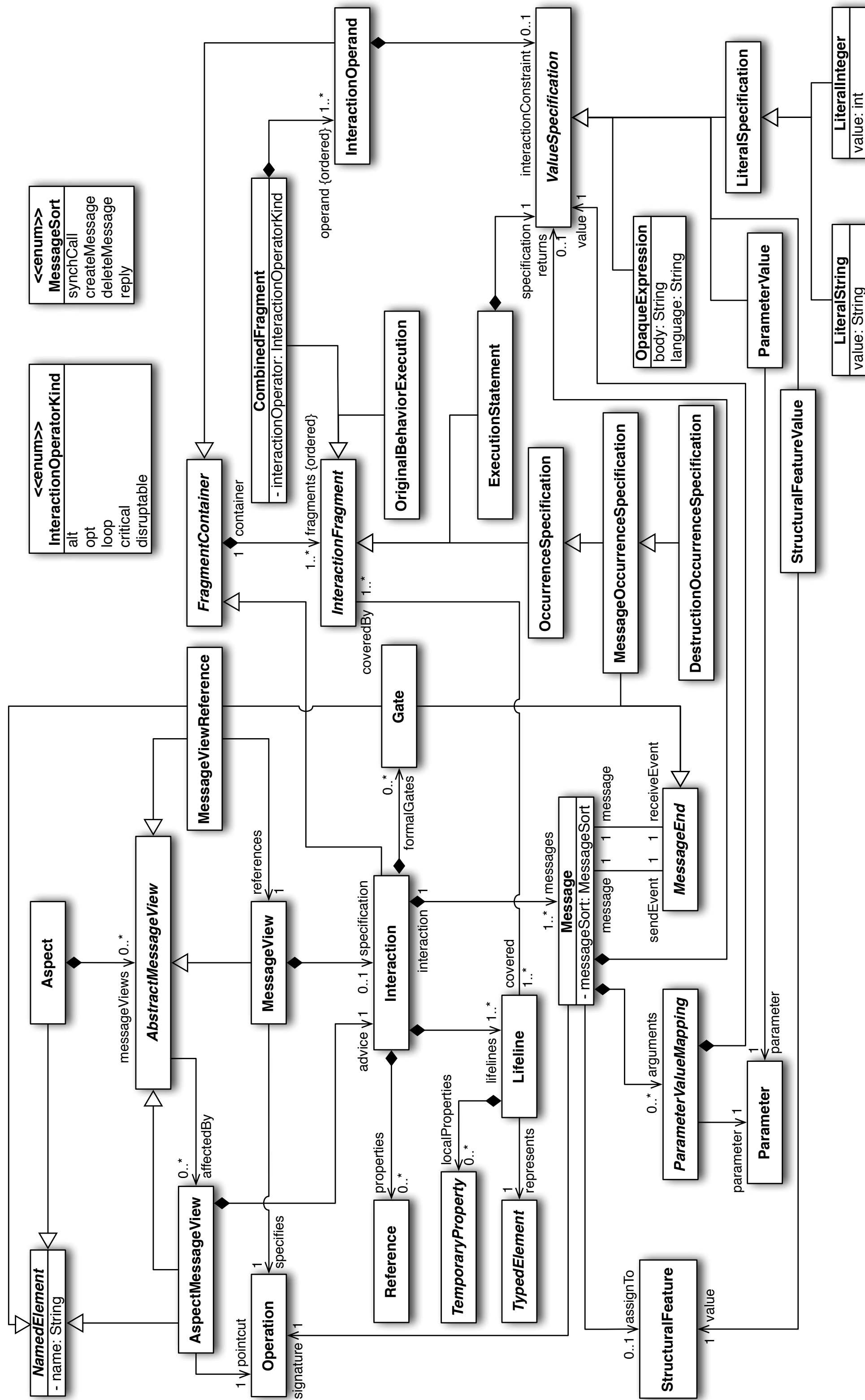

Figure 3.12: The complete message view meta-model.

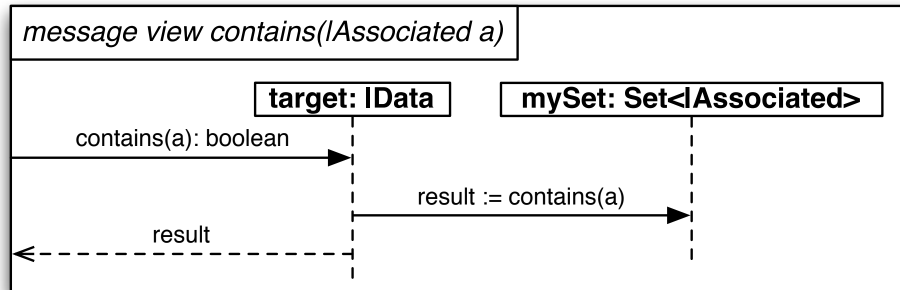

Figure 3.13: The updated visualization of message views using the example of *contains* of the *ZeroToMany* aspect.

return of a message call was stated on the calling message as Figure 2.1 shows. In the given message view of *contains* the calling message states that the result of the call is *result*. The purpose is to show that and what the call to *contains* returns and it might be misleading, especially as the caller was removed. However, a call can have several replies with different values. Therefore we proposed to show the return type of an operation on the first incoming message and the reply messages will show exactly what is returned. Figure 3.13 highlights the different visualization with the proposed changes. The *caller* lifeline and the *default instantiations* were removed.

3.5 Open Features

The main features of message views were integrated into the meta-model, making it possible to model message views with a high level of detail. There are more features that weren't been integrated yet. These are either message view or RAM related, but affect message views. We will discuss some of these features in this section.

3.5.1 Message View Features

Message Views for Getter and Setter

Operations for getting and setting a structural feature of a class always have the same behavior in most cases. A *getter* returns the structural feature and the *setter* takes the parameter and overwrites the current value of the structural feature with it. In current graphical diagrams of RAM aspects such message views are just specified as *getter* and *setter*, however, we haven't integrated it into the meta-model yet. While it is possible to explicitly define message views for them, doing so can be time consuming. In order to assist a modeler in the best way, a corresponding tool could assist the modeler by

offering to create the message view. Another option is to add this information to the meta-model. A possible way on how this can be achieved is to specialize *MessageView* with a type that represents *getters* and *setters*. Additionally it would have an association to a *StructuralFeature*, denoting which feature it returns or sets. However, when such a message view is extended with behavior by an *AspectMessageView*, it is necessary to create an *Interaction* for it with its specification in order to be able to weave additional behavior into it. The corresponding tool or weaver could take care of that in order to assist the modeler.

Throwing of Exceptions

When it comes to generating code, it is necessary to allow a modeler to specify in which circumstances an exception is thrown, and what kind of exception it is. UML has no notation to specify this in sequence diagrams. With our meta-model it is possible to create a temporary variable of the type of the exception to be thrown. A reply message could then return this variable. The semantic meaning, however, wouldn't be throwing an exception, as the reply message specifies what a message returns. No difference would be made between regular and exceptional execution paths. One could argue, however, that it is part of the message interchange between objects. In terms of generating code, the generator would be able to differentiate between regular replies and throwing of exceptions in order to create the corresponding source code.

Defining and Validating Complex Expressions

In certain situations it is required to define more complex expressions for constraints of an *InteractionOperand* or when specifying statements to be executed using an *ExecutionStatement*. When referring to variables or properties, only existing and reachable ones should be allowed to be referred to. For example, as depicted in Figure 3.2, when looping through the set of objects, two properties are contained in the constraint of the loop: *o* and *objects*. Such constraints should be evaluated and validated in order to assist a modeler allowing only the use of existing properties. This is an important prerequisite for generating code.

3.5.2 RAM Features

Conflict Criteria

Combining several aspects might lead to conflicts between some of them. This can occur when the aspects extend the same existing functionality and might cause inconsistency in the semantically correct functionality [13]. RAM contains so called *conflict resolution aspects* that allow to resolve these issues.

A *conflict resolution aspect* has the same features as an aspect, i.e., it contains the different views, but in addition, *interference criteria* can be specified that defines in what situations the content of the conflict resolution will be applied. *Conflict resolution aspects* are not integrated yet in the meta-model of RAM. However, as they contain the same features as aspects, the existing views can be reused. Only additional information like the *interference criteria* has to be added.

Reusing Result of Original Behavior

In current RAM message views it is possible in an advice to specify at what point the original behavior of the advised operation should be executed. Reusing the return of the original behavior, however, is currently not supported. It is possible, however, to catch exceptions that are caused by the original behavior by putting it inside a combined fragment using the operator *disruptable*. AspectJ, for example, contains a special method `proceed()` that can be called and the return be stored in a variable. The current concept of the original behavior doesn't facilitate the use of such a technique. Storing a return value is currently done when calling a message by assigning it to a property. Possible solutions could be the use of a special message kind or to call the same operation and treat it specially. This concept has to be further investigated in the future.

Parametrization

The ability to map an element from a lower-level aspect to more than one element from the higher-level aspect leads to an interesting issue. This can be illustrated using the *StockExchange* aspect from the example given in Section 2.2.1, which depends on the *Observer* aspect (see Figure 2.2). If we want to receive the update request for each information of *Stock* separately instead, we would map |*update* to more than one operation. Such an example is depicted in Figure 3.14 where *Stock* contains the operations *setName* and *setPrice* that are mapped to |*modify*. The *StockWindow* contains *updateName* and *updatePrice*, which are mapped to |*update*. This will lead to a problem as the relation between the mapped operations |*modify* and |*update* is unclear. When the advice of *notification* of the *Observer* aspect will be applied to *setName* and *setPrice* it is unclear whether to call *updateName* or *updatePrice*.

A possible way of achieving this has been proposed in previous research by parametrizing operations. In the given example, |*modify* would be parametrized by |*update*. The notation is |*modify*<|*update*>. The mapping for our example would then state |*modify*<|*update*> → *setName*<*updateName*>, *setPrice*<*updatePrice*>. This ensures that when *setName* is called it is certain that *updateName* has to be called. The current meta-model doesn't support this yet. Before any changes will be made, more research has to be

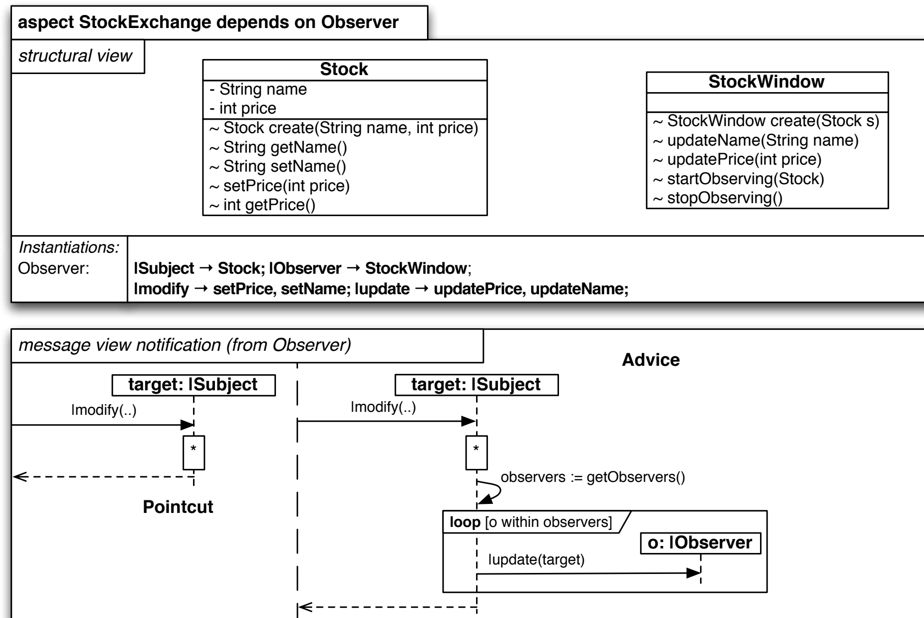

Figure 3.14: The issue of parametrization illustrated using the example of *StockExchange*.

done as other features regarding mappings have been discussed and should be considered.

The operation |*modify* is very general returning any type and receiving any number of parameters. This is denoted as '. . . ' inside the brackets of the operations definition instead of specific parameters. The idea is to allow it to be mapped to operations with parameters. We identified the following features that include this and other features which should be supported:

- Supporting any number of parameters. Operations that may contain any number of parameters are denoted as '. . . ' inside the brackets. Parameters can then be ignored or forwarded to another operation. In our example, the parameters of |*modify* could be passed to |*update*. *setName* would then pass the new name (parameter *name*) to *updateName*.
- Supporting any type of parameters. Parameters should be able to have any type (denoted using '*'). When an operation is mapped, the parameter then has to be mapped to a parameter of the mapped operation.
- Using one or several parameters. An operation can define specific parameters (e.g., |*m(String name, . . .)*), and use the specific one and, for example, pass the others to another operation.
- Support for introducing new parameters into operations. For example,

this could be used for a constructor. When a class is extended with a name, requiring the name to be given when the instance of this class is constructed.

These features are only allowed for partial methods. Supporting such features would offer a high flexibility to aspect modelers. In order to achieve this, a potential solution is to specify grouped mappings such that when mapping a container, its elements have to be mapped as well. For example, when mapping |*Subject*, its operations have to be given in the same mapping. As |*modify* itself is parametrized, a mapping has to be provided as well.

3.6 Facilitating future meta-model changes

In *Model-Driven Engineering*, models evolve during the development process. We will discuss the issues that result from this in this section, however, no solution has been implemented so far. This requires mechanisms to handle future changes in order to facilitate easy adaption for components that build on those models. As our meta-model is the foundation that other components are built on, this is of high importance. Even if no components exist yet that use the meta-model, instances of that meta-model that are created during the modeling process have to be updated as well. Appropriate mechanisms even facilitate changes in the future after a release of a software system.

During the development or modeling process of the meta-model, models often exist locally or at least in one place (like a version control system), and only people responsible for the development have access to them. In that stage, a solution to the evolution problem is not necessarily required. Changes to existing models that are not too complex can, for example, be made with the stream editor *sed*—a *Unix* command-line utility.

However, this is tedious and error-prone. Furthermore, once models are more wide-spread, e.g., when the software system is deployed, a more sophisticated solution is required. Different solutions are possible to achieve this:

- Transformations can be used to transform a model to a different model based on another meta-model. This can be used to migrate a model. However, this can be time consuming and requires extensive transformations as the complete model has to be transformed to a new instance, meaning that actions have to be performed for elements that weren't changed.
- The Eclipse Modeling Framework (EMF) provides an approach[3] where, when loading an instance based on an old version of the meta-model, adjustments will be performed in order to transform it into a model compatible with the new version of the meta-model. This allows

[3] See http://wiki.eclipse.org/EMF/Recipes#Recipe:_Data_Migration

a seamless process for the user who is still able to open old models without any interruption or having to explicitly import it.

- A project compatible with EMF called EDAPT[4] allows to record the changes being done to a meta-model. Additional code can be specified for complex changes in the structure. A component for migrating models can be generated in order to use it in an application.

A well-defined and well thought of meta-model is crucial, as it is the foundation of a modeling notation and therefore changes to it affect a whole range of components or tools. However, it can be necessary in some cases to perform changes. Additionally, changes shouldn't be prevented because the effects have a large impact. The options mentioned above assist in keeping old models compatible by migrating them.

[4] See http://www.eclipse.org/edapt

Chapter 4

Message View Weaving

The defined meta-model facilitates the creation of aspect models with message views. An important step for a modeler during the development process is weaving. Weaving combines aspect models together to a final woven model that has no dependencies. This final model can then be transformed into an executable version, the final application. However, weaving is not only important for the final application. It allows to assist a modeler during the design process. A modeler is enabled to see the results of the composition decisions made and correct them if they prove to be erroneous. Effects can be examined and verified. This facilitates the identification of errors early in the design process, which increases the quality of the software system.

Currently, it is possible to weave the structure of aspects and support for weaving is integrated in TouchRAM. Weaving of message views has only been done in theory. Before we will describe the formalization of the message view weaving algorithm in this chapter, we will briefly explain how weaving of structural views works. Understanding of this is essential, since weaving message view builds on structural view weaving.

4.1 Weaving Structural Views

The existing weaver for structural views offers two different functions. It is possible to weave a complete hierarchy, which we refer to as *complete weave*. All aspects the highest level aspect directly and indirectly depends on are woven into that aspect, resulting in an aspect without any dependencies. The weaver supports hierarchies of arbitrary depth. When weaving completely, the aspects are woven together in any order. Furthermore, a *single weave* weaves two specific aspects that are directly dependent within a hierarchy together, for example, when the modeler wants to see the result of applying an aspect to another one. The *single weave* can be used as a basis for implementing the *complete weave*: It is used to combine two directly related aspects, and then applied again to weave another aspect into the result, and so on. This

requires to update the instantiations, because an aspect woven into another one might depend on other aspects. The mappings therefore have to be updated.

Both of the weaving operations are based on the same algorithm. The algorithm weaves two aspects together, i.e., a *single weave* is performed. One aspect represents the *base* and the other the *dependee*. The *dependee* is the aspect being instantiated by the *base* and is conceptually woven into it. For the *complete weave*, this algorithm is executed until there are no instantiations left in the *base* and an aspect with no dependencies is retrieved.

The following steps summarize the actions performed by the weaver when weaving an aspect A into an aspect B:

1. **Pre-process *extends* instantiations**: Using *extends*, B can augment the functionality of an aspect A on the same level of abstraction. In this case, default mappings for all classes in A, where a class with the same name in B exists, are created. For each class where such a mapping is created, also all its operations are mapped. This leads to the classes being merged in step 3.
2. **Check for name clashes**: In class diagrams, i.e., structural views, class names must be unique, i.e., two classes having the same name are not permitted. If such a case is detected, the weaving is aborted with an exception requiring the modeler to resolve the conflict. The modeler can either rename the class or provide a mapping, which leads to the classes being merged in step 3. However, if there are two classes with the same name representing the same design, they are identical and will be merged. For example, in an aspect there could be two instantiations for the *ZeroToMany* aspect described in Section 2.2.1. If $|Associated$ is mapped to the same class both times , this will result in two *Set* classes that are identical, because they have the same type. Hence, only one *Set* class is sufficient.
3. **Weave**: The mapped classes from A and B are merged. When merging classes, all attributes, associations and information about super types are merged by copying them from A into B. For operations, if they are not mapped, they will be copied over as well. For classes with no mapping, the class from A is copied into B, including the containing elements. In this case, the visibility of operations is changed from *public* to *aspect-private* to ensure encapsulation of lower-level details [1].
4. **Update instantiations**: Because it is possible to weave any two directly dependent aspects within a hierarchy together, it is necessary to update instantiations at the end. For our given example in Section 2.2.1, when weaving *Observer* into *StockExchange*, the mappings of *ZeroToMany* in *Observer* have to be updated to now designate model elements of *StockExchange*. The resulting mappings are shown in Figure 4.1.

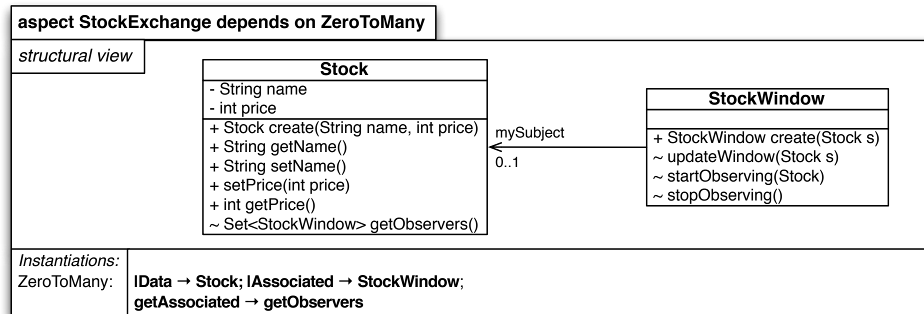

Figure 4.1: The resulting structural view after weaving the *Observer* aspect into *StockExchange*.

This yields the woven model $A + B$. When weaving completely, this woven model is taken to continue the weaving process. The process is repeated until there are no instantiations left. If only two aspects are woven together, the modeler can inspect $A + B$ and is able to weave other instantiations into it. In Figure 2.4 we showed the resulting model after weaving all dependent aspects into *StockExchange*. Figure 4.1 depicts the woven model when only *Observer* is woven into *StockExchange*. The resulting aspect now depends on *ZeroToMany* (as *Observer* depended on it). The mappings have been updated such that now *|Data* maps to *Stock* and *|Associated* to *StockWindow* now. The modeler would then be able to weave *ZeroToMany* into *StockExchange* which would result in an aspect model equal to Figure 2.4, which is equivalent to weaving *StockExchange* completely.

4.2 Message View Weaving Requirements

Weaving of message views is performed by weaving the *advice* of an *aspect message view* into *message view*s that have the *aspect message view* stated as being *affected by*. An *aspect message view* is only woven if it is specified as *affected by*. If it is not specified, the *aspect message view* will not be woven, even if the pointcut matches. In RAM, elements are aware of what other aspects or elements they are extended by. The advice of the aspect message view is woven in at the point related to the *original behavior*. This can be either before, after or around the original behavior.

The woven message views of the *StockExchange* example from Section 2.2.1 are shown in Figure 4.2. Message views that are not affected were omitted for space reasons as they weren't modified. The *aspect message view*s *initializeAssociation* and *cleanupAssociation* from *ZeroToMany* were woven into *create* and *destroy* of *Stock* after their original behavior. *notification* of *Observer* was woven into *setName* and *setPrice* of *Stock* after their original

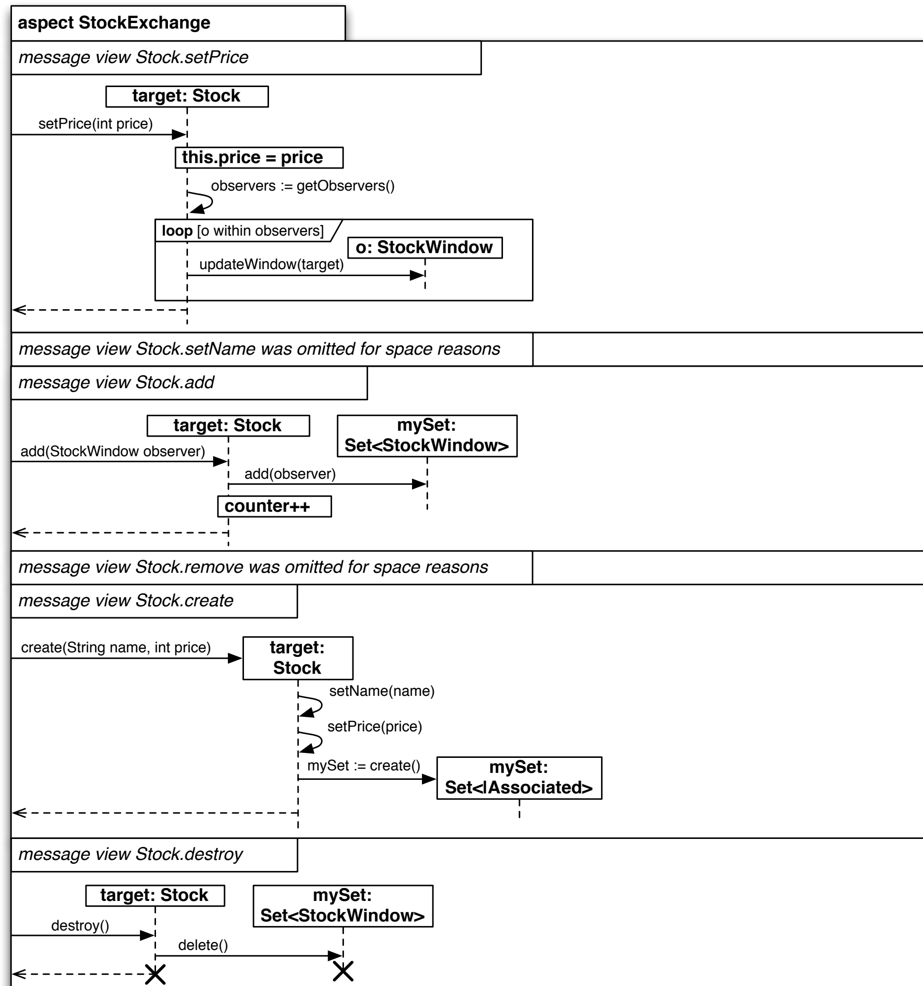

Figure 4.2: Some of the message views of the *StockExchange* aspect after weaving.

behavior.

However, the weaving of message views cannot be done in the same way as weaving structural views where the order is not important and any two directly dependent aspects in a hierarchy can be woven together. To illustrate why this is not possible we will extend our *StockExchange* example from Section 2.2.1. Let us consider that we want to know how often objects are added and removed from the association provided by the *ZeroToMany* aspect. In the context of the example this means that we want to know the amount of *StockWindow*s added or removed to a *Stock*, i.e., the calls made to *add* and *remove*. Therefore, we create an aspect that extends the functionality of *ZeroToMany*, adding a counter to *|Data* and modifying the behavior of *add*

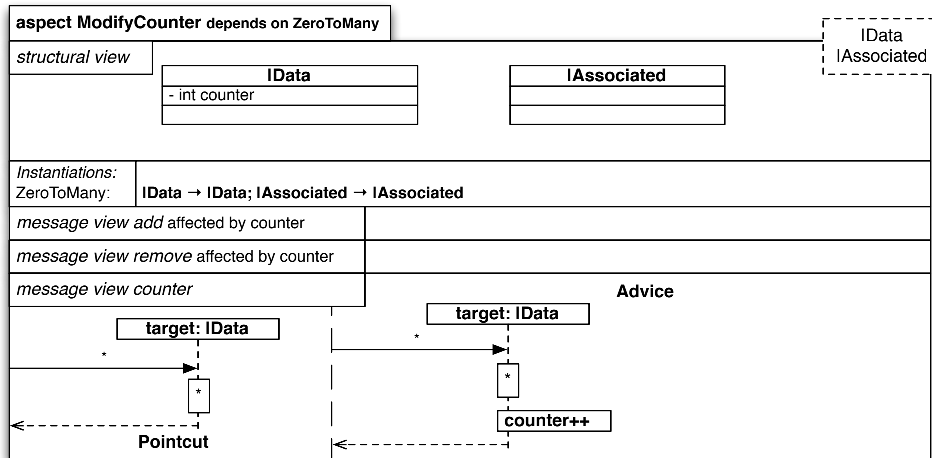

Figure 4.3: The *ModifyCounter* aspect which extends the functionality of *ZeroToMany*.

and *remove* to increment the counter. This is done by specifying an *aspect message view* called *counter* and two *message view references*, one of each referencing *add* and *remove*. *counter* is specified as *affected by* for both of them. This aspect called *ModifyCounter* is highlighted in Figure 4.3.

We then instantiate *ModifyCounter* in *StockExchange* and map *|Data* to *Stock* and *|Associated* to *StockWindow*. This will result in an aspect hierarchy that is depicted in Figure 4.4. The hierarchy has a diamond-like structure. *create* of *StockWindow* calls *startObserving* of *|Observer* which in turn calls *add* of *|Data*. The same applies to *stopObserving* and *remove* respectively. The *add* and *remove* being called should have the extended behavior provided by the *ModifyCounter* aspect.

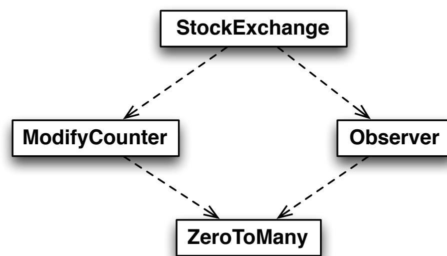

Figure 4.4: The dependency hierarchy of the extended *StockExchange* aspect.

Independent from the order of how the aspects are woven into each other, if for each two aspects that are woven, the structural view and the message views get woven, the weaver will have a conflict at the end. If this is done,

before *ModifyCounter* and *Observer* are woven into *StockExchange*, there will be two *add* and *remove* operations coming from each of them. The operations in *ModifyCounter* contain the *advice* that was woven in from *counter*, whereas the operations in *Observer* have their original behavior defined in *ZeroToMany*. The weaver has the conflict of which message view to choose that will be part of *StockExchange* and added to *Stock*. For our example this could be resolved by merging the two message views. This would require comparing them and figuring out the different parts of the message views in order to find the correct composition of both. While this might work for our example, it gets more and more complex for more sophisticated message views.

Therefore, the message view weaving defined in theory specifies that it is performed different to the weaving of structural views. When weaving two directly dependent aspects together, all message views simple are copied from the lower-level aspect to the higher-level aspect. At the end, when there are no dependencies left in the highest level aspect, all message views are woven. The weaving of message views takes place only once. For complex message views this also allows better performance, as no conflict resolution is required.

This requires to weave aspects in a depth-first order by beginning to weave the lowest-level aspect first and moving up. In our example, *ZeroToMany* would first be woven into *Observer* and *ModifyCounter* before weaving them into *StockExchange*. Furthermore, the same functions of the structural weaving have to be supported. Weaving a complete hierarchy (*complete weave*) and weaving two directly dependent aspects in a hierarchy together (*single weave*).

4.3 Formalizing Weaving

The requirement of copying message views and only weaving them once splits the weaving process into two phases. The first phase is done for each two directly dependent aspects A and B that are woven together. After weaving the structural view from A into B, all message views of A will be copied into B. Following that, in phase two all *aspect message view*s are woven into the message views that are affected by that *aspect message view*. When the modeler decides to perform a *single weave*, only phase one is performed. This defines the functions of the overall weaving process which includes the structural view, message views and state views. However, *state views* are not integrated yet. The main functions of the weaver and the two phases of message view weaving are described in more detail as follows.

A *single weave* weaves an aspect A into an aspect B. As both aspects are directly dependent in a hierarchy, this means that B instantiated A to reuse its functionality. Therefore, an instantiation for A exists in B, additionally

describing mappings. The formal algorithm for a *single weave* is defined in Algorithm 4.1. First, the structural view is woven as described in Section 4.1. The formal algorithm for weaving structure is beyond the scope of this thesis.

Algorithm 4.1: Weaving two directly dependent aspects together (*single weave*)

Precondition: *instantiation* must be an instantiation contained by *aspect*

```
1: function weaveSingle(aspect, instantiation)
2:     WeaveStructuralView(aspect, instantiation)
3:     weavingInfo ← weaving information of structural view weaving
4:     lowerLevelAspect ← aspect of instantiation
5:     CopyMessageViews(aspect, lowerLevelAspect, weavingInfo)
6:     return aspect
7: end function
```

In order to be able to correctly copy message views in phase one, information from the structural weaving process is required. In Section 4.3.1 we explain the details of copying message views and why this information is important. The weaver performing the structural view weaving has the knowledge of what element of an aspect A is now which element in B. When mappings are specified, the element from A gets merged with the element in B. Otherwise the element from A gets copied into B. In order to have this information available, the weaving of structural views has to be extended and the information added. We call this the *weaving information*. It is provided in form of a mapping, mapping from the element in A to a set of elements in B. The set of elements is necessary because an element can be mapped to more than one element. An aspect where this is required is *StockExchange*, which is shown in Figure 2.3. *|modify* is mapped to *setName* and *setPrice*. The mapping will therefore look as follows: $|modify \rightarrow \{setName, setPrice\}$. After weaving the structure, the *weaving information* is retrieved and used when message views are copied in Line 5.

When performing a *complete weave* for a higher-level aspect C, the complete hierarchy of C is woven into C. Aspects are woven in depth-first order, meaning that the weaver starts with the lowest-level aspect in the hierarchy weaving it into its *instantiator*. This is considered a *single weave* as two directly dependent aspects in a hierarchy are woven together. Resolving all dependencies is highlighted in Algorithm 4.2. Beginning from the highest-level aspect C, all instantiations are woven into it. In Line 8 the *single weave* is executed for those two aspects. This is only done once the instantiated aspect has no instantiations itself, i.e., is independent. If an instantiated aspect has instantiations, the dependencies of that aspect are resolved first. In Line 6 the algorithm is calling itself recursively to facilitate this. As explained in Section 4.1, this process is repeated until no instantiations are left, i.e.,

each aspect the *single weave* algorithm is executed for is independent. The result is an independent aspect.

Algorithm 4.2: Resolving all dependencies of an aspect recursively

Postcondition: *aspect* has no dependencies

```
 1: function ResolveDependencies(aspect)
 2:     while instantiations of aspect ≠ empty do
 3:         for all instantiation ∈ instantiations of aspect do
 4:             lowerLevelAspect ← aspect of instantiation
 5:             if instantiations of lowerLevelAspect ≠ empty then
 6:                 ResolveDependencies(lowerLevelAspect)    ▷ call recursively
 7:             else
 8:                 WeaveSingle(aspect, instantiation)
 9:             end if
10:         end for
11:     end while
12: end function
```

Algorithm 4.3 shows the definition of the *complete weave*. Phase one is performed in Line 2 by resolving all dependencies, weaving the structure and copying all message views into this aspect. After all dependencies are resolved, phase two is executed in Line 3 by weaving all message views. The details of how message views are woven are described in Section 4.3.2.

Algorithm 4.3: Weaving an aspect completely (*complete weave*)

Postcondition: completely woven *aspect* with no dependencies

```
1: function WeaveComplete(aspect)
2:     ResolveDependencies(aspect)
3:     WeaveMessageViews(aspect)
4:     return aspect
5: end function
```

4.3.1 Copying Message Views

Copying all message views from A (lower-level aspect) into B (higher-level aspect) requires additional steps for different cases. The steps are formally described in Algorithm 4.4.

- A *message view reference* references a *message view* from A and specifies what *aspect message views* it is extended (i.e., affected) by. When copying the message views to B, both *message view* and *message view reference* are on the same level. Therefore, the *affected by* information of the *message view reference* can be added to the referenced *message view*

Algorithm 4.4: Copying message views from the lower-level to the higher-level aspect

Precondition: *base* (higher-level) is aspect instantiating *dependee* (lower-level)
Precondition: $weavingInfo$ contains the mapping of woven elements from the structural view weaving

```
 1: function CopyMessageViews(base, dependee, weavingInfo)
 2:     for all message view type mv ∈ dependee do
            ▷ handle message view duplicates
 3:         if message view emv specifying the same operation as mv
                    exists in base then
 4:             emv[affectedBy] ← emv[affectedBy] ∪ mv[affectedBy]
            ▷ handle mapped operations
 5:         else if specified operation op of mv was woven and
 6:                 mv[affectedBy] ≠ empty then
 7:             for all woven operations wop ∈ weavingInfo[op] do
 8:                 if message view emv for wop exists in base then
 9:                     emv[affectedBy] ←
                            emv[affectedBy] ∪ mv[affectedBy]
10:                 end if
11:             end for
12:         else
13:             copy mv to base
14:         end if
15:     end for

        ▷ handle message view references
16:     for all message view references mvr ∈ base do
17:         if referenced message view mv of mvr exists in base then
18:             mv[affectedBy] ← mv[affectedBy] ∪ mvr[affectedBy]
19:             remove mvr from base
20:         end if
21:     end for

        ▷ update references
22:     for all message views mv ∈ base do
23:         update references of mv and all its elements
24:     end for
25: end function
```

itself, and the *message view reference* deleted. This step is performed in Line 16–21.

- Mappings for operations are provided in general for *partial* operations (from the lower-level aspect)—but can also be defined for non-partial operations. Partial operations don't require to have a message view, unless they are extended by an *aspect message view*. In such a case, there is an empty *message view* with *affected by* information specified.

In the higher-level aspect there exists a message view since the *partial* operation has to be mapped. This would result in two *message views* in *B*. Therefore, this case is handled in Line 5–11 by adding the *affected by* information of the message view from *A* to the *message view* of *B*, which has a specification. Furthermore, this has to be conducted for all message views that specify the mapped operations, as an operation can be mapped to more than one operation. In the *StockExchange* example depicted in Figure 2.3, this has to be done for the operations *setName* and *setPrice*. The *affected by* information of *|modify* is added to both of their message views. The message view from *A* is not required anymore and not copied to *B*.

- In case an operation is coming from more than one lower-level aspect and is added to the same class, this results in two message views being added. In our example *add* is woven twice into *Stock*. The first time coming from *ModifyCounter* and the second time from *Observer*. The operation is, however, only added once to *Stock*, as the structural weaver recognizes that the same method already exists, i.e., it has the same signature. However, this results in two message views. Because the message views are only copied and not woven, they have the same specification, but their information on what *aspect message views* they are extended by might differ. Therefore, the *affected by* information of the message view from *A* is added to the already existing message view in *B*. This step is handled in Line 3–4. The message view from *A*, however, still references the operation of a class in *A*. The new operation can be retrieved from the *weaving information*. In order to know if there is an existing message view, it is necessary to compare specified operations of existing message views with the woven operation. The message view from *A* is not required anymore and not copied to *B*.

After the message views were copied and the additional steps performed, certain references have to be updated in the message views. This step is performed in Line 22–24 of Algorithm 4.4.

Updating References

The meta-model of message views contains references to elements of the structural view. When copying a message view from the lower-level aspect *A* to the higher-level aspect *B*, the references still point to the elements in *A*. Therefore, it is necessary to update these references. This has to be done for all elements in the structural view. The elements that are referenced are described in the definition of the meta-model in Section 3.4. For example, when weaving *ZeroToMany* into *Observer*, the class *|Data* gets merged with *|Subject*. The message view for *add* of *|Data* references this class in the property *represents* of the lifeline representing *|Data*. This lifeline receives the call to *add*. Furthermore, the reference to *add* points to the operation of

the class contained in the structural view of *ZeroToMany*. All these references have to be updated after copying the message view to the higher-level aspect.

As described in the beginning of this section, when weaving the structural view the knowledge of what element in A is now which element in B is kept in the *weaving information*. Therefore, when updating references, for each affected element it just has to be checked whether the current referenced element is contained in the mapping as a key. If so, it will be replaced by the value of the mapping. An example of how this is done for a *message view* is described in Algorithm 4.5. A special case exists when an operation was mapped more than once, e.g., *|modify* in *StockExchange*. In the corresponding *aspect message view* called *notification*, the operation referenced as the *pointcut* and in the message in the *advice* is *|modify*. However, as we will explain in the following section, those references are not required when weaving the advice into a message view and it is therefore not required to update them. A possible resolution is to duplicate the *aspect message view* in order to have one for each affected *message view*. This then allows to update the references of *pointcut* and the message in *advice* for each *aspect message view*.

Algorithm 4.5: Copying message views from the lower-level to the higher-level aspect

$mv \leftarrow$ message view of type *MessageView*

1: **if** $weavingInfo[mv[\text{specifies}]] \neq \text{empty}$ **then**
2: $mv[specifies] \leftarrow weavingInfo[mv[\text{specifies}]]$
3: **end if**

4.3.2 Weaving

Once the preliminary work is done, all message views are contained in the highest-level aspect. Only *message views* and *aspect message views* exist at this point. It is then possible to compose these together by weaving the advice into the affected message views. The formal procedure is described in Algorithm 4.6. In order to do that, the weaver looks at all *message views* and their *affected by* information. Each advice of the specified *aspect message view* that affects a *message view* is then woven into the message view in Line 8. This process is simple because message views only contain synchronous messages. Only one message is possible at a time and messages are executed consecutively. Therefore, deterministic weaving is possible. In Section 4.4 we describe an approach that supports asynchronous messages as well and how those can be woven.

Before the weaving can take place, a special case has to be handled. When an aspect with *partial* classes and operations is woven completely,

Algorithm 4.6: Weaving all message views of an aspect

Precondition: structurally woven *aspect* with no dependencies

```
1: function WeaveMessageViews(aspect)
2:     for all message views mv ∈ base do
3:         if mv[affectedBy] ≠ empty then
4:             for all aspect message views amv ∈ mv[affectedBy] do
5:                 if mv has no specification then
6:                     create initial specification using advice of amv
7:                 end if
8:                 WeaveMessageView(mv, amv)
9:                 mv[affectedBy] ← mv[affectedBy] \ {amv}
10:            end for
11:        end if
12:    end for
13:    for all aspect message views amv ∈ base do
14:        remove amv from base
15:    end for
16: end function
```

some operations might have no specification. This is for example the case in *Observer*. In *ZeroToMany* there is a message view for *create* (the constructor) that is affected by *initializeAssociation*. |*Subject* in *Observer* has no *create* specified. When *ZeroToMany* is woven into *Observer* there will be a message view for *create*. However, it does not have a specification since it is partial. Before anything can be woven into it, a specification has to be created with the initial structure. This is handled in Line 5–7 of Algorithm 4.6. The initial structure includes the incoming call, i.e., the message and the send (a *Gate*) and receive events (a *MessageOccurrenceSpecification*), as well as the lifeline for the class receiving the call. The information for those objects can be taken from the *aspect message view* as this contains the same information. The required objects can therefore be copied and added to the specification.

Inside an *Interaction*, the contained *InteractionFragment*s are ordered. Therefore, they define the total order of behavior in a message view, i.e., how the message view is structured. The first fragment always is the receiving event of the incoming message call. This is the operation whose behavior is defined by the message view. As this fragment already exists in the message view it can be ignored. The location where the advice gets woven in depends on the position of the *OriginalBehaviorExecution* fragment. Figure 4.5 depicts the following locations that are possible:

- The *OriginalBehaviorExecution* is located after the first fragment. This means that the advice will get woven in after the original behavior. Consequently all fragments have to be added after the last fragment of the existing message view specification.

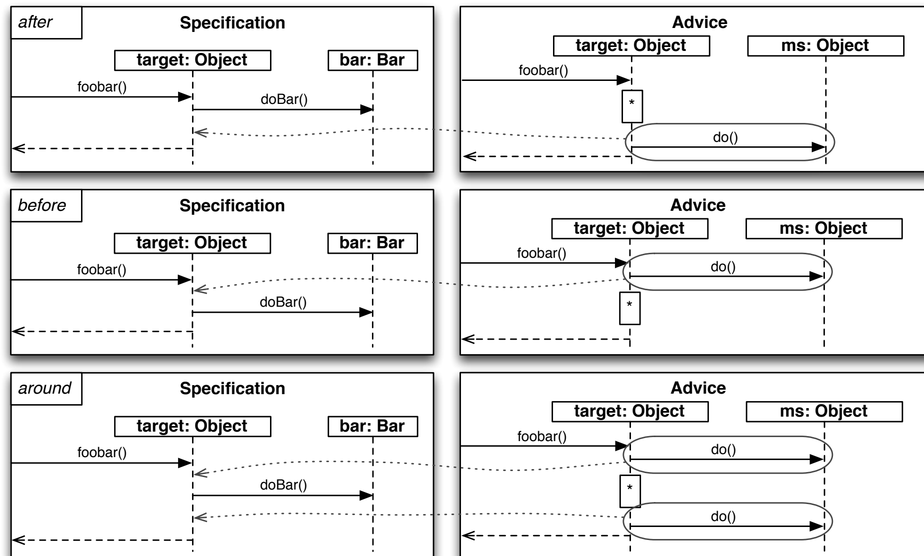

Figure 4.5: The different positions the advice can get woven in: After, before or around the original behavior.

- The *OriginalBehaviorExecution* is located at the end. This means that the advice will get woven in before the original behavior. All fragments therefore have to be added after the first fragment.
- The *OriginalBehaviorExecution* is located in between. This means, that the part of the advice before that fragment gets woven in before the original behavior. The part of the advice after that fragment gets woven in after the original behavior. Both cases are described above.

To handle this correctly, the weaver keeps track of the index in the list of fragments where the next fragment will be inserted. The index starts after the first fragment. The formal procedure is defined in Algorithm 4.7.

When the advice of an *aspect message view* (*amv*) is woven into a *message view* (*mv*), the weaver performs the following steps for each fragment of the advice of *amv* as specified in detail in Algorithm 4.7:

1. Each fragment of *amv*, except the *OriginalBehaviorExecution* and the first one, is copied and added into *mv* at the appropriate index in Line 11. When the *OriginalBehaviorExecution* fragment is reached, the index is set to the size of the set of fragments existent in *mv* in Line 7. This way the next fragment will be added at the end.
2. The copied fragment has to be updated. Algorithm 4.8 formally describes the steps executed. Updating a fragment includes the references to covered lifeline(s) and in case of *MessageEnd*s the corresponding message. In case the fragment is a *CombinedFragment*, this has to be

Algorithm 4.7: Weaving an *aspect message view* into a *message view*

Precondition: *mv* has a specification

```
1: function WeaveMessageView(mv, amv)
2:     nextIndex ← 1
3:     advice ← advice of amv
4:     weavingInfo ← ∅                  ▷ keep information about woven elements
5:     for all interaction fragments f ∈ advice[fragments] \ first fragment do
6:         if f is original behavior fragment then
7:             nextIndex ← size of mv[fragments])
8:         else
9:             fc ← copy of f
10:            UpdateFragment(fc, weavingInfo)
11:            mv[fragments][nextIndex] ← fc
12:            nextIndex ← nextIndex + 1
13:        end if
14:    end for
15: end function
```

performed for all fragments of the *CombinedFragments* operands as well. As nested combined fragments can occur, the function of updating a fragment is called recursively in Line 13 for each fragment of all operands.

3. In Line 2–8 a check is performed whether the covered lifeline in *amv* already exists in *mv*. A *Lifeline* already exists if there is a lifeline with the same *represents* (a *TypedElement)*. For example, the lifeline receiving the first call represents an instance of the class that contains the specified operation. If the lifeline doesn't exist yet in *mv* a copy of the lifeline from *amv* is created and added to *mv*. The fragments covered reference is updated and the information about the old and new lifeline kept for other fragments that cover the same lifeline. When a lifeline already exists, the local properties of the old referenced lifeline have to be copied to the existing lifeline.
4. In case the fragment is a *MessageEnd*, i.e., either a *Gate* or *MessageOccurrenceSpecification*, the reference to the corresponding message is updated in Line 17–23. As a message has a send and receive event, for the first fragment of a message, the message does not exist yet in *mv* and therefore has to be copied. When the second fragment is copied from *amv* to *mv*, the message already exists and only the reference has to be updated. Therefore the information about the old and new message is kept as well.
5. At the end, the *aspect message view* is removed from the *affected by* list.

After all message views were woven, all *aspect message views* are removed

Algorithm 4.8: Updating an interaction fragment of a *message view*

Precondition: *mv* has a specification

```
1: function UpdateFragment(f, weavingInfo)
2:     if covered lifeline of fc does exist in interaction then
3:         copy local properties of old lifeline to existing one and add them to
   weavingInfo
4:         weavingInfo[oldLifeline] ← existing lifeline
5:     else if covered lifeline of fc does not exist in weavingInfo then
6:         copy lifeline and add it to interaction
7:         weavingInfo[oldLifeline] ← copied lifeline
8:     end if
9:     update fc[covered] with existing or woven lifeline
10:    if fc is CombinedFragment then
11:        for all operands o ∈ fc[operands] do
12:            for all interaction fragments of ∈ o[fragments] do
13:                UpdateFragment(of, weavingInfo)          ▷ call recursively
14:            end for
15:        end for
16:    end if
17:    if fc is MessageEnd then
18:        if fc[message] does not exist in weavingInfo then
19:            copy message and add it to interaction
20:            weavingInfo[oldMessage] ← copied message
21:        end if
22:        update fc[message] with woven message
23:    end if
24: end function
```

in Line 13–15 of Algorithm 4.6. This leaves only *message view*s at the end.

Order of Weaving

The order in which *aspect message view*s are woven into a *message view* can be of importance. Let us consider an application where a modeler wants to integrate *authentication* and *transactions*. All calls to specific operations are secured by requiring a user to be logged in. Furthermore, calls to those operations should be made transactional. For this purpose, the modeler reuses the existing aspects *Authentication* and *Transaction*, which provide this functionality. Instead of starting a transaction and then checking the authentication, the expected result is to have the authentication check before a transaction is started. This depends on the order of how message views are woven, i.e., the woven result could be the unwanted version.

A possible approach is to consider the level of an aspect in the hierarchy. Functionality from lower-level aspects are woven in before functionality of higher-level aspects. For this to be feasible it is necessary that *affected*

by of message views are ordered. In Section 4.3.1 we described in what circumstances and how the information of *affected by* is merged. However, the information is not consistently added. For example, in the first case for *message view references*, the *message view* from the lower-level is kept and the information of the *message view reference* from the higher-level aspect added to this *message view*. This already is correct behavior. In the second case, however, the information of the lower-level message view is added to the one in the higher-level aspect, since the lower-level one doesn't have a specification. In this case it is necessary to adjust the behavior by adding the information at the beginning of the list. With the currently defined algorithm, further *aspect message views* are added to the list of *affected by* at the end. For the third case we defined that the existing *message view* is kept and the *affected by* of the *message view* from the lower-level is added.

Let us consider, that in our extended stock exchange example from Section 4.2, *Observer* extends *add* and/or *remove* of *ZeroToMany* as well. In that case, how the *affected by* is merged depends on the order in which the aspects are woven into *StockExchange*. Therefore, the order in which *affected by* is merged cannot be defined at this point. Generally, this represents the case in which two aspects are instantiated on the same level in a hierarchy. In our example using *Authentication* and *Transaction* these aspects would be instantiated on the same level. This requires a modeler to be able to specify the order of instantiations. Therefore, we propose to have an ordered list of instantiations and allow a modeler to order the instantiations in order to facilitate this.

4.4 Related Work

In [14], Klein *et al.* propose new formal definitions of join points which make it possible to detect pointcuts even when messages occur between the pointcut. *UML Sequence Diagrams* are used to express behavioral scenarios. Pointcut and advice are expressed by basic sequence diagrams (bSD), describing finite scenarios. Using combined sequence diagrams (cSD), bSDs can be composed using sequence, alt and the loop operator. Therefore, cSDs allow to define more complex behavior. Unlike RAM, their approach allows to use regular expressions in order to match many different messages in the pointcut. Asynchronous messages are supported and therefore, the events of the sequence diagrams are not totally ordered. A partial order is established using event couples, specifying a preceding and a succeeding event.

Due to the possibility of using sequence diagrams with finite behavior as a pointcut, a problem arises when weaving multiple aspects. If one aspect is woven in, another pointcut might not be matched anymore as other messages may occur between the messages specified in the pointcut. This happens if a strict sequence of message has to be adhered to. Therefore, Klein *et al.*

propose four different notions of join points:

- *strict part* is the most restrictive, enforcing a strict sequence of message without being surrounded by another message or a message in between.
- *general part* is the least restrictive, allowing to be surrounded by a message and messages in between.
- *enclosed part* is a variant of *strict part* which can be surrounded by a message.
- *safe part* is a variant of *general part* where the order on the events specified in a pointcut has to be preserved.

For each part they define a notion called *is part of* to determine whether a *bSD* is part of another *bSD*. In order to allow messages in between or surrounding messages, a morphism is used to map events from the pointcut to the base scenario. This results in four strategies for detecting pointcuts to overcome the issue of multiple aspect weaving.

In RAM, only synchronous messages are supported which prevents surrounded messages. Furthermore, the pointcut is defined as one method being called. The advice represents a sequence diagram, however, it can be of complex nature as *combined fragments* (for alternatives, loops etc.) are allowed to be used. When weaving multiple aspects, the problem of pointcuts not being detected anymore does not occur in RAM since the one method call can always be detected. Only if a method call would be removed completely by one aspect such a problem could occur. The approach of Klein *et al.* consists of a mechanism to prevent applying an advice within successive join points, where parts of a join point are used in another one. In our approach this is not necessary since it has to be possible to use the pointcut method for multiple aspects.

Another approach of Klein *et al.* [15] presents a semantic based weaving algorithm. It uses *Hierarchical Message Sequence Charts* (HMSC) which are based on *Message Sequence Charts* (MSC). Similar to the previously described approach, basic MSCs describe simple communications with finite behavior. HMSCs allow the composition of basic MSCs with operators such as sequence, alternative and loop. *UML Sequence Diagrams* (SD) and MSCs are very similar as SDs are inspired by MSCs meaning that the approach could be applied to SDs as well.

On the one hand, weaving can be performed on an abstract syntactic level, where a specific pattern is searched for and replaced by the syntactic definition of another behavior. On the other hand, it can be performed at the semantic level, where the semantics of the description define what part of a behavior is replaced with another behavior. Semantic weaving becomes necessary when message calls cannot be detected as a strict sequence of messages on the syntactic level, i.e., it doesn't appear explicitly in a scenario. This is required, for example, when loops are used and pointcuts that expand across loop boundaries have to be detected.

Therefore, they define a pointcut matching algorithm that allows to match pointcuts across basic MSCs when they are sequentially composed. A pointcut is detected within all executions of a scenario. This means that loops are unfolded in order to match such pointcuts. Potential matches are observed by studying started matches on longer executions. Through unfolding, the obtained HMSC can then be woven with syntactic weaving.

This approach as well allows pointcuts that are represented as a scenario of finite behavior without alternatives. As RAM does not use such pointcuts, the issue of matching pointcuts semantically does not arise.

Furthermore, both approaches don't support the notion of *original behavior*. The formal definitions of weaving replace the behavior matched by the pointcut completely with the behavior of the advice. In case of behavior being added, this requires to specify the same messages as in the pointcut in the advice as well plus additional behavior. In our approach, only the *original behavior* plus the additional behavior has to be specified.

Chapter 5

Message View Support For TouchRAM

In Chapter 3 a meta-model was defined which, in addition to the structural view, facilitates to model message views of aspects. Furthermore, in Chapter 4 weaving of message views was formalized. The previous steps now allow the integration of modeling and weaving message views into TouchRAM, the multi-touch enabled tool of the Reusable Aspect Models approach. At the beginning, this chapter presents some brief background on the frameworks used. Following, some insights into the definition of the meta-model, the implementation of the formalized weaving algorithm and the implementation of visualization support in TouchRAM are given. Furthermore, the chapter presents ideas for message view editing and suggestions on how the architecture of TouchRAM can be improved and ends with related work.

5.1 Background

5.1.1 Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) [23] is an Eclipse project offering a model-driven approach to create structured data models that can be used within applications. It allows to define meta-models based on *Ecore*, a subset of the *Meta Object Facility* (MOF) specified by the Object Management Group (OMG) [19] and very close to *Essential MOF* (EMOF). *Ecore* is similar to a class diagram of UML. An *Ecore* model (i.e., an instance of *Ecore*) defines the meta-model of a desired data structure. EMF provides a facility to produce different Java code from the defined meta-model:

- The *model code* that represents the model defined in *Ecore*.
- *edit code* provides adapter classes that offer viewing and command-based editing (e.g., for undo/redo) of the model.

- A generic editor supporting the creation, viewing and editing of models.
- And *test code* allows the generation of test cases that can be further extended.

The code is generated as ECLIPSE plug-ins facilitating to be used in ECLIPSE applications, but can also be used in standalone Java applications. Furthermore, EMF provides serialization in the *XML Metadata Interchange* (XMI) format. The model-driven mechanism allows to modify the meta-model at any time and re-generate code. Generated code can be modified and marked in order for the generator to ignore when generating code again.

5.1.2 Kermeta

The KERMETA workbench [8] provides a meta-programming environment on an object-oriented *Domain Specific Language* (DSL). Meta-model engineering is facilitated by providing an object-oriented language including lambda expressions similar to the *Object Constraint Language* (OCL). KERMETA is offered as an ECLIPSE plug-in and has support for EMF. The key features of KERMETA are aspect weaving and model transformation. Aspects can be described to extend the functionality of existing models, e.g., elements of an EMF-based model can be extended. Model transformations facilitate transforming models to different instances of a meta-model or from one meta-model to another. In TOUCHRAM, KERMETA was used to implement the weaver of the structural view. KERMETA code can be compiled into Scala code which runs in Java virtual machines, making it possible to integrate it into any Java application. Furthermore, an interpreter is included, however, it has a much longer execution time. During this thesis, a new version (2.0) of KERMETA was released, which intends to provide better integration with EMF as well as better performance.

5.1.3 Multitouch for Java (MT4j)

MULTITOUCH FOR JAVA (MT4j) [27] is an open source Java framework with a special focus on multi-touch. The framework supports creating visual applications in 2D or 3D using OpenGL for software or hardware accelerated graphics rendering. It is cross-platform compatible allowing to be run on the latest Windows, Linux (Ubuntu) and Mac OSX versions. MT4j is designed to support a wide range of input devices including multi-touch enabled ones. Besides the latest multi-touch software and devices in consumer electronics, such as Windows 7 and Apples multi-touch enabled trackpad, it supports the TUIO (*Tangible User Interface Objects*) protocol. Furthermore, regular mouse and keyboard can be used as well, and by providing an extensible event stack, different input methods can be used at the same time. Additionally, MT4j provides an extensible multi-touch gesture system with pre-defined common gestures and pre-built multi-touch enabled user interface components.

5.2 Overview of TouchRAM

An overview of the architecture of TouchRAM is presented in Figure 5.1. TouchRAM consists of the frontend, i.e., the graphical user interface (GUI), and the backend. The backend builds the base of the tool and contains the RAM meta-model and RAM model weaver. The meta-model defines the abstract syntax for RAM models and is defined using the *Eclipse Modeling Framework*. The weaver is invoked by the GUI on command by the user and performs the weaving process. It is implemented with Kermeta.

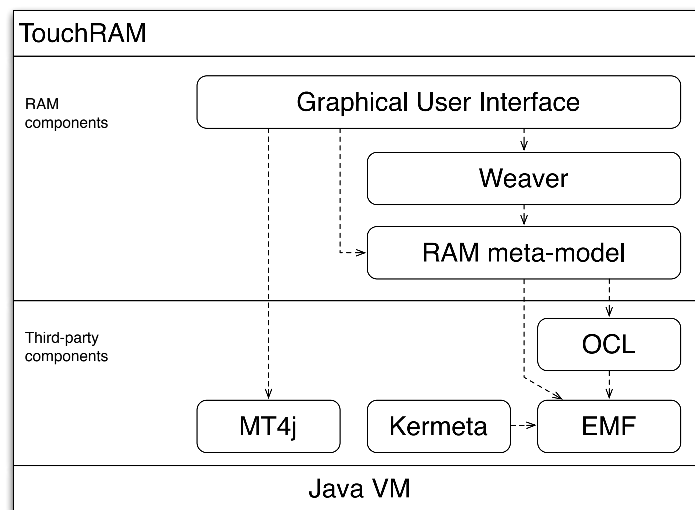

Figure 5.1: The architecture of TouchRAM.

The GUI of TouchRAM is realized using MT4j. All other components, however, are decoupled from the user interface (UI) based on a Model-View-Controller design. Therefore, EMFs built-in notification mechanism is used to observe the model and notify the user interface of any changes. The UI components of MT4j were extended for TouchRAM in order to support the developer. In particular, this includes layouts and a redefined event stack.

The current version of TouchRAM is capable of creating aspects with a structural view and weaving the structural views of aspects. Figure 5.2 shows TouchRAM and the visualization of the structural view of the *StockExchange* aspect from Section 2.2.1.

5.3 Meta-Model in EMF

The RAM meta-model is defined as an *Ecore* model. This allows TouchRAM to use the generated code as the base and to serialize RAM models in XMI. While the definition of the meta-model in *Ecore* is straightforward as per

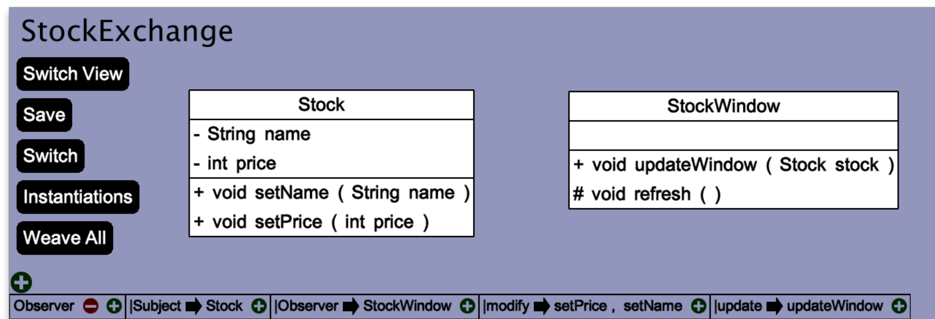

Figure 5.2: The visualization of the structure of the *StockExchange* aspect in TouchRAM.

the definition in Section 3, we will discuss some key points that the use of EMF facilitates and an idea on how the overall architecture of TouchRAM can be improved.

Using OCL The project *Eclipse OCL* [25] offers an implementation of the *OMG Object Constraint Language* (OCL) for EMF-based models. OCL can be used for constraints, derivation and operation bodies. When operations are defined in *Ecore*, the code generator creates empty operation bodies that the developer has to implement. This requires the modification of generated code, which will be ignored by the generator in future iterations. This can lead to errors when fundamental changes are made in a meta-model. OCL can be directly integrated in *Ecore* models, and the generator creates appropriate code. For message views, constraints were integrated which allow to verify correct models. This is especially important after message views were woven in order to check that the result is conforming. Besides, defining operation bodies allows some helpful additions. For example, the attribute name of primitive types should be consistent throughout models. Therefore, the operation *getName*—which already exists through the super type *NamedElement*—was defined for each primitive type. This results in an overwritten *getName* operation. The operation body just describes the return of the specific name. Furthermore, in Section 3.4.3 and 3.4.4 we describe the addition of *TypedElement* to the structural view meta-model in order to be used for lifelines. However, *TypedElement* does not contain the type information, instead each sub type has its own. In order to be able to retrieve the type of a typed element without having to consider the exact type, we introduced *getType* to *TypedElement*. For *Attribute*, *Reference* and *Parameter* this doesn't require any changes, as they contain this method (and now override it). Solely for *AssociationEnd* the operation body has to be specified. In Section 3.4.4 we described that in this case the type is the containing class of the opposite end.

Command-based editing *EMF.Edit*—the part of EMF that the generated *edit code* uses—allows to use command-based editing to facilitate fully automatic undo/redo functionality. Instead of modifying the model elements directly by calling the appropriate *setter* methods, commands are created and executed on a command stack. Therefore, we propose to use the commands to be able to offer undo/redo to the modeler. The current version of TouchRAM directly modifies the model, but as controller classes are already used for most model manipulation this can be easily integrated.

Making use of adapter classes The generated *edit code* contains an adapter class for each class in the meta-model. Adapters are required by EMFs facility when using command-based editing. In addition, adapters provide more functionality. An adapter, for example, provides a label provider with the method *getText* to retrieve a textual representation of a given object. Furthermore, an adapter contains property descriptors that are used for viewing and modifying properties of an object. It provides a method *getChoiceOfValues* that returns all possible choices that should be presented to the user. Both methods can be modified in order to adjust it to desired functionality. Therefore, we propose to make use of the provided functionality, which will lead to a better separation of concerns, higher code quality and make it easier for the developer to implement the user interface. Furthermore, it allows to use different views. For example, the generated generic editor of EMF could be used along TouchRAM as well. This supports the developer during the development phase of the tool as it allows to create, view and modify example models. To evaluate the feasibility we integrated this into the message view visualization, which we describe in Section 5.5.

5.4 Integration of Message View Weaver

Previously the weaver only consisted of the structural weaver. The functions of the structural weaver were called directly from within the tool. In Section 4.3 we described the overall weaving process consisting of weaving the structural view and message views. Therefore, we introduced a general weaver *RAMWeaver* which is responsible for orchestrating the weaving process and calling the *StructuralViewWeaver* and the *MessageViewWeaver*. Figure 5.3 depicts the overview of the weaver architecture. The architecture takes into account that at some point in the future a weaver for state views will be added. In that case, a class *StateViewWeaver* can be added and called from within *RAMWeaver*.

As mentioned in Section 4.3, the structural weaver has to be extended with the *weaving information* in order to keep the information of what element of the lower-level aspect is (woven to) which element in the higher-level aspect. The class *WeavingInformation* encapsulates a map from an object

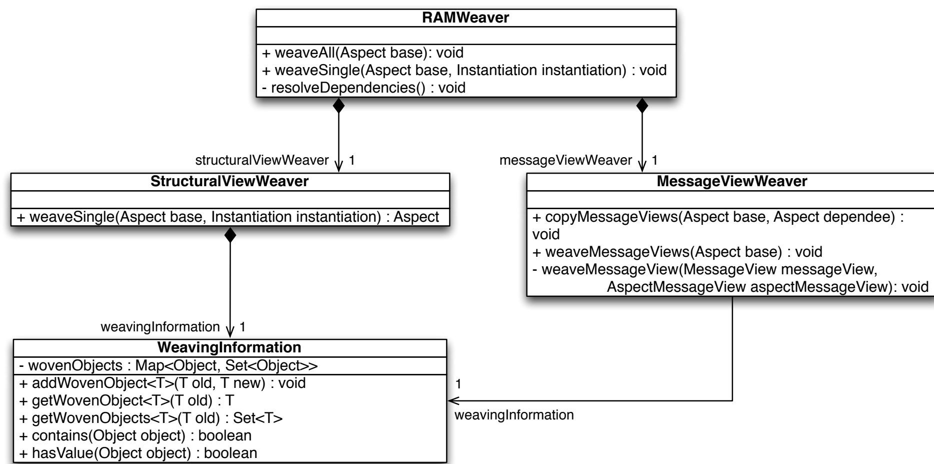

Figure 5.3: The structure of the weaver component.

to a set of objects. This information is stored in the *StructuralViewWeaver* for all elements that are or get mapped. Whenever an element is cloned and added to the higher-level aspect, this information is saved as well. The *weaving information* is stored for two aspects that are woven together, i.e., a *single weave*, and used when updating references in the first phase of copying message views. For the second phase, i.e., weaving message views, a separate *weaving information* is used to store woven elements of the message views.

5.4.1 Implementation Details

The implementation of the structural weaver was initially done with Kermeta 1.4. When implementing the message view weaver, we used the same version at the beginning, but due to several issues it was necessary to migrate to Kermeta 2, which was released during development time. The use of a hash table in the form of *Hashtable<Object, Sequence<Object>>*, with a set as the value, made it impossible to compile the weaver. Furthermore, copying message views completely caused an unresolvable exception. Trying to use Java code by reusing the *EcoreUtil* utility class of EMF wasn't successful as objects have to be transformed into special Kermeta objects. When just adding a message view to another aspect, Kermeta removes the message view from its original container. For example, this happens when using an aspect multiple times, like *ZeroToMany* in the extended *StockExchange* example from Section 4.2. With Kermeta 2 Java code can be called directly.

Exploiting Aspect-Orientation

One feature of Kermeta is the ability of aspect-orientation. Aspects can be defined for existing classes allowing to add new behavior or modifying existing behavior. Aspects can be defined for classes of Kermeta as well. For example, if an additional method is required for any object, an aspect class is defined for *Object*. This support for aspect-orientation is used for updating all references in message views in phase one. For all classes containing references to elements of the structural view, an aspect is defined adding the method *updateReferences*, which takes *WeavingInformation* as a parameter.

This method is first called on the message view types. Each type defines its own method, updates its references and forwards the call to its children. Listing 5.1 depicts the implementation of *updateReferences* for *MessageView*, which delegates to its specification (*Interaction*) after updating the reference to the specified operation.

```
aspect class MessageView
{
   method updateReferences(weavingInformation : WeavingInformation) : Void is do
      if (weavingInformation.contains(self.specifies)) then
         self.specifies := weavingInformation.getWovenObject(self.specifies)
      end

      // specification can be null if it is an empty message view
      if (not self.specification.isVoid()) then
         self.specification.updateReferences(weavingInformation)
      end
   end
}
```

Listing 5.1: Updating references shown using the example of *MessageView*.

In order for the message view weaver to be able to call *updateReferences* on any message view kind, an aspect is defined for *AbstractMessageView* as well with an empty operation body. The same applies to *InteractionFragment*, which allows calling the method on any of its sub types without knowing what type it exactly is.

Additionally, when fragments are updated, a check is performed to find any lifeline that already exists in the current message view. A lifeline is defined by the element it represents. For this case, in the lifeline aspect class, *equals* is overridden, taking into account whether the represented element is equal as well.

The use of aspect-orientation allows to keep the responsibility to the element itself, especially as they are aware of their own structure. It facilitates clearer code as no utility classes are necessary. Furthermore, it increases the maintainability. However, a developer has to be aware that additional

behavior is introduced through aspects.

Updating fragments

When copying an *InteractionFragment*, the reference to the lifelines covered is not copied. This is due to the association being bi-directional between *Lifeline* and *InteractionFragment*, and the copier of EMF therefore does not update the reference. This requires to keep the original fragment in order to update *covered* accordingly.

To find an existing lifeline, the covered lifeline of the original fragment is copied and *updateReferences* called on it. This updates the *represents* reference and all local properties. Next, all existing lifelines are compared to the updated lifeline to check whether this lifeline already exists. As described in the previous section, *equals* of *Lifeline* was adjusted for this case. A special case exists if the represented type is a *Reference*. Let us consider that the aspect message view *initializeAssociation* of *ZeroToMany* (see Figure 2.1) is woven into *create* of *Stock* (see Figure 2.3). In *initializeAssociation*, the lifeline receiving the operation call is named *new*. The name results from a *Reference* of type *|Data* with that name. Accordingly, the lifeline of *create* of *Stock* is named *target*. Both lifelines represent the same lifeline when *StockExchange* is woven, however, if comparing all properties of *Reference* the weaver would not recognize this. Therefore, in case of a *Reference* all properties except name are checked for equality. Due to this, *equals* is not overwritten as this could lead to side-effects.

In case of a *CombinedFragment*, all contents are already copied, however, fragments of all operands have to be updated as well. As *covered* of those fragments is not set, the original fragments have to be looked at. Listing 5.2 shows how this is implemented. Using the index of the original element (*InteractionOperand* and *InteractionFragment*), the copied element is retrieved. For each copied fragment, the method *updateFragment* to update a fragment is called recursively. This at the same time allows to handle nested combined fragments.

For each type of *MessageEnd* the message has to be updated. A message has a send and receive event. Therefore, for the first occurrence, the message has to be copied, its references updated and added to the interaction. It is then added to the *weaving information*. When the second occurrence is updated, the message from the *weaving information* is just taken and the fragments message updated. Furthermore, a reply message has a *Gate* as its receive event. As *Gate* is not an *InteractionFragment* it wasn't copied before. This is done explicitly at that point.

```
if (fragment.isTypeOf(CombinedFragment)) then
    var combinedFragment : CombinedFragment init fragment.asType(
            CombinedFragment)

    combinedFragment.operands.each { operand |
        // get the copied operand that corresponds to the current operand
        var operandCopy : InteractionOperand init fragmentCopy.asType(
            CombinedFragment).operands.elementAt(combinedFragment.operands.
            indexOf(operand))

        // go through all fragments of the operand
        operand.fragments.each { oldFragment |
            // get the copied fragment that corresponds to the current fragment
            var operandFragmentCopy : InteractionFragment init operandCopy.
                fragments.elementAt(operand.fragments.indexOf(oldFragment))

            // recursively call this operation as nested CombinedFragments are possible
            updateFragment(weavingInformation, specification, oldFragment,
                operandFragmentCopy)
        }
    }
end
```

Listing 5.2: Updating the *covered* property of all the contained fragments for a *CombinedFragment*.

Testing

Weaving is a complex process and it is necessary to verify the correctness of weaving results during the development phase. As described in Section 5.1.1, constraints are used for the meta-model, making it possible to perform checks during modeling. The woven model can then be checked in order to verify that all constraints are still fulfilled. Kermeta supports the definition of unit tests that can be run. This mechanism was used in order to define different test cases that focus on different parts of the weaving process. Furthermore, an approach used for testing the structural weaver is to model the expected results of the weaving and compare it with the actual weaving result. This can be done either manually using tools that allow comparing or integrated into a test case. However, more research is necessary in order to find a suitable solution. EMF provides tools to compare models and an evaluation is necessary to find out whether this can be used as part of test cases.

5.5 Displaying Message Views in TouchRAM

The integration of the message view meta-model into the RAM meta-model makes it possible to define RAM models with message views. While there is no tool support yet, message views can be defined using the provided editor

of EMF. The implementation and integration of the message view weaver into the existing weaver enables integration of weaving into TouchRAM. These preliminary works enable to implement the visualization of message views in TouchRAM.

When displaying an aspect, only the structural view was shown previously. Therefore, an additional button was introduced allowing the user to switch between the structural view and message views. Switching hides the current view and shows the other view. All message views are added to a container which places the message views one below the other. This is sufficient for viewing, but when editing message views is introduced a modeler might just want to see the particular message view. The container furthermore allows to pan the message views in case they don't fit completely on the screen. Panning is restricted to the y-axis.

For each element of a message view that has to be visualized, a view class is specified. Each view is responsible for visualizing that particular element. The container of an element is responsible for placing the element at the appropriate position. One extension of TouchRAM for MT4j is the introduction of layouts. Each view class inherits from *RamRectangleComponent*, which is an extension of the MT4j class *MTComponent* and offers some general features such as automatic child resizing and handling of events. Currently, horizontal and vertical layouts are defined that place the elements either in a column vertically or a row horizontally. The layouts additionally take care of the size of elements. However, message views cannot make use of those layouts as elements have to be placed differently and might overlap each other. This requires to set the width and height of an element explicitly. In the future, a special layout should be implemented which takes care of that, because setting the width and height at creation time can be difficult. Often the width of the parent or the width of children has to be considered. This information is available once the element is added to a container. In the current architecture, components (views) are built in the constructor. For some views it was necessary to move it to a separate build method, which is called after the component is added to its containing component.

The views *InteractionView* for *Interaction*s and *InteractionOperandView* for *InteractionOperand*s both share behavior as both of them are containers of fragments. However, the *InteractionView* is responsible for building elements in the view. Therefore, *InteractionOperandView* delegates the responsibility for building its contents to the *InteractionView*.

Elements like messages and lifelines are placed by adding some extra space to the position of the latest added element. Fragments use the position of messages. The sending and receiving ends of a message are not visualized, however.

When visualizing an *aspect message view*, the content is split between *pointcut* and *advice*. As a *pointcut* just refers to an operation, an *Interaction* is created based on the operation and the advice. This allows to visualize

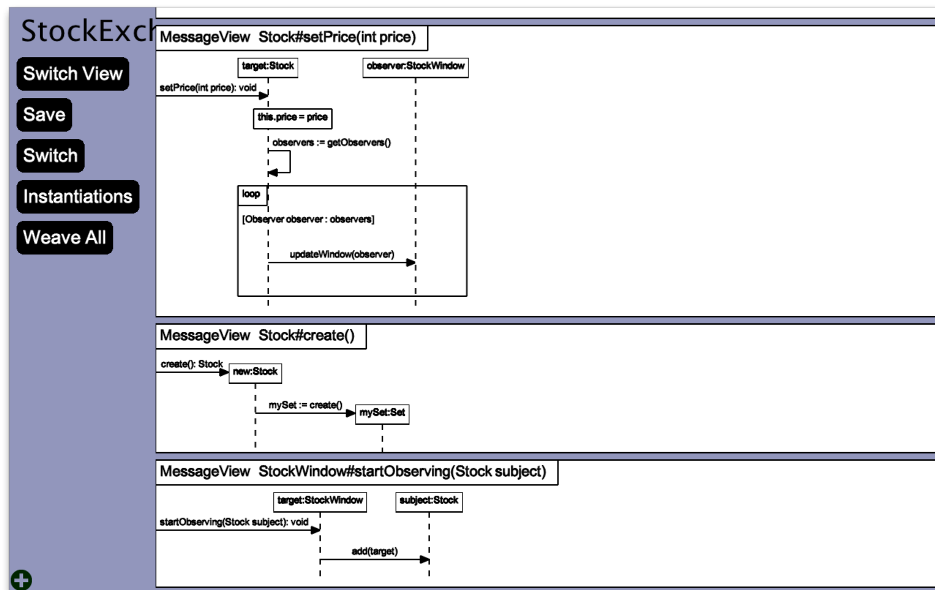

Figure 5.4: The visualization of message views in TouchRAM using the example of the woven *StockExchange* aspect.

the pointcut and the visualization of an *Interaction* is reused.

Figure 5.4 shows the visualization of message views in TouchRAM. Note that all the textual representations of elements are retrieved from the adapter classes (item providers).

5.5.1 Using adapters in TouchRAM

As described in Section 5.3, one goal of the visualization was to integrate the capabilities EMF.Edit offers. The first step is to adjust the *getText* method of the item provider in order to receive the appropriate textual representation of an element. Listing 5.3 shows the definition of *getText* for the item provider of a *MessageView* (*MessageViewItemProvider*). In order for the generator to not modify a modified part of source code, the annotation `generated` can be extended by `NOT` as defined in Line 5. The textual representation for the specified operation is received by delegating it to the item provider of the operation in Line 15, thus reusing other item providers. The method returns the label, i.e., the operations textual representation, after the name of the type in Line 20. The type name is not omitted in order to ensure compatibility with the generated editor. The generated editor provides a generic tree with the contents of the model. If the type name is not shown in front of an element it is difficult for a user to distinguish between different element types. When used in TouchRAM, the type name is simply stripped

before shown in the user interface.

```
1 /**
2  * This returns the label text for the adapted class.
3  * <!-- begin-user-doc -->
4  * <!-- end-user-doc -->
5  * @generated NOT
6  */
7 @Override
8 public String getText(Object object) {
9     MessageView messageView = (MessageView) object;
10
11    String label = null;
12    Operation operation = messageView.getSpecifies();
13
14    if (operation != null) {
15        label = RamEcoreUtil.getTextFor(getAdapterFactory(), operation);
16    }
17
18    return label == null || label.length() == 0 ?
19            getString("_UI_MessageView_type") :
20            getString("_UI_MessageView_type") + " " + label;
21 }
```

Listing 5.3: The modified *getText* method of *MessageViewItemProvider* in order to include the specified operation of a message view.

An item provider provides several different features, one of them being the label provider. It can be obtained by calling *adapt* on the *AdapterFactory*. In Line 15 of Listing 5.3 this is delegated to *RamEcoreUtil* which provides this functionality. This functionality is defined in Listing 5.4. The method *stripTypeName* simply strips the type name of the given object.

```
1 public static String getTextFor(AdapterFactory adapterFactory, EObject eObject)
                {
2     IItemLabelProvider itemProvider = (IItemLabelProvider) adapterFactory.adapt(
                eObject, IItemLabelProvider.class);
3
4     return stripTypeName(eObject, itemProvider.getText(eObject));
5 }
```

Listing 5.4: The implementation of *getTextFor* for receiving the textual representation of an object using the adapter class.

In order to use these features in a standalone application like TouchRAM, the generated *edit code* must be available. For each aspect, an adapter factory instance has to be created as shown in Listing 5.5.

```
1 // Create an adapter factory that yields item providers.
2 adapterFactory = new ComposedAdapterFactory(ComposedAdapterFactory.
                 Descriptor.Registry.INSTANCE);
3
4 adapterFactory.addAdapterFactory(new RamItemProviderAdapterFactory());
5 }
```

Listing 5.5: The initialization of the adapter factory for an aspect.

Displaying Text

The component *RamTextComponent* is available to display text in the user interface. This component extends *RamRectangleComponent* and offers text displaying capabilities such as alignment, modification using a keyboard and a cursor. In order to exploit the possibilities that the item providers offer, we introduced a labeled text component *LabeledRamTextComponent*. Instead of setting the text directly, it has a reference to the object that a text should be displayed for. The text is retrieved via the item provider of that object. Furthermore, the component observes the object to receive updates and if notified, updates its text. [24] describes how notifications can be received for customized textual representations that depend on other objects or properties.

Future Enhancements

The capabilities could be further enhanced. Each item provider further has property descriptors for all the properties. For example, the item provider of a *Message View* contains a property descriptor for the specified operation (*specifies*). A property descriptor is responsible for viewing and updating a property. Therefore, it contains a label provider as well. For a better understanding, let us consider that we want to show a class name and be able to update its name. When showing the class name, we want to see the full name including *partial*. However, when editing the class name we just want to see the name without any additional information. For the former, the item provider can be used to get the textual representation. For the latter, the property descriptor can be used to get the actual value of the property and allow to update it. The label provider of the property descriptor can be modified as well in case it has to be different than the item provider of the object. Furthermore, this can be used with commands, meaning that for setting a value a command is executed.

When setting properties that are a reference, the designer is required to select a value. A property descriptor defines the method *getChoiceOfValues*. This method can be overwritten in order to filter out unwanted choices. The designer can then be provided with a choice in the style of a drop-down box.

By exploiting these capabilities, this information is separated from the

user interface component allowing other user interfaces to use the same. This results in a consistent visualization throughout applications. For example, in our case—once our proposal is integrated throughout TouchRAM—models can be modified either by using the TouchRAM user interface or the generic editor provided by EMF. Furthermore, by separating the concerns properly, changing the framework for the GUI only requires to replace GUI related components. In addition, components like *LabeledRamTextComponent* can be defined that retrieve their information by delegating to other objects or adapters. This makes it easier for a developer and results in GUI code that is clearer and easier to maintain, leading to better code quality.

5.6 Streamlined Message View Editing

Currently, message views can be viewed in TouchRAM, but not created or modified. Thus, we give an outlook on how the tool can offer streamlined editing in order to support a modeler for rapid, easy and intuitive creation of message views. UML tools offer various possibilities for the user and often focus mostly on the visual aspects. In RAM, the focus lies on the design of aspects with a high emphasis on conformance to the meta-model in order to support weaving and—later—code generation. Let us consider the use case where a modeler created the structural view and wants to create the message view for a particular operation.

Creating a new message view When creating a new message view, the tool can create the initial interaction, i.e., including the incoming message, the receiving lifeline and—if the operation returns something—a reply message. When creating a new *aspect message view*, the fragment for *original behavior* can be placed on the lifeline such that the modeler just has to move it to the appropriate position.

Adding new messages While it is unclear at this point how exactly messages will be created by the user, the tool can assist the user by displaying possible lifelines where a message can be sent to. The possible lifelines can be retrieved by the properties of the represented classifier of the sending lifeline, local properties of that lifeline or through mappings. For example, when modeling message views for the *Observer* aspect, |*Data* of *ZeroToMany* is mapped to |*Subject*. This means that properties of |*Data* are available as well. An alternative could be to let the user specify the operation that should be called and then creating the lifeline and message for the user.

Selecting an operation The possible signature of a message is defined by the lifeline that receives the call. Therefore, the user can be presented with the operations of the represented classifier, but also operations that become

available through mappings. For example, when defining *startObserving* of |*Observer*, calls to |*Subject* can include operations of |*Data* as it is mapped to |*Subject*.

Consistency checks The tool can perform consistency checks and inform the user or even suggest changes. For example, when specifying that a certain operation is affected by a particular *aspect message view* the tool can check whether the *aspect message views* pointcut matches.

5.7 Related Work

To the best of our knowledge, TouchRAM is currently the only aspect-oriented modeling tool supporting aspect hierarchies. Unfortunately we were not able to verify that claim since the other existing AOM tools that can be used for software design are not readily available for the general public.

Among the tools is the MOTOROLA WEAVR [4], a tool developed and used in an industrial setting. It focuses on MDE with aspect-orientation using an aspect-oriented modeling engine for UML 2.0 state diagrams. WEAVR is an add-in for TELELOGIC TAU. Detailed models with regard to code generation are described using composite-structure architecture diagrams—defining a hierarchical decomposition of a system—and transition-oriented state diagrams—defining detailed behavior. An action language is used for the complete implementation at the model level. A profile for UML 2.0 is used to define aspect models which crosscut certain classes. The designer is supported by a join point visualization engine to visualize and simulate effects of an aspect on a model. Pointcut and advice are expressed by state diagrams. The weaving is then performed right before the code generation.

MATA (*Modeling Aspects Using a Transformation Approach*) [28] tackles join point matching problems by viewing aspect composition as a special case of model transformations. It is an asymmetric approach where base and aspect are composed by specifying a graph rule. Instead of defining transformations on the meta-level—over the abstract syntax of a modeling language—MATA uses graph rules over the concrete syntax of the modeling language. MATA supports the UML meta-model which can be represented as a graph. It is provided as a plug-in for IBMs RATIONAL SOFTWARE MODELER. Class, sequence and state diagrams are supported, however, the authors claim it can be applied to any modeling language with a well-defined meta-model. In contrast to RAM, pointcut and advice are specified in one diagram. Furthermore, no join points exist, as the composition is viewed as a special case of model transformation.

The theme approach THEME/UML [3] was initially developed as a MOF-based extension for UML 1.3. Carton *et al.* [2] then defined a process based on *Model-Driven Architecture* (MDA) combined with the aspect-

oriented Theme/UML approach. Theme/UML is an extension to UML as a profile and can be used with any standard UML tool. It supports class and sequence diagrams. The design process consists of the modeling, composition and transformation phase. The modeling phase consists of modeling base application concerns and crosscutting concerns. Through the definition of composition relationships a designer specifies how they are composed. Models are then automatically composed using transformations. The transformation phase transforms the platform-independent model (PIM) into a platform-specific model (PSM). The PSM can be further re-factored with low-level details by the designer before code is generated. However, the use of UML 2.1 is unsuitable for code generation of sequence diagrams.

Chapter 6

Conclusions And Future Work

This thesis presented the transformation of message views of the *Reusable Aspect Models* approach in theory to message views in practice in order to enable tool support of message views. Therefore, we evaluated the different features of message views that were used in theory and defined a meta-model that suits these requirements. In order to facilitate a compatibility with other UML tools we based our meta-model on the UML meta-model of sequence diagrams and extended it with a higher level of detail. A high emphasis was set on defining a consistent meta-model. This proved to be of importance in order to facilitate the various requirements at later stages. The existing meta-model for structural views had to be extended as well. Furthermore, certain features were found not necessary and omitted and suggestions were made on how message views can be visualized in a clearer way for the designer. In addition, open issues were discussed and possible solutions presented. Defining message views is facilitated for almost all cases, despite the message view related open issues.

The weaving of message views was prior defined in theory as well. We formalized the weaving algorithm and integrated both the weaving of structural view and message views into a general weaving process. This weaving process is prepared to be extended by state view weaving. The weaver allows a designer to weave a complete aspect hierarchy or two directly dependent aspects in a hierarchy together.

Definition of the meta-model and weaver facilitated the integration of message view support into TouchRAM. The meta-model was defined using the Eclipse Modeling Framework and the weaver implemented with the Kermeta framework. Furthermore, the graphical user interface of TouchRAM was extended by message view visualization, allowing a designer to view existent message views. In addition, we proposed several ways on how the architecture of TouchRAM can be improved by using features provided by

EMF. These include the use of adapter classes for retrieving the textual representation of elements, command-based editing for fully automatic undo and redo functionality and editing support using property descriptors of adapter classes. This will lead to a better separation of concerns by decoupling the user interface from such responsibility tasks allowing to use different views. In our case, the editor generated by EMF can already be used to fully create message views that can then be visualized in TouchRAM. Furthermore, the use of such features will allow a developer to focus on the design of the UI and lead to a higher quality of code increasing the maintainability.

As a next step, creation and editing of message views can be added to TouchRAM. In Section 5.6 we discussed some possibilities on how the tool can offer streamlined editing of message views for a modeler. The aim is a streamlined user interface that enables intuitive and fast model editing for agile software design modeling. This allows a modeler to focus mainly on the design of aspect models. Further future work should focus on integrating the editing features provided by EMF into the whole application as outlined in Section 5.5.1.

The support of state views is the last view of the multi-view modeling approach RAM that is not supported yet. Future work should address the integration of state view support. This thesis presented an approach on how this can be achieved by evaluating the view defined in theory and defining a meta-model. Following, the weaving algorithm can be formalized and TouchRAM extended by support for state views. During this thesis we prepared the meta-model and weaver to be extended by state view support in the future.

In Section 3.5.2 we outlined some features of RAM that are currently not supported and need to be addressed in the future. Among these is the reuse of the result of original behavior in the advice. A modeler might want to store the result and use it in the additional behavior or catch and solve an exception. Possible solutions were presented, but it is necessary to evaluate their feasibility. Furthermore, the issue of parametrization is of importance as existing models make use of such features. Currently it is not possible to express this appropriately. More research has to be conducted on this topic. In Section 3.5.2 we presented an overview of those features. Once a solution is found, the meta-model, weaver and user interface have to be adjusted accordingly to support this. Our defined weaving algorithm currently does not require the pointcut, as message views are aware of what aspects they are affected by. Further research should focus on addressing these issues in order to evaluate whether this is necessary.

By addressing these future works and offering TouchRAM to the public we believe that it will increase the likelihood that the Reusable Aspect Models approach and its TouchRAM tool will be adopted and used by users which will provide more feedback through applying it in other software development projects.

Bibliography

[1] Wisam Al Abed and Jörg Kienzle. “Information Hiding and Aspect-Oriented Modeling”. In: *14th Aspect-Oriented Modeling Workshop, Denver, CO, USA*. Oct. 2009, pp. 1–6.

[2] Andrew Carton et al. “Model-Driven Theme/UML”. In: *Transactions on Aspect-Oriented Software Development VI*. Ed. by Shmuel Katz et al. Vol. 5560. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2009, pp. 238–266.

[3] Siobhán Clarke and Elisa Baniassad. *Aspect-Oriented Analysis and Design: The Theme Approach*. Object Technology Series. Addison Wesley, 2005.

[4] Thomas Cottenier, Aswin Van Den Berg, and Tzilla Elrad. “The Motorola WEAVR: Model Weaving in a Large Industrial Context”. In: *Proceedings of the International Conference on Aspect-Oriented Software Development, Industry Track*. AOSD ’06. Bonn, Germany: ACM, Mar. 2006.

[5] Robert Filman et al. *Aspect-Oriented Software Development*. First Edition. Addison-Wesley Professional, 2004.

[6] Robert E. Filman and Daniel P. Friedman. “Aspect-Oriented Programming Is Quantification and Obliviousness”. In: *Aspect-Oriented Software Development*. Ed. by Robert E. Filman et al. Boston: Addison-Wesley, 2005, pp. 21–35.

[7] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA, USA: Addison-Wesley, 1995.

[8] IRISA/INRIA. *Kermeta – Breathe life into your metamodels*. URL: http://www.kermeta.org/. July 2012.

[9] Gregor Kiczales et al. “An Overview of AspectJ”. In: *ECOOP 2001 — Object-Oriented Programming*. Ed. by Jørgen Knudsen. Vol. 2072. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2001, pp. 327–354.

[10] Gregor Kiczales et al. “Aspect-Oriented Programming”. In: *ECOOP’97 — Object-Oriented Programming*. Ed. by Mehmet Akşit and Satoshi Matsuoka. Vol. 1241. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 1997. Chap. 10, pp. 220–242.

[11] Jörg Kienzle, Wisam Al Abed, and Jacques Klein. “Aspect-Oriented Multi-View Modeling”. In: *Proceedings of the 8th International Conference on Aspect-Oriented Software Development*. AOSD ’09. New York, NY, USA: ACM, 2009, pp. 87–98.

[12] Jörg Kienzle, Ekwa Duala-Ekoko, and Samuel Gélineau. “AspectOptima: A Case Study on Aspect Dependencies and Interactions”. In: *Transactions on Aspect-Oriented Software Development V*. Berlin / Heidelberg, Germany: Springer, 2009, pp. 187–234.

[13] Jörg Kienzle et al. “Aspect-Oriented Design with Reusable Aspect Models”. In: *Transactions on Aspect-Oriented Software Development VII*. Ed. by Shmuel Katz, Mira Mezini, and Jörg Kienzle. Vol. 6210. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010, pp. 272–320.

[14] Jacques Klein, Franck Fleurey, and Jean-Marc Jézéquel. “Weaving Multiple Aspects in Sequence Diagrams”. In: *Transactions on Aspect-Oriented Software Development III*. Ed. by Awais Rashid and Mehmet Aksit. Vol. 4620. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2007, pp. 167–199.

[15] Jacques Klein, Loïc Hélouët, and Jean-Marc Jézéquel. “Semantic-based weaving of scenarios”. In: *Proceedings of the 5th international conference on Aspect-oriented software development*. AOSD ’06. New York, NY, USA: ACM, 2006, pp. 27–38.

[16] Max Kramer and Jörg Kienzle. “Mapping Aspect-Oriented Models to Aspect-Oriented Code”. In: *Models in Software Engineering*. Ed. by Juergen Dingel and Arnor Solberg. Vol. 6627. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2011, pp. 125–139.

[17] Gunter Mussbacher and Jörg Kienzle. *Integrating Aspect-Oriented Requirements and Design Models with AoURN and RAM*. Tech. rep. SOCS-TR-2011.2. Montreal, Canada: McGill University, Apr. 2011.

[18] Gunter Mussbacher et al. “Requirements Modeling with the Aspect-oriented User Requirements Notation (AoURN): A Case Study”. In: *Transactions on Aspect-Oriented Software Development VII*. Ed. by Shmuel Katz, Mira Mezini, and Jörg Kienzle. Vol. 6210. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010, pp. 23–68.

[19] Object Management Group. *OMG Meta Object Facility (MOF), Core Specification*. Version 2.4.1. Aug. 2011. URL: http://www.omg.org/spec/MOF/2.4.1/.

[20] Object Management Group. *OMG Unified Modeling Language (OMG UML), Superstructure*. Version 2.4.1. Aug. 2011. URL: http://www.omg.org/spec/UML/2.4.1/.

[21] Andrea Schauerhuber et al. *A Survey on Aspect-Oriented Modeling Approaches*. Tech. rep. Business Informatics Group, Vienna University of Technology, 2006.

[22] Douglas C. Schmidt. "Guest Editor's Introduction: Model-Driven Engineering". In: *IEEE Computer* 39.2 (2006), pp. 25–31.

[23] David Steinberg et al. *EMF: Eclipse Modeling Framework*. 2nd. Boston, MA, USA: Addison-Wesley Professional, 2009.

[24] The Eclipse Foundation. *EMF/Recipes – Recipe: Custom Labels*. URL: http://wiki.eclipse.org/EMF/Recipes#Recipe:_Custom_Labels. July 2012.

[25] The Eclipse Foundation. *Model Development Tools (MDT) – OCL*. URL: http://www.eclipse.org/modeling/mdt/ocl. July 2012.

[26] The Eclipse Foundation. *Model Development Tools (MDT) – UML2*. URL: http://www.eclipse.org/modeling/mdt/uml/. July 2012.

[27] The Fraunhofer Institute for Industrial Engineering IAO. *MT4j – Multitouch For Java*. URL: http://www.mt4j.org. July 2012.

[28] Jon Whittle and Praveen Jayaraman. "MATA: A Tool for Aspect-Oriented Modeling Based on Graph Transformation". In: *Models in Software Engineering*. Ed. by Holger Giese. Vol. 5002. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2008, pp. 16–27.