MAPPING REUSABLE ASPECT MODELS TO ASPECT-ORIENTED CODE

MAX E. KRAMER

Study Thesis

September 2010

Supervisors:

Prof. PhD. Jörg Kienzle
School of Computer Science
Software Engineering Laboratory
McGill University
Montréal, Canada

Prof. Dr. Ralf H. Reussner
Institute for Program Structures and Data Organization
Software Design and Quality
Karlsruhe Institute of Technology
Karlsruhe, Germany

Max E. Kramer: *Mapping Reusable Aspect Models to Aspect-Oriented Code*, Study Thesis, © September 2010

Für Lotte.

ZUSAMMENFASSUNG

Aspektorientierte Modelierung bemüht sich bereits während des Entwurfs von Softwaresystemen Querschnittsbelangen auf einer abstrakten Ebene zu begegnen, während aspektorientierte Programmiersprachen wie AspectJ es möglich machen detaillierte Interaktionen und Aspekte auf einem niedrigen Abstraktionslevel zu definieren. Die Fragestellung wie modellierte Aspekte in wiederverwendbaren Code überführt werden können, sodass Querschnittsbelange erhalten bleiben und auf beiden Ebenen zurückverfolgt werden können, wurde bisher jedoch nicht abschließend beantwortet.

In dieser Studienarbeit stellen wir eine Abbildung von Reusable Aspect Models auf AspectJ vor, welche versucht diese Probleme anzugehen, indem sie Quellcode erzeugt der die Struktur und Prinzipien des Modells erhält und eine zurückhaltende Schnittstelle bereitstellt. Anhand von Beispielen der AspectOPTIMA Fallstudie, die ein Transaktions-Framework realisiert, beschreiben wir im Detail wie Modellelemente mit Hilfe von Konstrukten einer aspektorientierten Programmiersprache implementiert werden können. Schlüsselelemente dieser Abbildung sind die unauffällige Wiederverwendung von Java Bibliotheken, die flexible Bindung von Parametern mit Hilfe von Annotationen, und die transparente Unterstützung von Aspekthierarchien und konfigurierbaren Produktlinien.

Der Quellcode den wir im Rahmen unserer Fallstudie erhielten veranschaulicht unsere Abbildungsvorschrift nicht nur anhand eines Projektes vernünftiger Größe, sondern er demonstriert auch ihre Abdeckung, da wir mehr als 84% der gesamten Implementierung generieren konnten.

ABSTRACT

Aspect-Oriented Modeling strives to address cross-cutting concerns at an abstract level during the design of software systems whereas Aspect-Oriented Programming tools like AspectJ make it possible to define detailed interactions and aspects on a low level. But, the problem how modeled aspects can be transformed in reusable code that allows designers to maintain and trace cross-cutting concerns on both levels has not been solved so far.

In this thesis we present a mapping from Reusable Aspect Models to AspectJ that tries to address these problems by generating source code that maintains the structure and principles of the model and provides a noninvasive interface. Using examples from the AspectOPTIMA case study that realizes a transaction framework, we describe in detail how modeling artifacts can be implemented with constructs of an aspect-oriented programming language. Key elements of this mapping are the unobtrusive reuse of Java libraries, flexible parameter binding through annotations, and transparent support of aspect hierarchies and product lines.

The code that we obtained for the case study does not only exemplify our mapping with a reasonable-sized project, it also demonstrates its comprehensiveness as we generated more than 84% of the complete implementation.

Genial muss einfach sein.

Max Urban

ACKNOWLEDGMENTS

I am grateful to Prof. Kienzle for all the time and energy he spent discussing and improving my ideas in Montréal. At my home university in Karlsruhe the immediate confidence of Prof. Reussner made this transatlantic project possible and worth the effort. Many thanks to both of you.

I am thankful to my friend and colleague Benjamin Niedermann for fighting his way through all the models, code snippets and explanations in this thesis in order to spot mistakes and to improve the understandability.

CONTENTS

1 INTRODUCTION

Throughout the last years, various aspect-oriented modeling approaches that try to address cross-cutting concerns have been developed. Most of these approaches make use of sophisticated model weaving techniques in order to transform the aspect-oriented model into a model without aspects. These woven models are typically object-oriented so that existing code generation techniques can be applied to them.

Aspectual information and resulting benefits are lost if object-oriented code is generated from woven models. In order to maintain traceability and consistency during code generation, efforts have been made to target aspect-oriented languages. Recent experiments have shown that targeting aspect-oriented platforms with an "all-aspectual approach results in smaller, less complex and more modular implementation". [11].

In this thesis we present a mapping from Reusable Aspect Models (RAM) to AspectJ that realizes such an all-aspectual approach. We describe in detail how elements of Reusable Aspect Models can be mapped to constructs of an aspect-oriented programming language. In order to develop and proof our conceptual mapping we applied it to the transaction system AspectOPTIMA [19], a case study for aspect-orientation. To increase the flexibility and comprehensiveness of our approach, we added support for Open Multithreaded Transactions to the original case study. Using the extended case study we are able to explain our mapping with various concrete examples.

We applied our mapping manually to our extended case study and showed that we were able to derive 88% of the implementation from the model. The code that we obtained for AspectOPTIMA proves that Reusable Aspect Models can yield almost complete implementations that maintain the logical structure of the model. To back up our claim that this is a general property of our mapping it has to be validated on other case studies.

The remainder of this thesis is structured as followed: Chapter 2 provides background information on Aspect-Oriented Modeling, Aspect-Oriented Programming, Reusable Aspect Models, AspectJ, the AspectOPTIMA case study, and Open Multithreaded Transactions. In Chapter 3 and Chapter 4 we present the updated models for AspectOPTIMA and its extension in order to give the reader the context that is necessary for the examples that we use to describe our mapping in Chapter 5. The in-depth explanation of our mapping forms the central part of this thesis.

It includes sections for each view and for special circumstances such as product line variations and multiple reuse with different bindings. Chapter 6 presents the interface for our AspectOPTIMA implementation and explains how it can be used along two small examples. Related work is discussed in Chapter 7. The last chapter concludes our thesis and presents our thoughts on future work.

2 BACKGROUND

Aspect-Orientation is a development paradigm that addresses concerns that cross-cut multiple modules or subsytems. It allows developers to clearly express cross-cutting concerns that would otherwise be scattered over multiple modules or subsystems. Aspect-Orientation integrates smoothly with other paradigms such as Object-Orientation, and it can be applied at any phase during the software development process.

2.1 ASPECT-ORIENTED MODELING

Entangling cross-cutting concerns in the model.

Aspect-Oriented Modeling (AOM) allows a system modeler to identify, represent, separate and compose cross-cutting properties while designing or analyzing a software system.

During the last decade Aspect-Oriented Programming [13] techniques have matured to powerful and sophisticated tools that allow developers to express cross-cutting concerns of large and complex systems separately [22]. Understanding the technical details of these mechanisms can be hard. Developers often need to separate the structure and logic of the analyzed concerns mentally from the syntax and structure of the code. This process can be error-prone and may introduce accidental complexity, thus leading to a shift of complexity instead of reducing it.

Aspect-Oriented Modeling approaches give a modeler the possibility to reason about cross-cutting concerns individually on a high level of abstraction without regards to the technical details of their implementation [17]. By using visual descriptions, AOM provides the means to transform cross-cutting concerns into interacting concerns. This allows a modeler to focus on their semantics, composition and conflicts.

One field of research in Aspect-Oriented Modeling tries to find appropriate abstractions that express these concerns, their interactions, and their relationships to non-cross-cutting concerns. When modeling notations for AOM are developed two possible approaches can be followed. Some researchers decide to develop extensions to existing modeling notations that introduce constructs that are similar to those of aspect-oriented languages such as pointcuts and advice [2]. Others develop completely new notations for AOM.

A part of the Aspect-Oriented Modeling community is concerned with model transformation and composition techniques in order to develop model weavers that resolve and flatten aspect-

oriented models to conventional models. Such model-weaving techniques provide consistency checks and they can ease the modeling process by offering a view of the resulting system. Furthermore, they give modelers the freedom to realize aspect-oriented systems on platforms without support for aspect-oriented language constructs.

Aspect-oriented modeling approaches cover the whole software development cycle and span from requirements elicitation (e.g AoURN [23]) to detailed design models. Schauerhuber et al. [24] present an in-depth evaluation of eight AOM approaches and a conceptual reference model for Aspect-Oriented Modeling.

2.2 aspect-oriented programming

Implementing cross-cutting functionality once.

Aspect-Oriented Programming (AOP) [13] is a programming paradigm that aims at increasing the modularity of software by allowing developers to separate cross-cutting concerns from business logic. AOP should not be seen as a replacement for other programming paradigms such as Object-Oriented Programming or Procedural Programming but it can extend both.

2.2.1 *Motivation*

Most software systems contain functionality that cannot be separated into a single module or subsystem as it affects various other parts of the system. Cohesive areas of such functionality may be expressed logically as a single concern but they result in the implementation of scattered or tangled code if programming languages without support for aspect-orientation are used. AOP tries to provide concepts to express such cross-cutting concerns at a unique location. It lets a programmer explicitly define the cross-cutting nature of concerns and gives him the possibility to specify the interaction with other parts of the system with special language constructs such as join points, pointcuts and advice.

2.2.2 *Basic Concepts*

In order to encapsulate cross-cutting concerns in one place, most AOP techniques offer at least three types of aspect-oriented concepts: *join points* specify certain points in the execution of a program that can be intercepted by a programmer during the compilation or runtime of the program. An example for a join point that is exposed by most AOP platforms is the execution of a method. *Pointcuts* can be used by a developer in order to specify which join points should be selected. For example all executions of methods that start with *set* could be captured by a single pointcut.

Advice make use of such pointcuts and define how the structure or behavior of a program should be modified if a join point that matches the used pointcut is detected. A possible advice could be a realization of the Observer pattern that calls an update function after every execution of a setter method.

A structural unit that encapsulates the three fundamental concepts that we presented is called an aspect, and it can be applied to other code in a process that is called weaving. Asymmetric Aspect-Oriented Programming approaches distinguish between aspects and a code base without aspectual information whereas symmetric techniques have the philosophy that every artifact belongs to an aspect.

2.3 REUSABLE ASPECT MODELS

2.3.1 *Overview*

An aspect-oriented multi-view modeling technique.

REUSABLE ASPECT MODELS (RAM) [20] is an Aspect-Oriented Modeling approach that supports detailed design using three modeling notations [18]. With a visual notation that is similar to the UNIFIED MODELING LANGUAGE (UML) and additional support for aspect-oriented techniques it allows developers to define symmetric models with three different view types. This multi-view modeling makes it possible to represent every individual subconcern with its most appropriate modeling notation and leads to a higher expressivity than single-view approaches.

Each stand-alone aspect defines the structure and behavior of a package of classes. It has to contain a structural view that is very similar to a UML class diagram. The invocation protocol for classes of the structural view can optionally be described in state views, which are similar to UML state diagrams. Finally, the behavior of methods that were mentioned in the structural view can be detailed in message views. They describe message sequences in a notation that is similar to UML sequence diagrams. Every of RAM's view types supports aspect-oriented features that are not part of UML such as parameters, pointcuts and advice.

The RAM model weaver has composing capabilities for every diagram type and offers additional consistency checks. Dependencies are automatically resolved and for each sequence diagram the corresponding state views are taken into account in order to verify that no message violates the invocation protocol. Even if this adds additional complexity to the weaving process, it reduces the likelihood of modeling errors. The use of a method that is not defined in the structural view or the invocation of a method that is forbidden in the current system state is already detected during the modeling process.

Unfortunately, we cannot discuss RAM in detail but we want to explain its main concepts and refer the interested reader to the corresponding publications [1] [18] [20] by Kienzle et al.

2.3.2 *Features*

STRUCTURAL VIEW The central part of a REUSABLE ASPECT MODEL is the structural view that presents all involved classes with their attributes, methods, and associations similar to class diagrams in UML. A speciality of RAM is that classes can either be complete or incomplete, which means that they need to be be instantiated prior to use. Such mandatory instantiation parameters are marked by prepending the vertical bar character | to their name[1]. They can be instantiated in reusing aspects with instantiation directives of the form *ParamName* → *InstantiatingName*. Existing classes and states can be bound with binding directives by writing *NameToBind* → *BoundName*. Note that RAM makes it possible to bind every class and method regardless of the fact whether they are marked as parameters or not.

STATE VIEW In order to allow the modeler to verify whether modeled method calls conform to a certain protocol he has the ability to define such an invocation protocol using state views. For every class that is mentioned in the structural view it is possible to define states and transitions that correspond to method calls using a notation that is similar to UML state diagram. The main difference to UML is that it is possible to modify states and transitions that were defined in other aspects by specifing them in a pointcut partition of the view and defining a replacement in an advice partition. In Section 5.2.2 we explain why we do not yet support state views in our mapping to ASPECTJ .

MESSAGE VIEW The concrete behavior of a system can be modeled in RAM using message views that make use of parts of the notation of UML sequence diagrams. It is possible to detail the behavior of methods that were newly defined in the structural view or to advice existing methods or method parameters. To this end an arbitrary sequence of messages can be modeled in the pointcut part of the message view and the behavior that should replace every matching message sequence can be defined in an advice part. That means that even the functionality of methods that are yet unknown to the modeler but bound to method parameters in reusing aspects can be modified with this technique. Existing behavior can be removed or replaced and new behavior can be inserted after, before, or around the original behavior. For this purpose it is possible to represent the unmodified behavior

1 Not to be confused with an uppercase i: *I*

of a method in its form before the current advice applied with a box that contains the wildcard character *. This wildcard box can then be reused in the advice part in order to define how new messages have to be inserted.

CONFLICT RESOLUTION ASPECTS In order to automatically detect and resolve conflicts that arise from the combination of different aspects RAM offers the possibility to define special conflict resolution aspects. These aspects have all features of ordinary aspects, but they cannot be instantiated. Instead, they contain *interference criteria* that define in which situations the content of the conflict resolution aspects has to be applied.

PRODUCT LINE CONFIGURATION The RAM approach can be used to model software product lines thanks to its support for optional and alternative aspect dependencies. The dependencies between aspect models are captured using feature diagrams and mentioned in the depending aspects. Within every aspect it is possible to define binding directives, instantiation directives, and message views exclusively for a certain configuration.

2.4 ASPECTJ

An aspect-oriented extension to the JAVA language.

AspectJ [14] is a popular aspect-oriented extension to the Java language that can be used in every Java development environment as it produces pure Java bytecode. A discussion of the detailed features of AspectJ is beyond the scope of this thesis, but we will give a quick introduction into its main concepts.

The available join points in ASPECTJ span from the execution or call of methods, over field access to the initialization of objects or classes. Pointcuts can be constructed using wildcard patterns, fully qualified names or annotation based patterns and they can be connected with boolean expressions. Advanced pointcuts allow programmers to detect the application of specific advice, to intercept and handle executions, and to define dynamic control flow conditions. Advice can either be dynamic or static. Dynamic advice can be applied before, after, or around pointcuts. Static advice can be used to introduce new member variables or methods into selected classes or interfaces. All these constructs are encapsulated within so called aspects that represent a structural unit similar to classes. For a detailed introduction into the AspectJ language Ramnivas Laddad's book "AspectJ in Action" [22] can serve as a valuable and comprehensive starting point.

2.5 ASPECTOPTIMA

A case study for Aspect-Oriented Software Development.

AspectOPTIMA [19] is a transaction framework that was designed as a case study for Aspect-Oriented Modeling and Aspect-Oriented Programming. We used the existing RAM model of AspectOPTIMA [18] in order to develop and test our mapping to AspectJ. During the work on our mapping we updated the models in order to leverage the full power of our mapping. We present the new models for AspectOPTIMA independent from our mapping in Chapter 3.

During the course of his master's thesis *Aspectual Decomposition of Transactions* [4] Bölükbası implemented AspectOPTIMA using AspectJ. At that time the model of AspectOPTIMA handled aspects of threads, contexts and transactions separately and included features like persistence that are not part of the actual model. In the following years a new model of AspectOPTIMA using Reusable Aspect Models [18] had been developed and AspectJ was subject to many changes and improvements. For these reasons the implementation of AspectOPTIMA that derived from the application of our mapping differs greatly from the first implementation. We were able to use recent features of AspectJ such as support for Generics and Annotations and included only very little manual refinements.

In order to obtain a wide range of different modeling circumstances we developed an extension to the AspectOPTIMA model that supports Open Multithreaded Transactions (OMTT). The principles of OMTT are explained in the following section and the corresponding Reusable Aspect Models are presented in detail in Chapter 4.

2.6 OPEN MULTITHREADED TRANSACTIONS

A transaction strategy for concurrent cooperation.

Open Multithreaded Transactions [16] is a transaction model that addresses concurrency problems of conservative transactional systems by providing features for controlled cooperation between threads.

2.6.1 *Motivation*

Sequential transaction models impede concurrency features of modern systems by restricting transactions to single threads in order to achieve isolation. Cooperation between threads is not foreseen and unregulated concurrency in transactional systems that were not designed for this purpose can lead to inconsistent states.

Transactional models that enable concurrency while still guaranteeing the ACID [8] properties are of great benefit to distributed

systems and modern multi-core platforms. Open Multithreaded Transactions were designed to provide required guarantees for transactional properties while leaving as much freedom as possible in order to support cooperation between concurrent threads.

2.6.2 *Features*

We present the main features of Open Multithreaded Transactions in order to give the reader a general idea of the requirements for a model that realizes this transaction model.

Opening - If an arbitrary thread starts a transaction, it will be the first joined participant of this open transaction.

Nesting - If a participant starts a new transaction, this transaction will be a nested child of the previous transaction of this participant. Sibling transactions that were created by different participants execute concurrently.

Joining - In order to join a transaction, a thread must either participate in a parental transaction or participate in no transaction at all, and the transaction to be joined must be open. A thread can only participate in one sibling transaction at a time.

Spawning - Participants of transactions can spawn new participants that will automatically become spawned participants of the innermost transaction of their creator.

Closing - Every participant of a transaction can close it at any time such that no new threads can join the transaction anymore. New threads can be spawned inside closed transactions and every transaction closes automatically once all participants voted on an outcome.

Isolation - Participants access transactional objects in complete isolation from other transactions even if these transactions are descendants of the current transaction.

Voting - Participants can vote on the outcome of the transaction that they are participating in. Threads that "disappear" without voting are implicitly voting *abort*.

Commiting - A transaction reveals changes that were made on its behalf to transactional objects to the outside world if and only if all participants voted *commit*.

Terminating - Spawned participants terminate immediately after voting on the outcome of a transaction.

Leaving - Joined participants cannot leave a transaction before its outcome has been decided. Therefore, they are blocked until the last participant voted *commit* or until a participant voted *abort*.

Figure 1 shows an example of two nested Open Multithreaded Transactions with four joined participants A, B, C and D and two spawned participants B′ and C′ that demonstrates the blocking behavior.

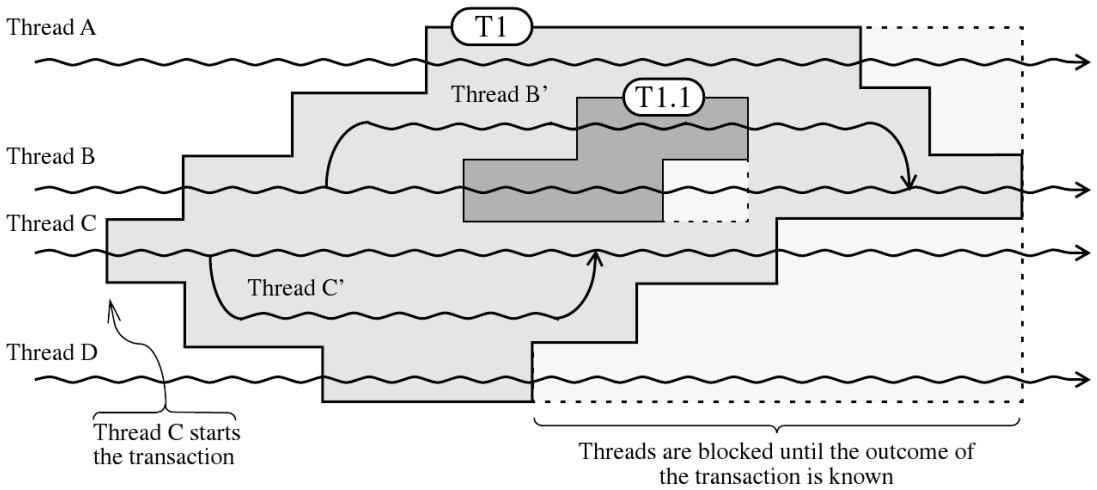

Figure 1: An Example of two nested Open Multithreaded Transactions involving four existing and two newly spawned threads (from Kienzle [15]).

3

ASPECTOPTIMA: A TRANSACTION FRAMEWORK

An updated modeling case study for our mapping.

As a case study for Aspect-Oriented Modeling the AspectOPTIMA transaction framework helped us to develop our general purpose mapping from Reusable Aspect Models to AspectJ that we present in Chapter 5. We will explain our mapping using examples from this framework that we introduced in Section 2.5. As some of the functionality of AspectOPTIMA and the interdependencies between its aspects are quite complex, we hope to ease the understanding of the later examples by presenting the complete model in advance. However, in order to understand our mapping examples it is not necessary to know all aspects in advance. We suggest the reader to use our model description as a work of reference whenever he is in doubt about the purpose or meaning of a model that is used to describe our mapping.

While we developed our mapping we modified the original case study in order to make use of the full power and flexibility of our mapping. Most of these modifications were made in order to follow a single modeling style and in order to reduce slight ambiguities that complicate code generation.

3.1 ASPECTS

AspectOPTIMA consists of 17 aspects and 5 conflict resolution aspects. We modeled all conflict resolution aspects and present them in the next section. As time constraints did not permit us two model the two aspects *OptimisticValidation* and *SemanticClassified* we present all remaining 15 conventional aspects in this section. In order to present aspects before they appear in reusing aspects, we start with the lowest level of reuse and proceed with aspects that directly depend on already presented aspects.

In order to give the reader an overview over the complete framework, we present a feature diagramof AspectOPTIMA in Figure 2. It includes all aspects, conflict resolutions, composition rules and variation points of the framework.

3.1.1 *AccessClassified*

Linking access information to methods.

The *AccessClassified* aspect that we present in Figure 3 introduces a method *getAccessKind* that returns the access kind for a given method. This method is introduced into every class that is bound to the mandatory instantiation parameter |*AccessClassified*.

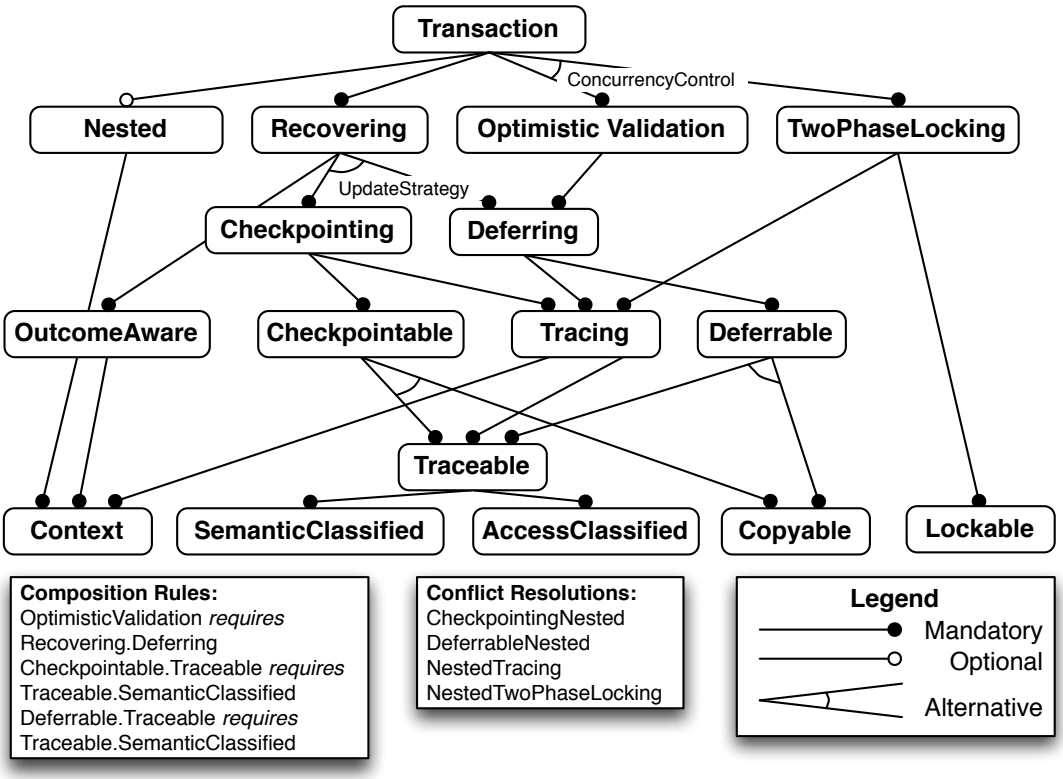

Figure 2: A feature diagram of the AspectOPTIMA framework showing all aspects of common single-threaded transactions.

The type *AccessKind* is specified as an enumeration that contains the enumeration literals *Read*, *Write*, and *Update*.

Methods for which the access kind shall be retrievable need to bind the mandatory instantiation parameter $!m$ and have to provide the access kind as parameter of the parameter. The behavior of the method *getAccessKind* is not modeled as we consider the way how this information is stored and retrieved an implementation detail.

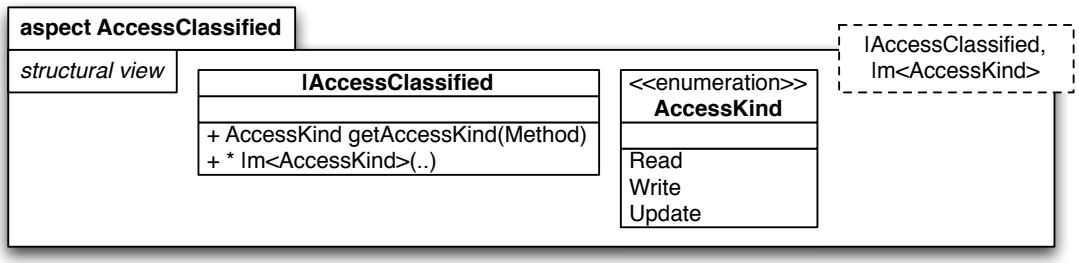

Figure 3: The *AccessClassified* aspect enables the retrieval of the access kind for each method that is given as a parameter.

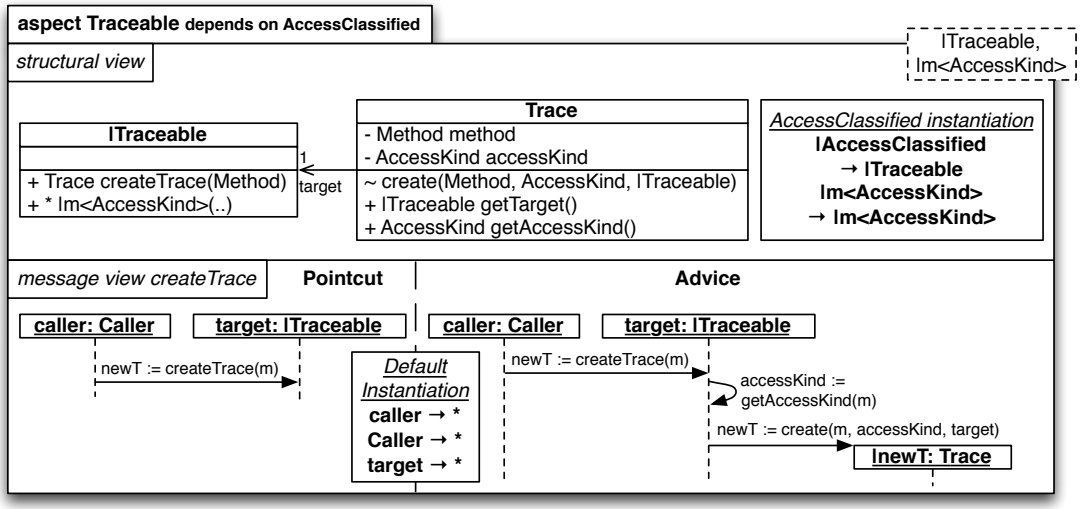

Figure 4: The *Traceable* aspect provides the functionality to create traces of method invocations on objects.

3.1.2 *Traceable*

Infrastructure for method invocation traces.

The *Traceable* aspect shown in Figure 4 relies on the *AccessClassified* aspect in order to provide the possibility to create traces for calls to access classified methods. The suffix "-able" indicates that this aspect is only providing the infrastructure for a tracing functionality but not the automatic creation of such traces.

With an instantiation directive in the structural view we specify that the newly defined mandatory instantiation parameters *ITraceable* and *Im<AccessKind>* are bound to the parameters *IAccessClassified* and *Im<AccessKind>* of the reused *AccessClassified* aspect. As a result every construct that is bound to the new parameter will automatically be bound to the linked parameter of the reused aspect in order to make use of its functionality.

The message view for the method *createTrace* defines this new method but uses the general pointcut-advice syntax. The pointcut defines the message sequence to which the advice should be applied and contains a single call to the method *createTrace*. Whenever this message is found the advice that is defined on the right side of the message view is applied. It shows that we retrieve the access information for a given method by calling *getAccessKind*. Afterwards, we create a new trace that contains this information together with a reference to the method and the target object. Note that methods that are named *create* and that return the enclosing type represent constructors in Reusable Aspect Models . Without such a constructor a class is incomplete and needs to be completed with an instantiation or binding directive or it has to be marked as a mandatory instantiation parameter by prepending the vertical bar character | to its name.

3.1.3 *Context*

Introducing contexts and their participants.

In the *Context* aspect the central notion of contexts and context participants is introduced as presented in Figure 5. As every participant is associated to at most one context and every context to at most one participant, we only need to consider cases in which a participant creates and immediately enters a new context.

The behavioral logic for this case is presented in the message views of the methods *createAndEnterContext* and *enterContext*. The first of these methods calls the second one which in turn modifies the corresponding association properties. When a context is left by invoking *leaveContext* the association information is discarded and the marker method *contextCompleted* is called.

In addition to these methods with message views we include infrastructural methods for the modification and use of the association information into the structural view. Furthermore, we mention a static method *getCurrent* that returns a *Participant*. It is used in other aspects to obtain the current participant when arbitrary method calls that have no direct access to context information are advised.

3.1.4 *Tracing*

Automatic storage of access information for method calls.

The *Tracing* aspect shown in Figure 6 adds an automatic tracing functionality to the infrastructure of the *Context* aspect based on the functionality provided by the *Traceable* aspect. It advises all calls to methods that have been bound to the mandatory instantiation parameter |*m* of a |*Traced* object by means of a message view *traceMethod*. After retrieving the current participant, obtaining its associated context, and creating the corresponding tracing information it stores this information using the obtained context. A rectangular box that contains the * character as a wildcard is used in the pointcut part in order to represent the original method behavior of the advised method. It is also used in the advice part in order to define that all the tracing information is created before the original behavior of the advised method. The methods of reused aspects that have to be woven into the message view are mentioned in the message view label using the keywords *affected by* in order to give the reader a fast overview of reused methods.

As Reusable Aspect Models have an automatic information hiding feature we have to reexpose all reused methods that we want to make accessible for reusing aspects. Therefore we mention the methods *getCurrent*, *getContext*, *createAndEnterContext*, and *leaveContext* of the *Context* aspect in the |*TracingParticipant* class with a public access modifier within the structural view. Automatic information hiding is discussed in detail in Section 5.6.3.

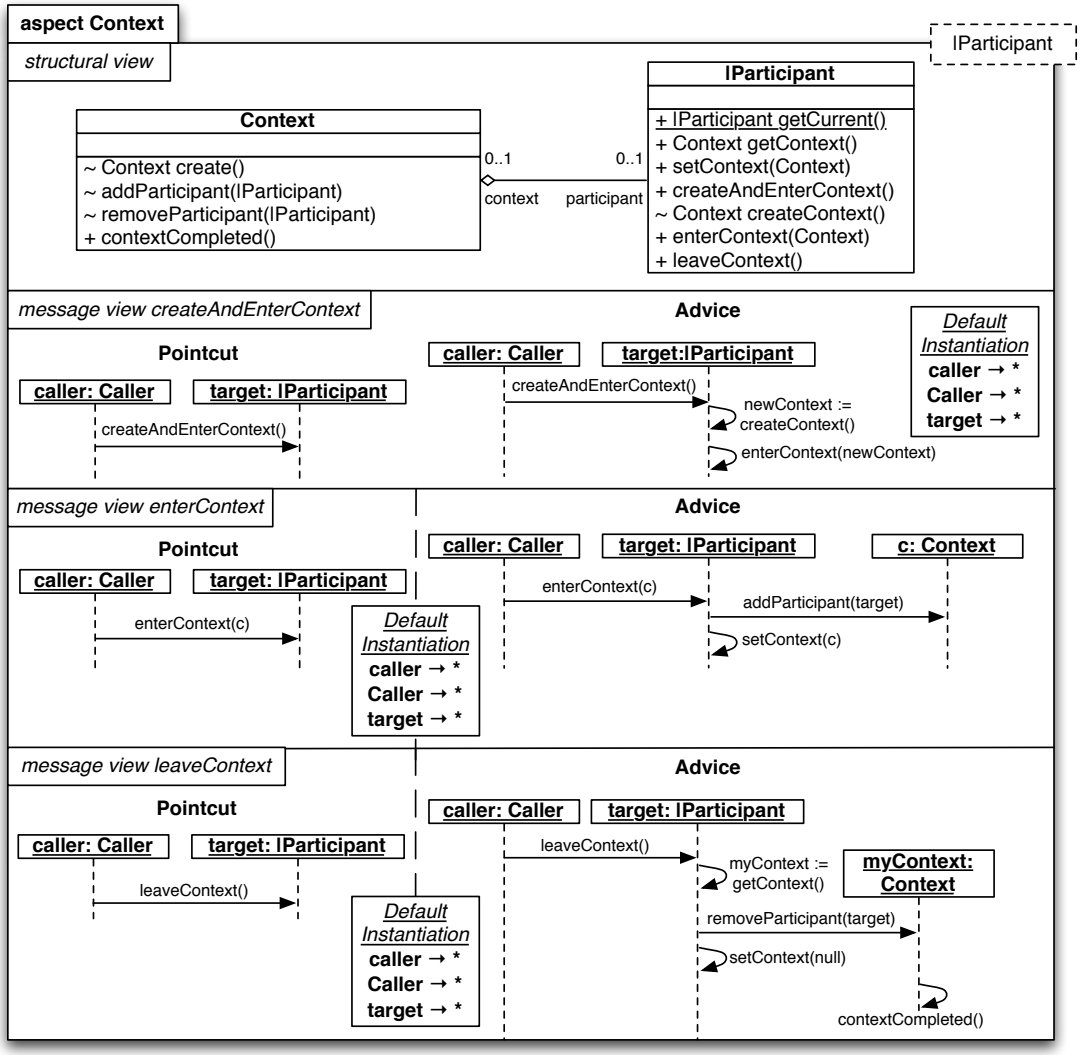

Figure 5: The *Context* aspect adds and administrates an association between participants and contexts.

The storage of tracing information is detailed in the structural view as a set of accessed |*Traced* objects and a list of *Traces* are associated to the class *TracingContext*. How these data structures are used is shown in message views. The behavior of *addTrace* is to obtain the target from the passed *Trace*, to insert a reference to this target into the set of accessed objects, and to finally add the trace itself to the list of traces.

Whether or not a |*Traced* object was already accessed can be answered by the method *wasAccessed*. It simply returns the result of a call to the function *contains* on the set of accessed objects. Already existing tracing information can be removed for selected objects with a call to *removeTraces*. The message view of this method demonstrates that our decision to store the set of accessed objects separately makes the removal of traces more complicated as we need to obtain the corresponding traces iteratively. But aspects that reuse the *Tracing* aspect are relieved from this additional complexity as they can directly reuse the method *removeTraces* by providing a set of objects without any knowledge of the corresponding *Traces*.

3.1.5 *Copyable*

Copying and replacing the state of objects.

The model for the *Copyable* aspect as presented in Figure 7 is concise but its functionality for cloning and replacing objects is crucial for many other aspects in the framework.

Two public methods *copy* and *replaceStateWith* are declared on the mandatory instantiation parameter class |*Copyable*, but they are not detailed in message views as we consider their behavior an implementation detail.

3.1.6 *Deferrable*

Creating and administrating multiple versions of objects.

The *Deferrable* aspect shown in Figure 8 makes use of the *Copyable* functionality in order to provide the possibility to obtain different versions of an object. In the structural view we associate each |*Deferrable* object with a map that links contexts to the corresponding versions of the |*Deferrable* object. We also link each |*Deferrable* object to exactly one |*Deferrable* object that we call its *original* as it refers to the object of which the object in consideration is a version of.

The method *getVersion* includes functionality to automatically create a new version if no version exists yet. First, we check whether the original association refers to the same object or not. If the object and its original are different, we are already inspecting a version and therefore we can return it immediately. But, if the object and its original are identical, we need to create a new version, and store a reference to it in the *contextToVersionMap*.

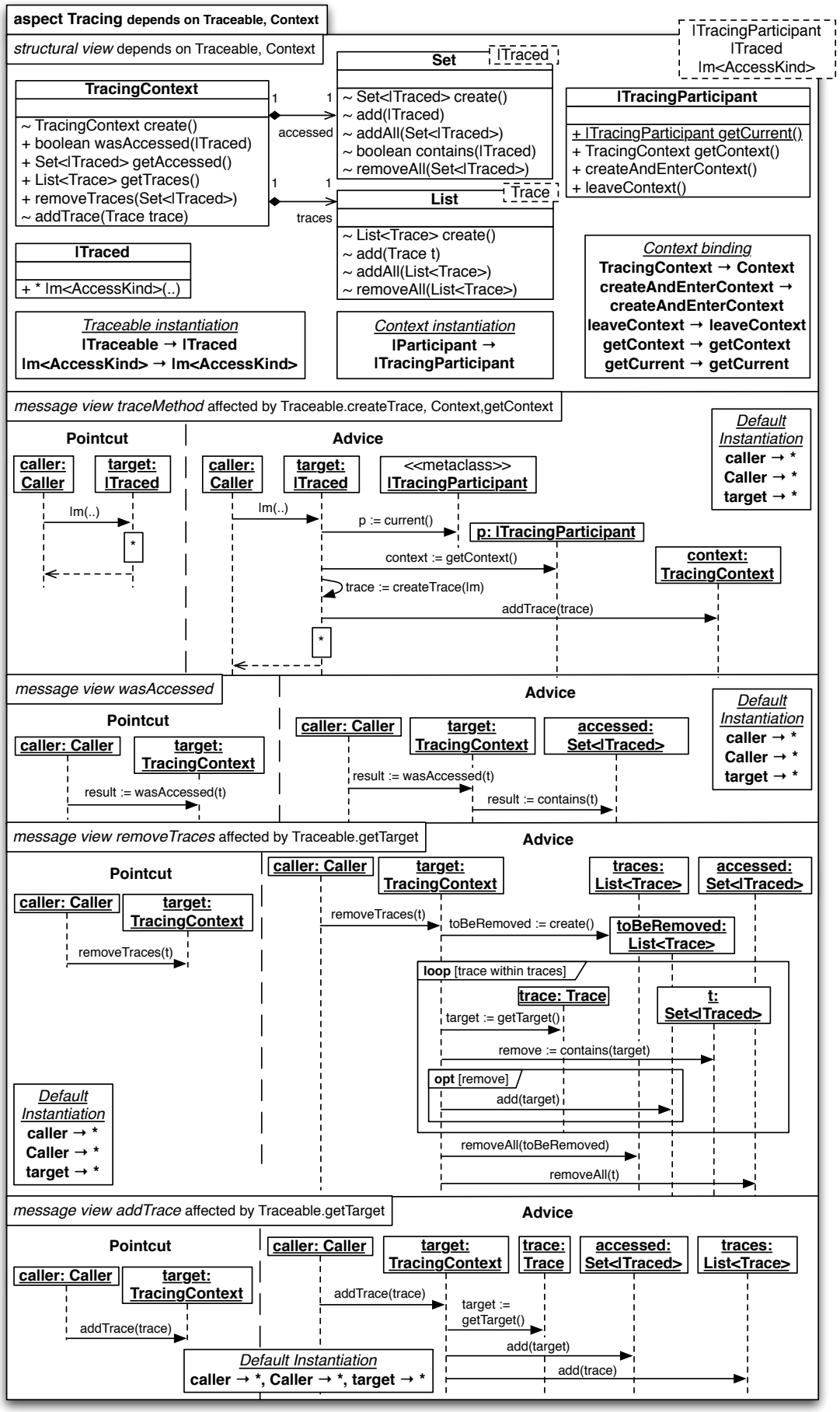

Figure 6: The *Tracing* aspect uses the *Traceable* and *Context* aspects to automatically create traces and it provides access to them.

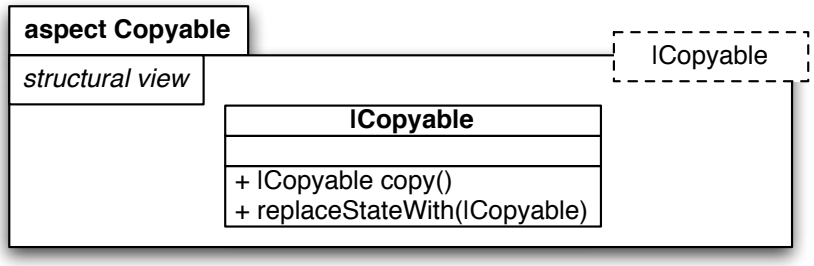

Figure 7: The *Copyable* aspect makes it possible to clone objects and to replace their state.

The message view for *updateOriginal* demonstrates how we use the *original* association and the map from contexts to versions in order to replace the state of the original object with the state of the obtained version. It also shows that a version is unnecessary once its state has been used for an update of the original. We simply delete a version after such an update.

Removing versions is very simple as we can see in the message view *deleteVersion*. All that has to be done is to delete the version from the associated map with a call to the *remove* method.

3.1.7 *Deferring*

Defering method calls by performing them on a version of the target.

AspectOPTIMA's *Deferring* aspect as presented in Figure 9 automatically defers calls to transactional objects and performs them on behalf of the context participant on a separate version that is provided by *Deferrable*. It also provides the method *performUpdate* that is detailed in a message view and that makes use of the access information provided by the *Tracing* aspect: for every object that was accessed on behalf of the context the method *updateOriginal* is called.

Every call to a method that is bound to the mandatory instantiation parameter |*m* of a |*Deferred* object is advised using the behavior of the message view *deferMethod*. In this view we retrieve the original object from the object in consideration in order to decide on which object the original behavior should be executed. If the original differs from the target, the unchanged behavior is executed on the target. Otherwise, the current participant is obtained, its context is retrieved, the version for this context is obtained, and the original behavior is executed on it.

3.1.8 *Checkpointable*

Saving and restoring different states of objects.

The *Checkpointable* aspect as modeled in Figure 10 makes it possible to establish and discard checkpoints of the state of transactional objects in order to undo operations in case of an abort.

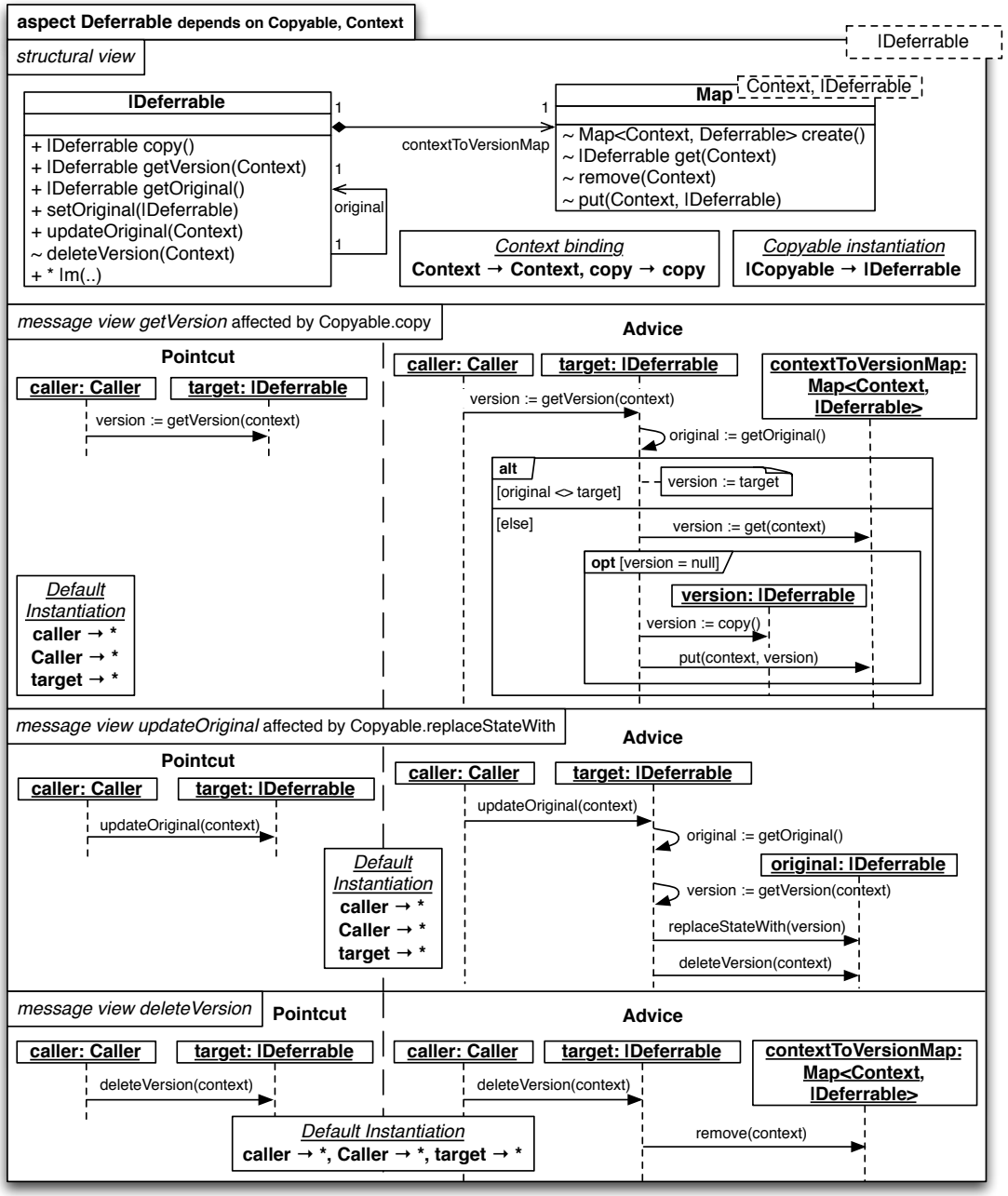

Figure 8: The *Deferrable* aspect provides facilities to defer operations by using *Copyable* in order to create and maintain different versions of an object.

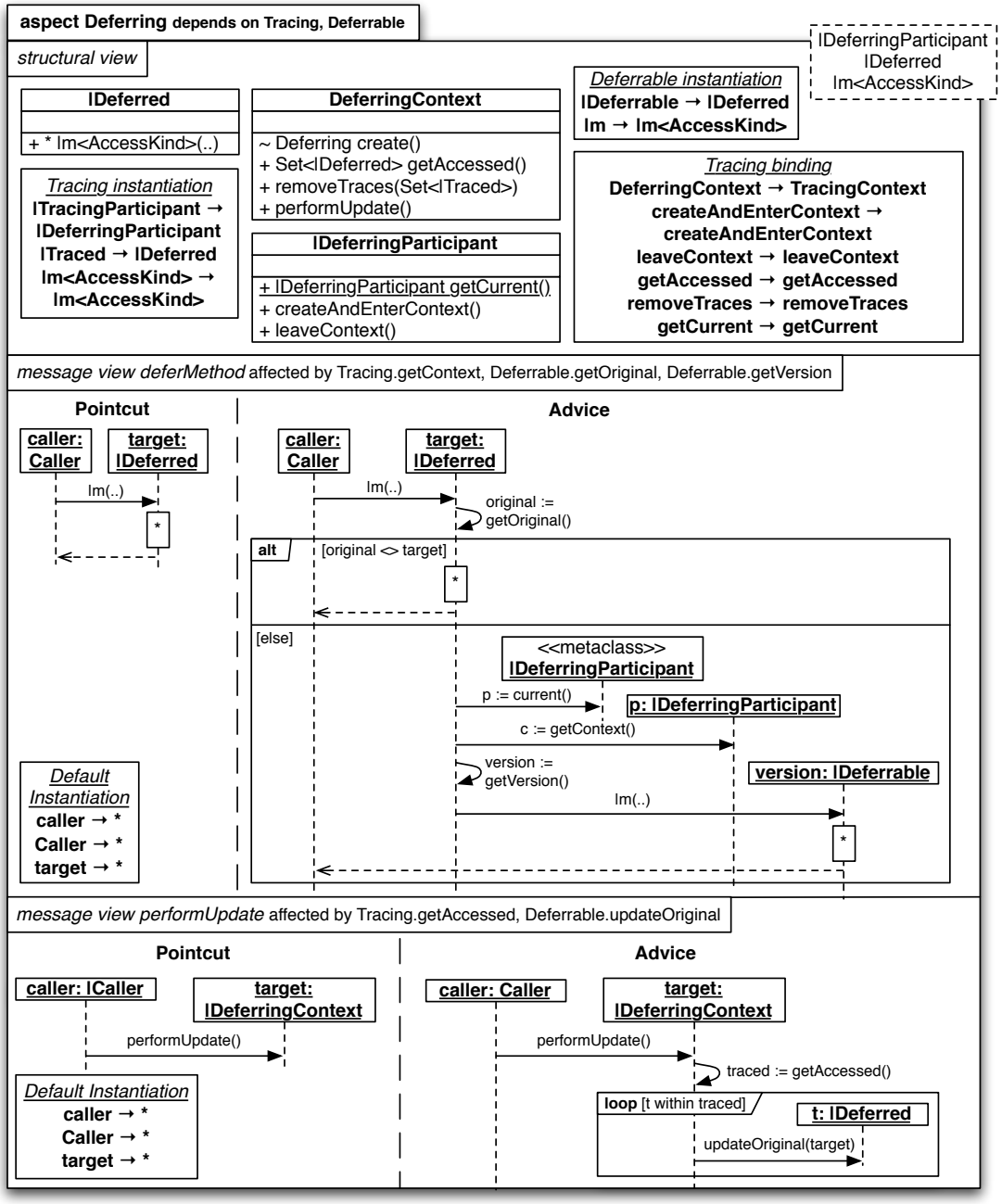

Figure 9: The *Deferring* aspect reuses *Deferrable* and *Tracing* in order to automatically defer operations based on the access history.

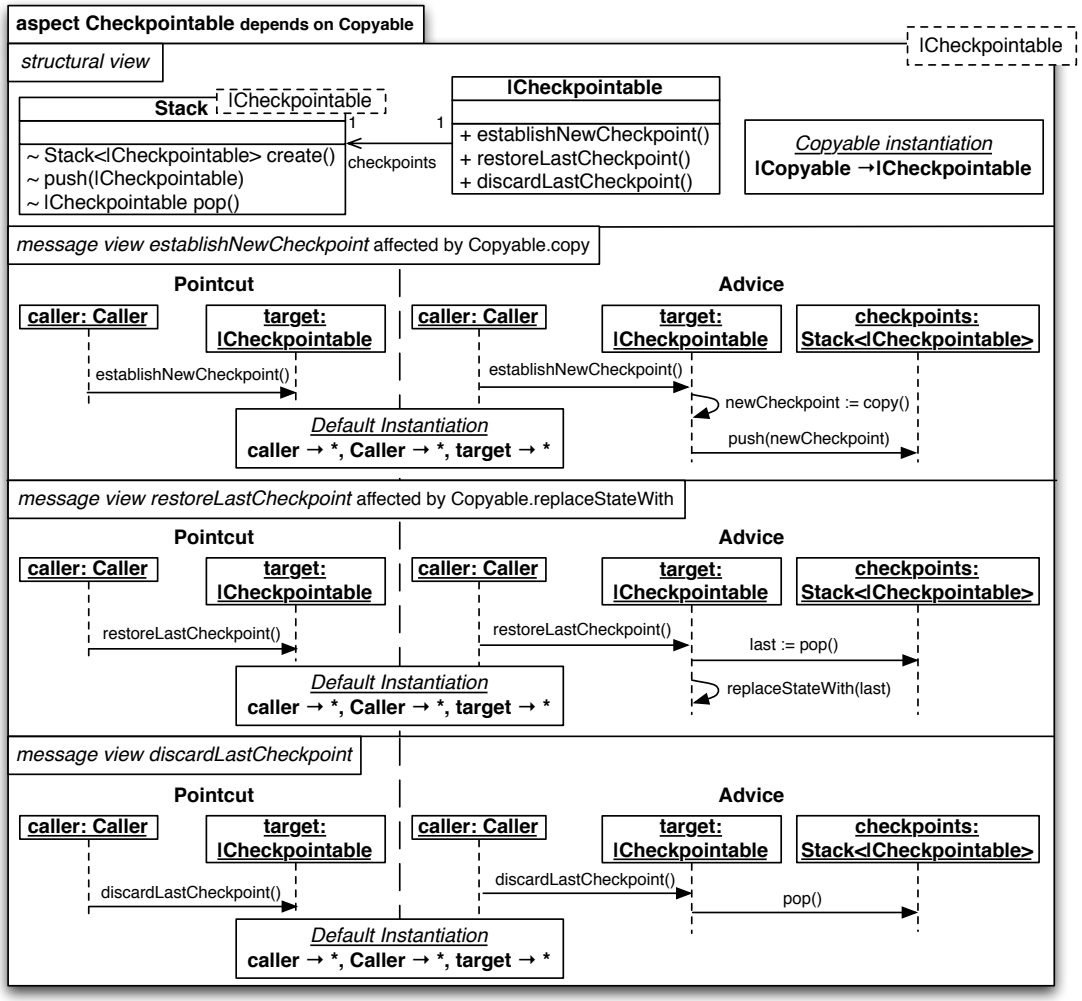

Figure 10: The *Checkpointable* aspect makes use of *Copyable* in order to store and retrieve snapshots of the state of objects.

These checkpoints are full copies of the transactional object that are provided by *Copyable*. As we only need to access the last checkpoint we can store them in a stack. Such a stack is associated to every *ICheckpointable* object and methods for its manipulation are added.

The message view for *establishNewCheckpoint* makes use of *Copyable*'s method *copy* and pushes the received duplicate on the stack. A past state of the object can be restored when the method *restoreLastCheckpoint* is called. It simply pops the last checkpoint from the stack and replaces the state of the current object with the checkpoint's state. Removing a checkpoint with the method *discardLastCheckpoint* is even simpler as the last element of the stack gets popped from the state without being used or returned.

3.1.9 *Checkpointing*

Automatic creation of checkpoints upon first access.

The *Checkpointing* aspect shown in Figure 11 makes use of the functionality provided by *Checkpointable* and automatically creates a checkpoint every time an object is accessed for the first time. The information needed for this check is provided by the aspect *Tracing*. It is also used to provide possibilities to restore or discard all checkpoints.

Within the message view *checkpointMethod* we advise every call to a method that is bound to the mandatory instantiation parameter |*m* of the |*Checkpointed* class. Before the execution of the original method behavior we retrieve the current participant, obtain its context and check whether the target of the advised call was already accessed before. If this is not the case, we establish a new checkpoint prior to executing the original behavior.

The message view of *restoreCheckpoints* shows how we restore the last checkpoint for every accessed object by iterating over the set of accessed objects and by calling the method *restoreLastCheckpoint* on it. As this operation is supposed to be an undo operation we also need to remove all traces that we created for these objects.

A possibility to remove checkpoints without restoring their state is shown in the message view of *discardCheckpoints*: similarly to the method *restoreCheckpoints* we iterate over all accessed objects, call the method *discardLastCheckpoint* on them and remove the corresponding traces.

3.1.10 *OutcomeAware*

Lets the participant vote on a outcome for its context.

The *OutcomeAware* aspect as presented in Figure 12 associates each context with an outcome that will make it possible to decide whether the corresponding transaction should be committed or aborted. A participant gets the possibility to vote on an outcome and leave a context at the same time.

In order to ease the extension of the framework with sophisticated voting and outcome mechanisms, we only require the abstract class *Outcome* to provide a method *isPositive* that returns a boolean. Our default implementation of an outcome is the class *BinaryOutcome*. It contains only a boolean value.

3.1.11 *Recovering*

Recovering the states of all accessed objects in case of failure.

The functionality of the aspects *OutcomeAware* and *Checkpointing* or *Deferrable* is combined by the *Recovering* aspect as shown in Figure 13 in order to automatically restore the state of transactional objects in case of a transaction with a negative outcome.

Recovering does not introduce any new methods or infrastructure. It specifies only how already existing aspects should be used

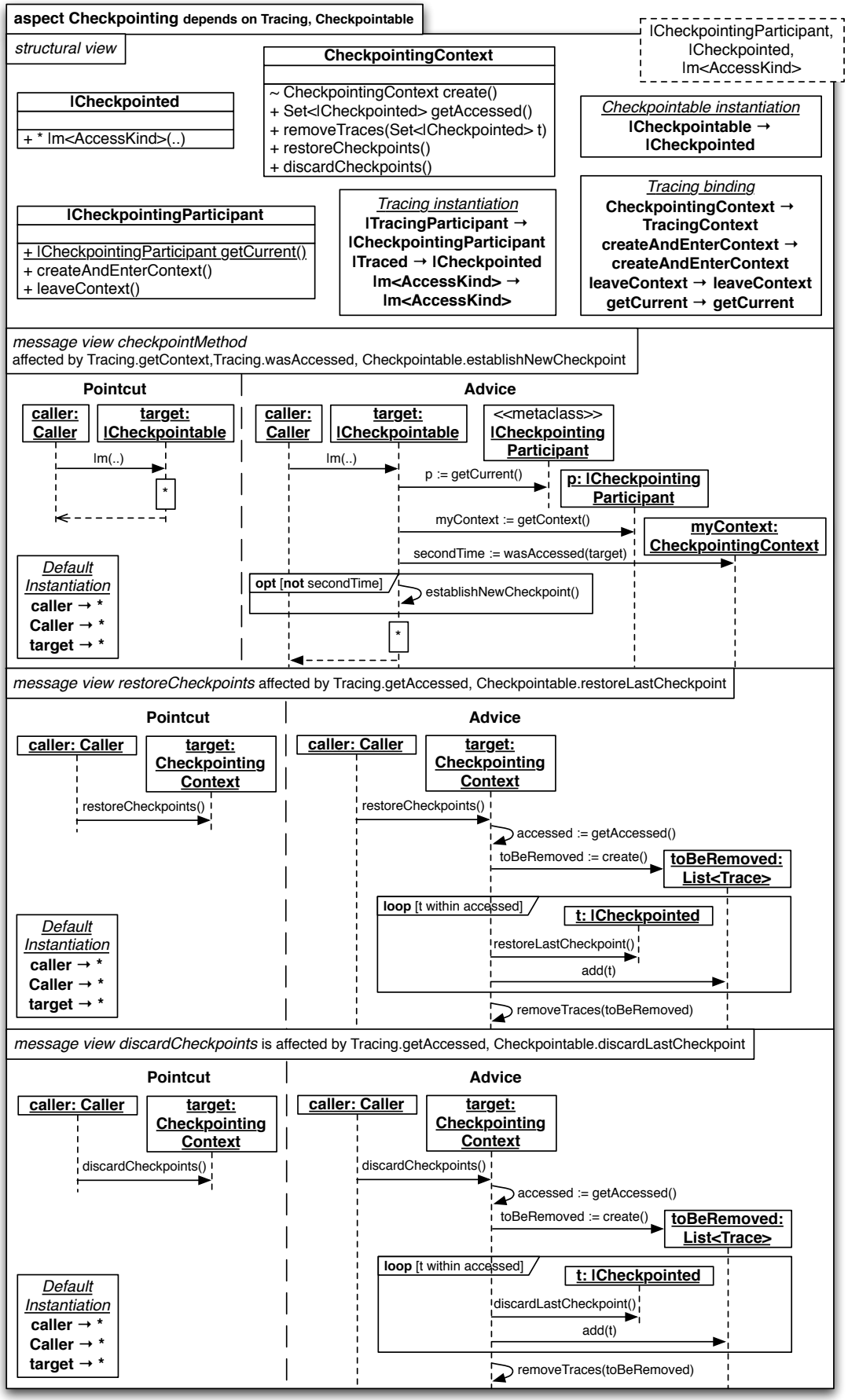

Figure 11: The *Checkpointing* aspect automatically maintains snapshots of objects based on the access history provided by *Tracing*.

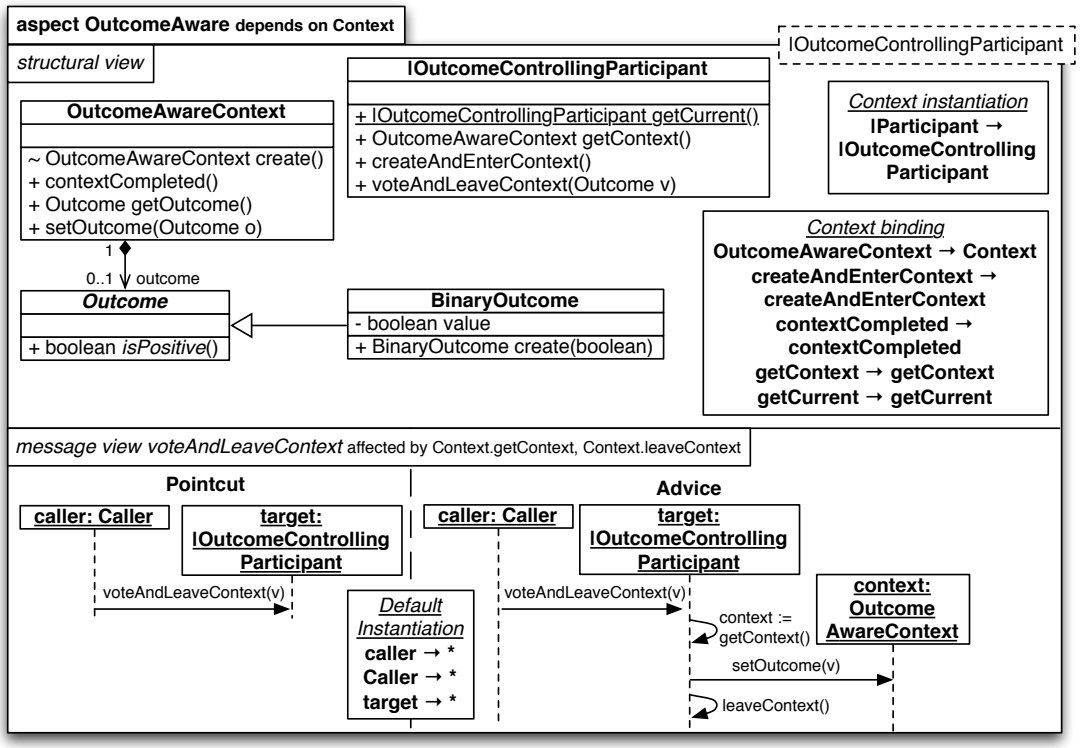

Figure 12: The *OutcomeAware* aspect associates contexts to outcomes and allows the participant to vote on the outcome.

and advises the marker method *contextCompleted*. We provide a message view for *contextCompleted* for the different update strategies as the recovery mechanism is different for both of them.

If the update strategy *Checkpointing* is chosen, we determine whether the transaction failed by retrieving the outcome and calling the method *isPositive* on it. In case of a negative outcome we call the method *restoreCheckpoints* to undo all operations of the transaction but if the transaction was successful we call the operation *discardCheckpoints* instead.

The *contextCompleted* method for the *Deferring* variant begins with a retrieval and inspection of the outcome in order to decide whether the transaction succeeded. The operations that were deferred are performed on the original objects by calling the method *performUpdate* if the transaction was successful. Finally, all traces are removed regardless of the outcome of the transaction.

3.1.12 *Nested*

Allowing contexts to create a nested hierarchy.

Even though the aspect *Nested* is not necessary for simple transactions, we decided to model it (Figure 14) in order to support nested transactions that can be aborted in case of an unsuccessful parental transaction. To make this possible we associate every context with children contexts and with a single parental context.

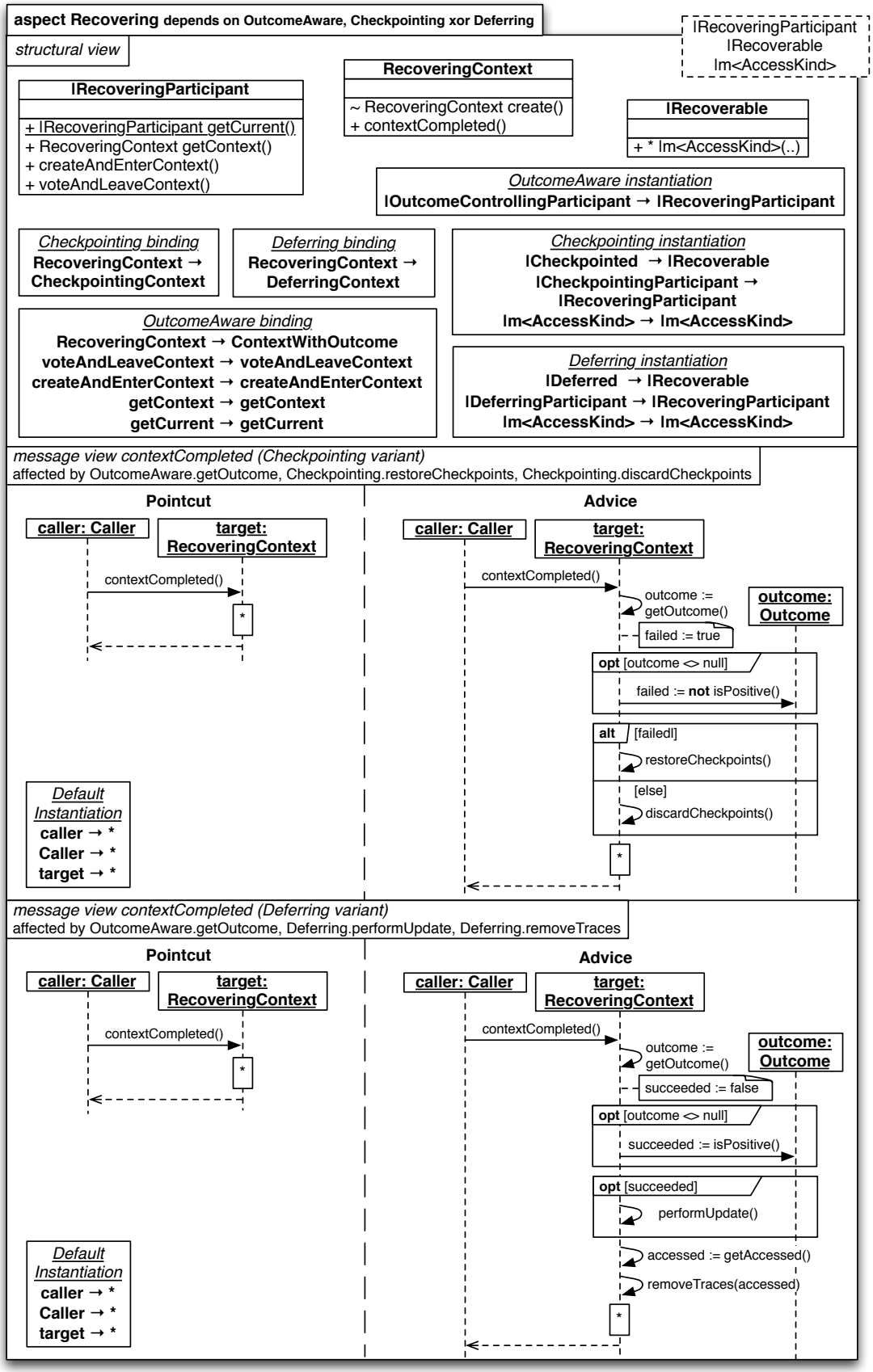

Figure 13: The *Recovering* aspect automatically recovers the state of objects if they complete with a negative Outcome and supports two different update strategies.

Additionally, methods to modify this association are introduced and the methods for creating and leaving contexts are advised.

By providing a message view for the method *createContext* we make sure that the old context of a participant that creates and enters a new context automatically becomes the parent of the newly created context. In the message view for the method *leaveContext* we check whether a context had a parent whenever it leaves a context. In such a case we migrate the participant to the parent context.

The message view *addChildrensResults* does not correspond to a method that is defined in the structural view but it is a message template that is reused in other aspects. This means that methods of other aspects can be bound to the method *m* of this message view in order to reuse the defined logic. The purpose of this message template is to apply a given method to every child context and to add the results of these method calls to the overall result. An example for methods that make use of this message template are the methods *getAccessed* and *getTraces* of the *Nested / Tracing* conflict resolution aspect that we present in Section 3.2.1.

3.1.13 *Lockable*

Automatically acquiring and releasing access kind specific locks.

The *Lockable* aspect as shown in Figure 15 provides the possibility to retrieve locks for transactional objects based on the access kind of methods in order to control concurrency. Every |*Lockable* object is associated to a lock object that provides methods to lock and unlock it.

In the message view of the method *getLock* we specify that the associated lock is initialized in a lazy manner when it is used for the first time. After the initialization the method *lock* is called on it. The message view for *releaseLock* is even simpler as it contains a single call to the *unlock* method of the associated lock.

As locks and semaphores are a common solution for concurrency problems we decided to keep the model for *Lockable* on an abstract level by omitting these implementational details. Although our model does not use the access kind information directly it allows for sophisticated implementations that use different locks for different types of access.

3.1.14 *TwoPhaseLocking*

Acquiring locks incrementally and releasing them in a single last step.

TwoPhaseLocking is a concurrency mechanism that acquires locks for transactional objects stepwise in a first phase and releases all of them simultaneously in a second phase after the successful completion of a transaction. The corresponding aspect is presented in Figure 16 and makes use of the aspects *Tracing* and *Lockable* in order to keep track of all access classified operations

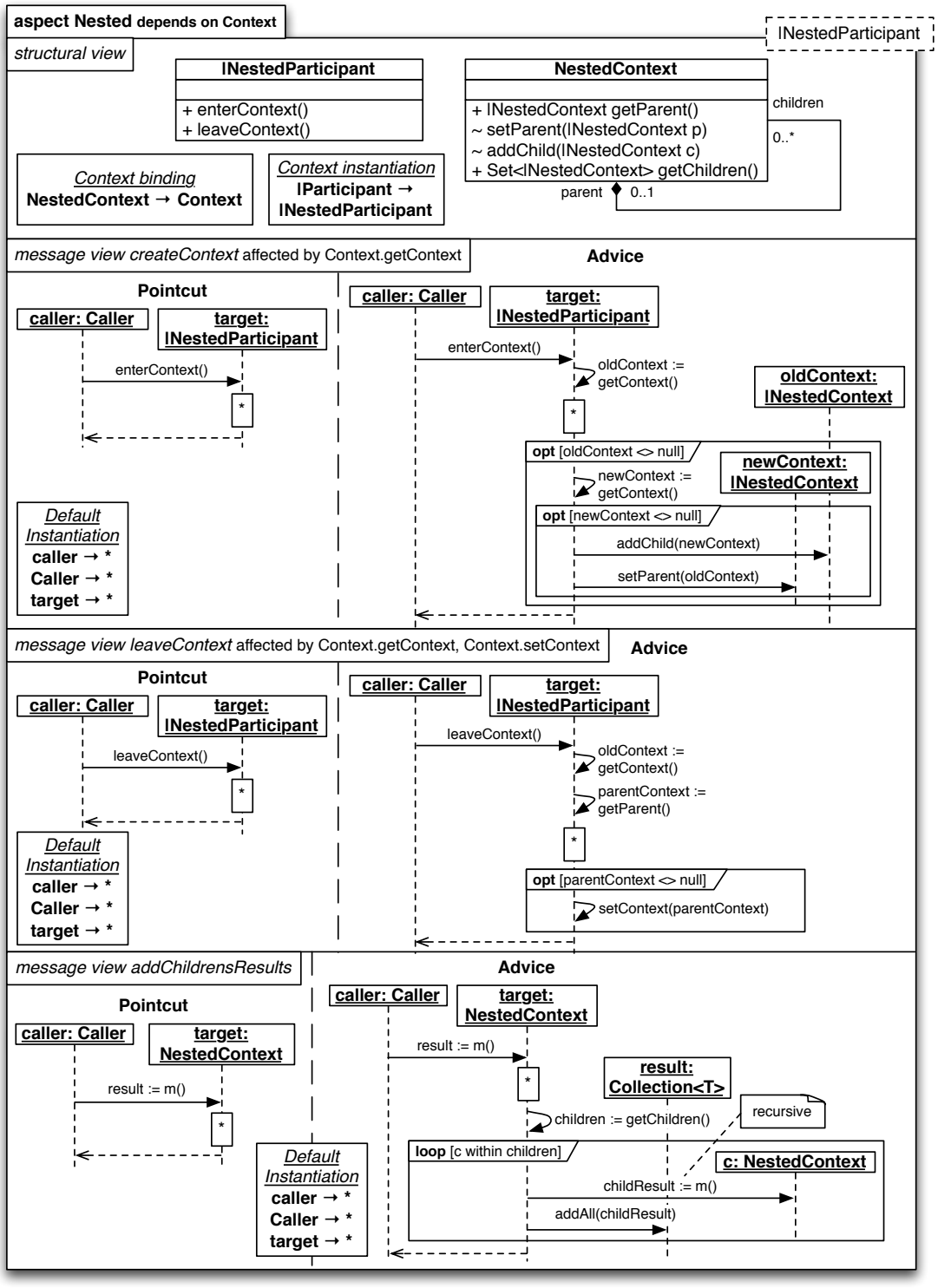

Figure 14: The *Nested* aspect allows for a nested hierarchy of contexts and maintains it automatically.

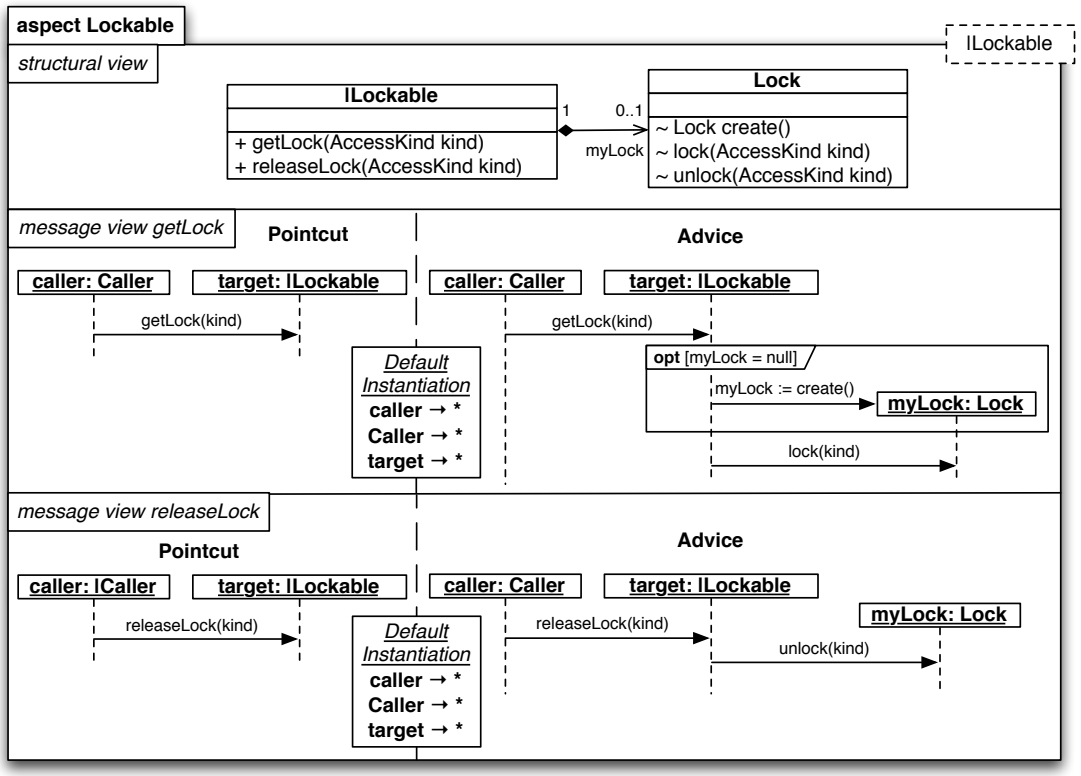

Figure 15: The *Lockable* aspect provides facilities to acquire and release locks based on access kinds.

involving transactional objects. If the participant does not already possess a lock of the corresponding kind for the accessed objects, such a lock is acquired automatically. When the transaction is completed and the participant left the context all these acquired locks are automatically released.

The message view *acquireLock* advises every call to a method that is bound to the mandatory instantiation parameter $!m$ of the *!TwoPhaseLocked* class. Before the original method logic gets executed the current participant is retrieved, its context is obtained, and the access kind of the advised operation is retrieved. A boolean variable is created and initialized to the constant *false*. The variable evaluates to *true* if the context already possesses a lock of the same kind. After the access kind was obtained all traces are retrieved and we iterate over them as long as we do not find out that we already possess a lock of the requested kind. For every trace we check whether the trace target is identical to the target of the advised method call. If this is the case we retrieve the access kind for the trace, and compare it to the access kind of the advised method. If they are equal, we know that we already accessed this object with the same access kind. That means that we already possess a corresponding lock so we can set the variable to the constant *true*. If the loop terminates and the variable is still *false* we know that we never accessed the same

object with the same access kind before so we need to acquire a corresponding lock.

The method *leaveContext* is advised in a message view that specifies that we store the context in a temporary variable before leaving it. After the original method logic we make use of this temporary variable as we call the method *releaseAcquiredLocks* and pass the old context to it.

The behavior of *releaseAcquiredLocks* is detailed in a message view as an iteration over all traces that calls the method *releaseLockIfPresent* using the target and access kind of the trace. We think that its functionality is revealed by its name and did not model it in a separate message view because we consider it to be an implementation detail.

3.1.15 *Transaction*

Employing all other aspects to provide an interface and the main functionality.

The *Transaction* aspect as shown in Figure 17 combines the functionality of the *Recovering* aspect with either the *TwoPhaseLocking* or the *OptimisticValidation* aspect, and optionally with the *Nested* aspect. The purpose of the *Transaction* aspect is to combine the different update strategies and concurrency models in a variable aspect that serves as the interface of the framework. The *OptimisticValidation* aspect is not yet included in the model, but we mentioned it as a dependency in *Transaction* as we already know that it will be used in a way that should not change the *Transaction* aspect.

In the message view *transactMethod* we outline the main functionality of the framework. We advise every call to a method that is bound to the mandatory instantiation parameter |*m* of the class |*Transactable*. Before the content of the advised method is executed we retrieve the current |*TransactionParticipant* and check whether the current method execution is already transacted. If this is the case, we proceed with the unchanged method behavior. Otherwise we add a call to *beginTransaction* before the original method content and add a call to *commitTransaction* after it.

The message view for *abortTransaction* shows how we reuse the *OutcomeAware* aspect that is indirectly provided by *Recovering*. We create a negative outcome and use it as an argument in a call to the *voteAndLeaveContext* method.

Committing a transaction is completely analogous to aborting it. Therefore the message view *commitTransaction* is exactly the same as that of *abortTransaction* except for the constant *true*.

The last message view of the aspect details how we decide whether a method *isAlreadyTransacted*. We assume that it is not transacted by initializing a boolean result variable to *false* and retrieve the current context of the participant. If such a context exists, then we change the value of the result variable to *true*.

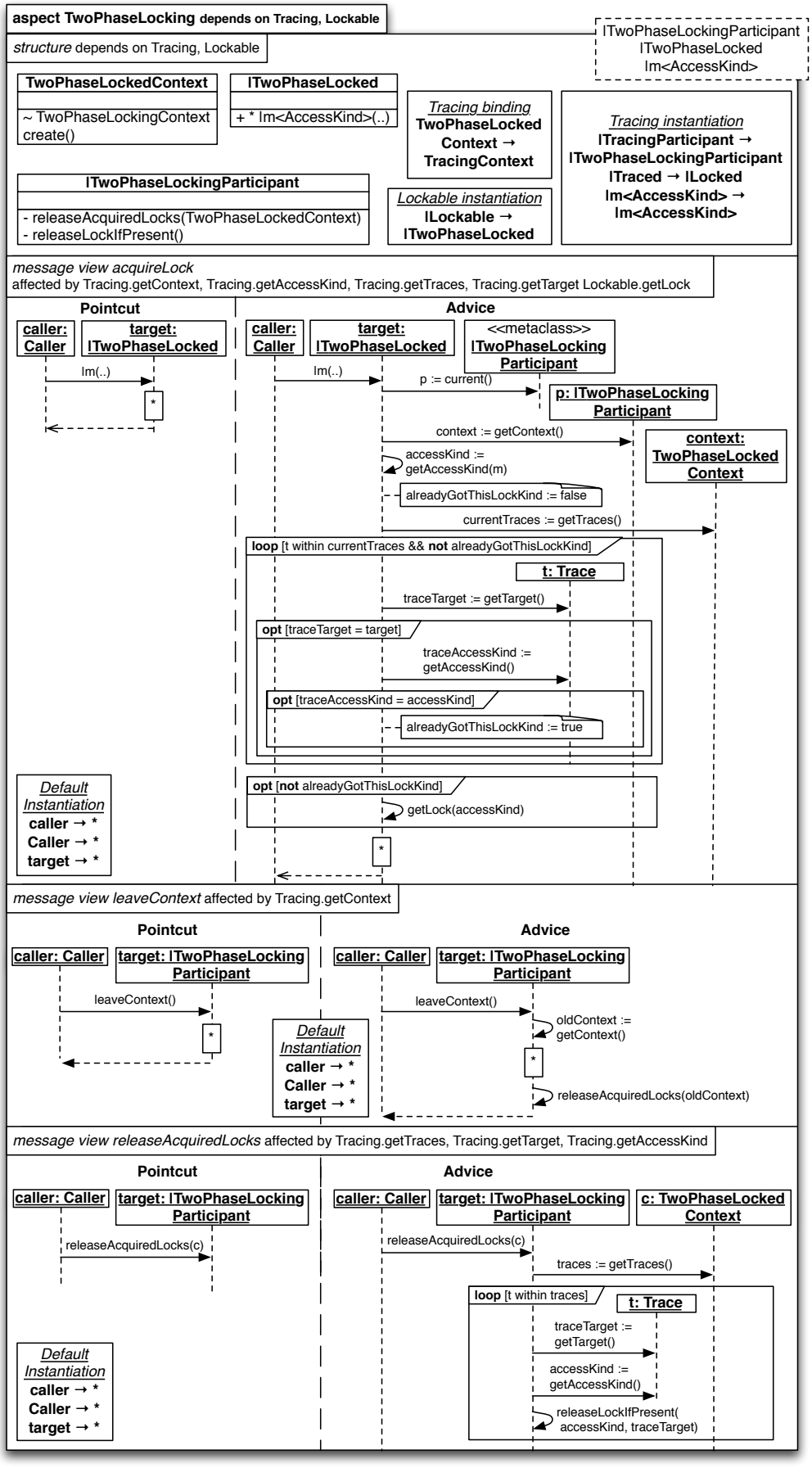

Figure 16: The *TwoPhaseLocking* aspect automatically acquires and releases locks based on the access history provided by *Tracing*.

Note that the crucial method *beginTransaction* is not detailed in a message view but directly bound to the method *createAndEnterContext* of the *Recovering* aspect.

3.2 CONFLICT RESOLUTIONS

As the combination of different aspects may result in conflicting behavior RAM offers the possibility to detect and resolve such conflicts with conflict resolution aspects. In Section 2.3.2 we introduced conflict resolution aspects and their interference criteria that specify under which circumstances they are instantiated. The AspectOPTIMA case study makes only use of interference criteria of the form *ClassA* = *ClassB* in order to activate a conflict resolution aspect only if *ClassA* is merged with *ClassB* as a result of instantiation or binding directives.

3.2.1 *Nested / Tracing*

Taking tracing information of child contexts into account.

In order to make the tracing information of child contexts available for parental context we modeled a conflict resolution aspect for *Nested* and *Tracing* as shown in Figure 18.

We advise the method *wasAccessed* in a message view by inserting additional logic after the original method behavior occurred and assigned a value to the return variable *accessed*. If the value of this variable is not *true*, we obtain the children from the context and iterate over them as long as the variable remains *false*. Within each iteration we assign the result of a call to the method *wasAccessed* that we perform on the child context to the result variable. Therefore, the advised method will return *true* whenever a context or one of its descendants accessed an object.

In order to be able to verify whether a context accessed an object directly without taken its children into account we specify a new method *wasDirectlyAccessed* in a separate message view. The content is the same as the content of the unmodified *wasAccessed* method that was defined in the *Tracing* aspect: a simple call to the *contains* method of the associated set of accessed objects.

The methods *getAccessed* and *getTraces* should also include the results of their children and therefore we bind them to the generic message template *addChildrensResults* of the *Nested* aspect. We specify that these methods are bound to the parameter *m* and provide the correct result type *Set<|Traced>* and *List<|Trace>*. This makes sure that the recursive addition of children results as defined in the *Nested* aspect is correctly applied to the original method behavior. In order to show the reuse of the method *addChildrensResults* we mention it in the message view labels with the keywords *affected by*.

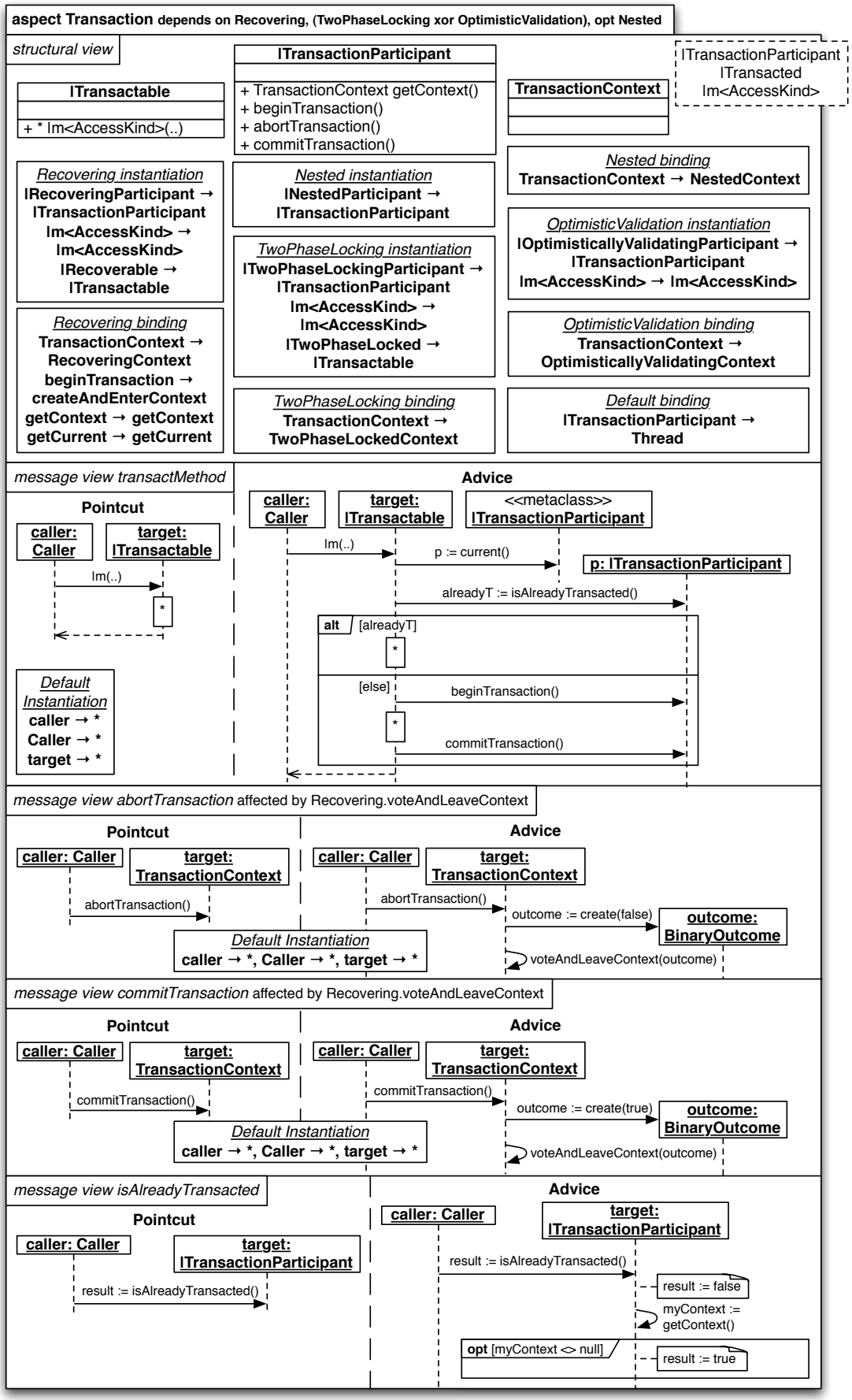

Figure 17: The *Transaction* aspect combines the functionality of all other aspects to provide the possibility of beginning, committing and aborting transactions.

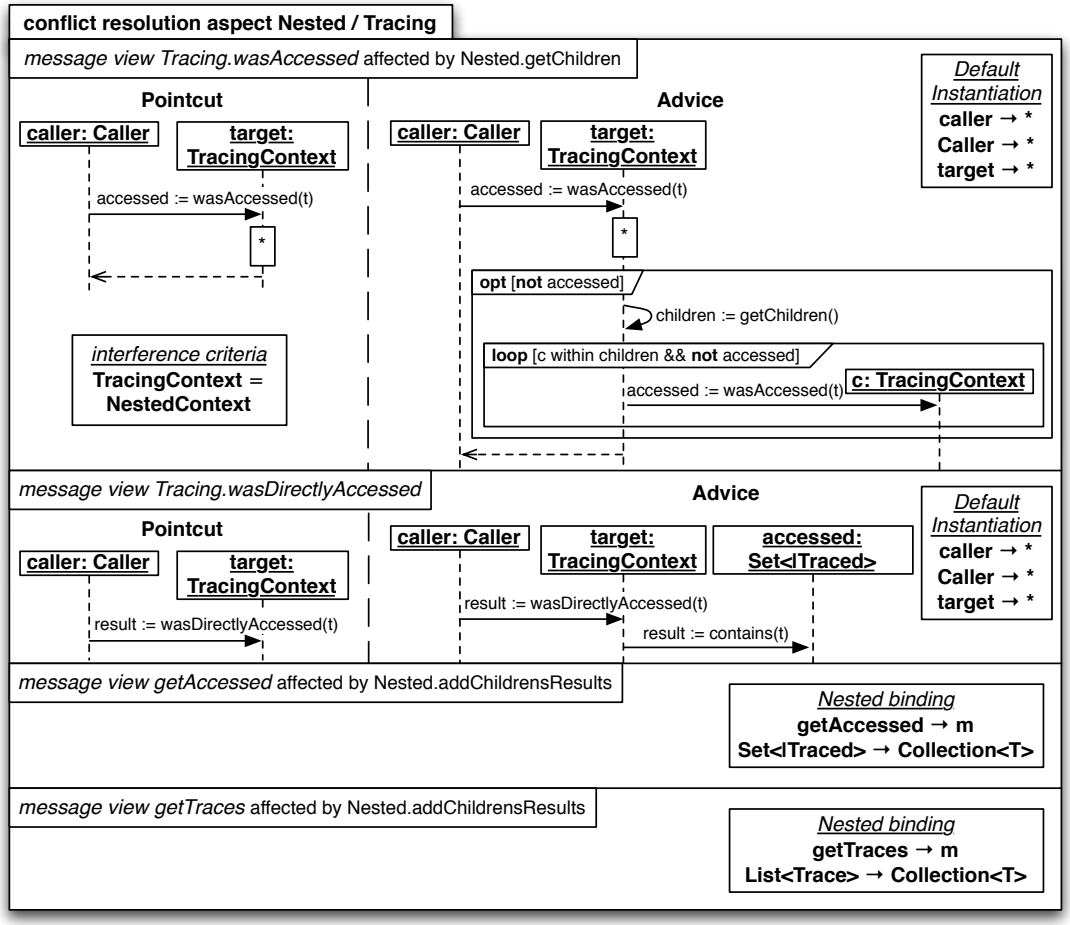

Figure 18: The conflict resolution aspect for *Nested / Tracing* recursively includes all tracing information of child contexts into queries.

3.2.2 *Checkpointing / Nested*

Avoiding the deletion of checkpoints that parental contexts might need.

As a child context may access objects that have not been accessed by its parent context we ensure that checkpoints of such objects are not discarded with a conflict resolution aspect for the aspects *Checkpointing* and *Nested* that we present in Figure 19.

We advise the execution of the method *discardCheckpoints* for all objects that are instances of the class *CheckpointingContexts* as well as instances of the class *NestedContexts*: we obtain the set of accessed objects from the context, create a list of traces that have to be removed and retrieve the associated parental context. While we iterate over all accessed objects we check whether a parental context exists. In such a case we use the *wasDirectlyAccessed* method that we defined in the previous *Nested / Tracing* conflict resolution aspect. If the accessed object that we are currently inspecting was accessed by the parental context we discard the last checkpoint of the corresponding object and add it to the list of objects that have to be removed. This means that checkpoints of objects that were not accessed by the parental context

are not discarded but kept for later undo operations. In a case without parental context we discard the last checkpoint of the inspected object and mark it for deletion without further checks.

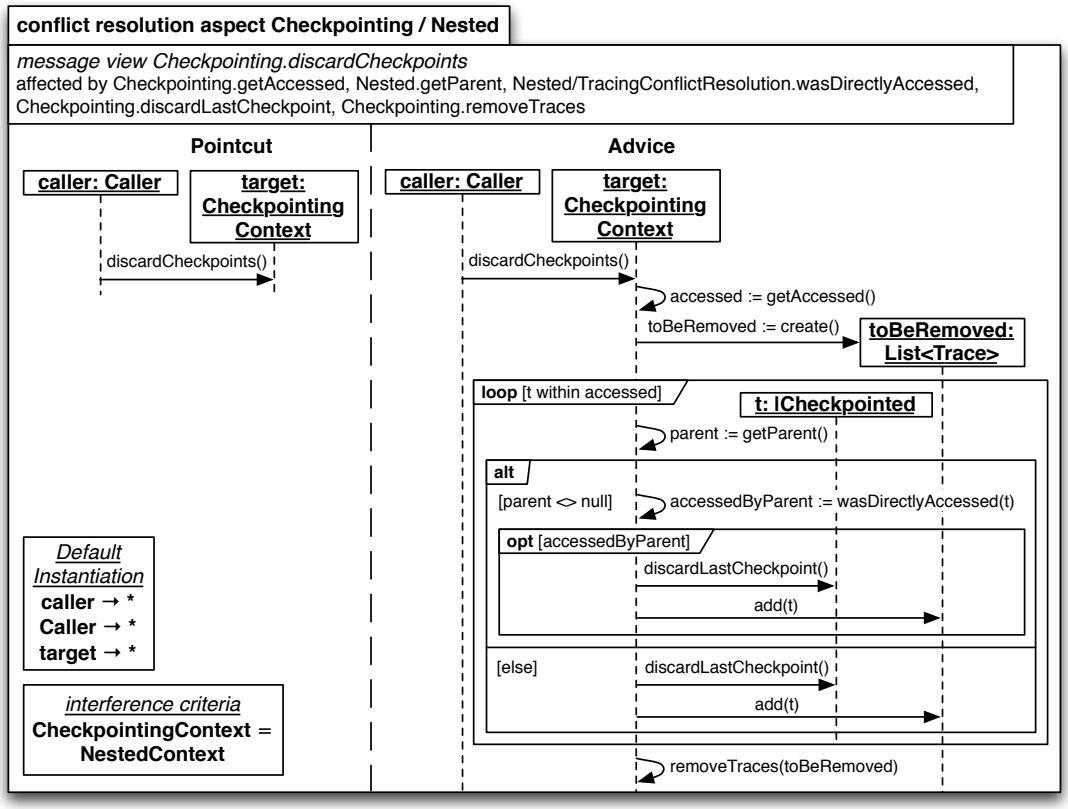

Figure 19: The conflict resolution aspect for *Checkpointing / Nested* keeps checkpoints of objects that were not accessed by the parent context instead of discarding them.

3.2.3 *Deferrable / Nested*

Originating new versions from existing versions of ancestors.

The purpose of the *Defferable / Nested* conflict resolution aspect as shown in Figure 20 is to make sure that the view on different versions of a deferrable object is consistent for hierarchies of nested conflicts. More specifically, we need to ensure that a nested context does not obtain a copy of the initial transactional object but a version that already reflects the modifications that were made by parental contexts.

We ensure this behavior by advising the method *getVersion* of the *Deferrable* aspect. At first, we try to obtain a mapped version directly from the associated *contextToVersionMap*. If this fails, then we iterate over all context ancestors until a context is found that is already in possession of a version of the transactional object in consideration. Once we found an ancestor with a version of the object we call the newly defined method *copyAndMapNewVersion* in order to duplicate and register a new version.

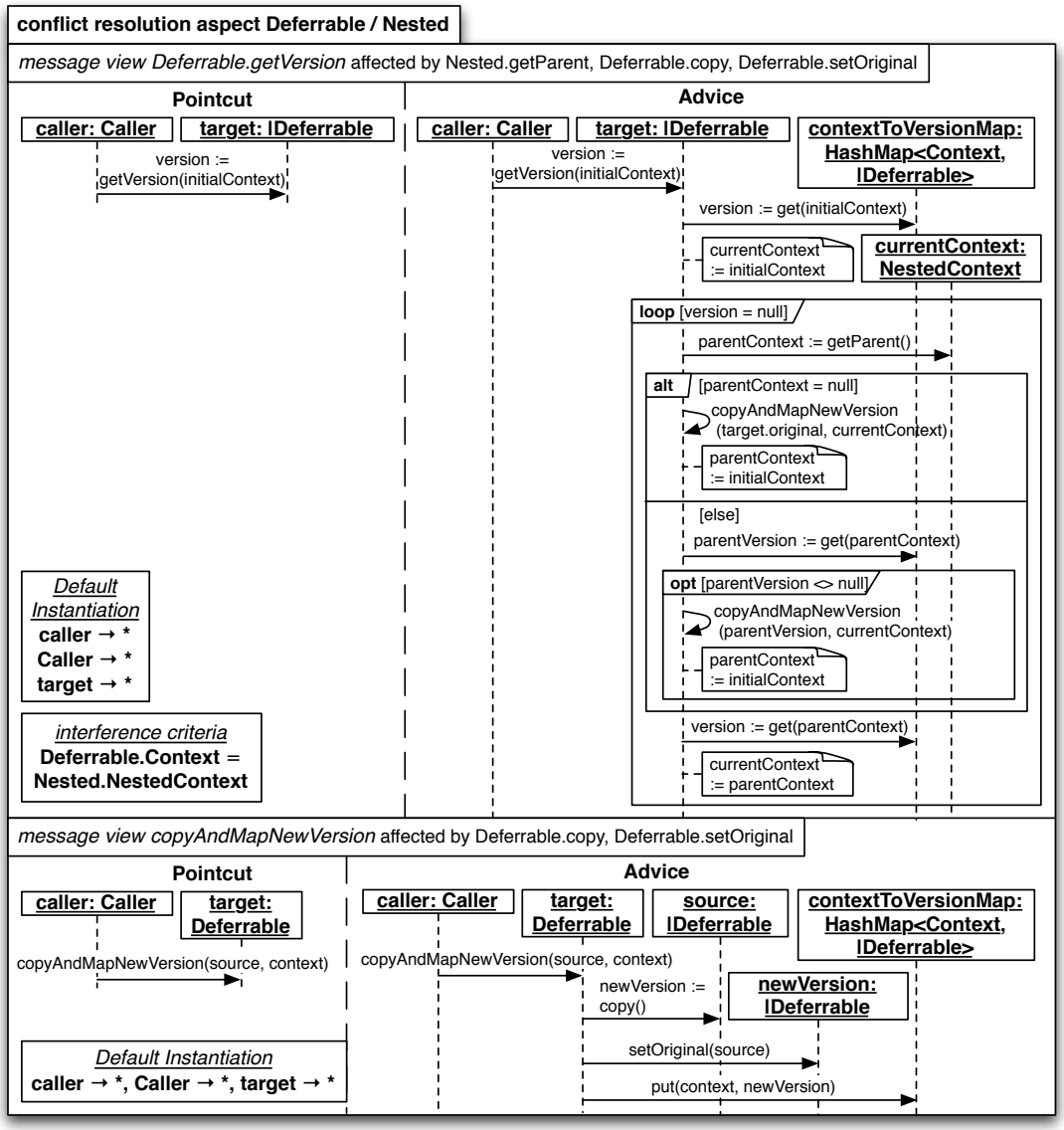

Figure 20: The conflict resolution aspect for *Deferrable / Nested* makes sure that versions of parent contexts are used as origin when new versions are created.

3.2.4 *Nested / TwoPhaseLocking*

Keeping locks of nested contexts for ancestors.

Our last conflict resolution aspect *Nested / TwoPhaseLocking* ensures that acquired locks are only released if no parental context exists that might require them as modeled in Figure 21. This modification reduces the probability of dead-locks and maintains the conformity to the principle of separated phases for acquiring and releasing locks. The content of the conflict resolution aspect is very short: we advise the *releaseAcquiredLocks* method such that its original behavior is only executed if no parental context exists.

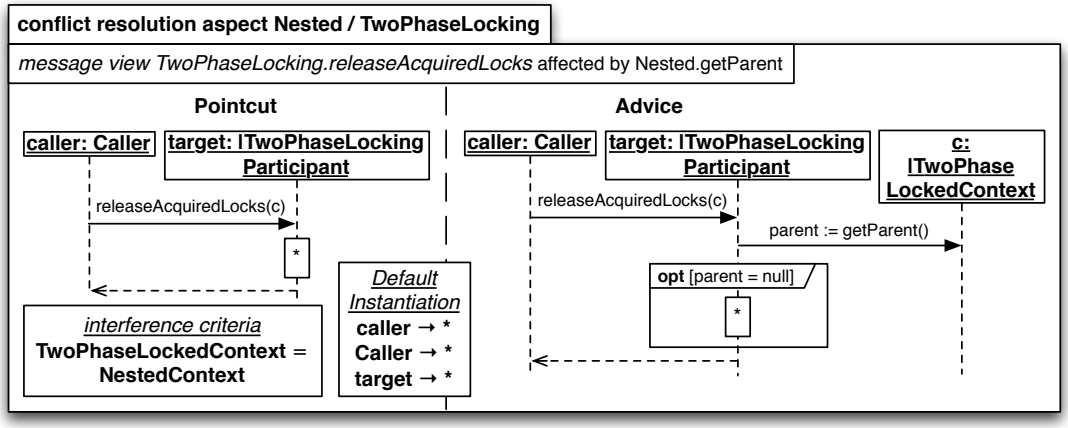

Figure 21: The conflict resolution aspect for *Nested / TwoPhaseLocking* makes sure that only the root context that has no parental context releases acquired locks.

4

OPEN MULTITHREADED TRANSACTIONS FOR ASPECTOPTIMA

An extension to the AspectOPTIMA case study.

We modeled an extension to the transaction system of AspectOPTIMA that supports Open Multithreaded Transactions in order to apply our general purpose mapping from RAM to AspectJ to a reasonable sized case study. The sheer number of aspects and the fact that a self-contained system is reused and extended generate a lot of interesting cases that we explored during the development of our mapping.

We decomposed the features of OMTT that we describe in Section 2.6.2 in twelve aspects that reuse each other and demanded the definition of four conflict resolutions. Our OMTT aspects depend directly on the aspects *OutcomeAware*, *Transaction*, *Nested*, *Context*, *Lockable*, and *AccessClassified* of the original AspectOPTIMA model. We were able to reuse their functionality seamlessly, so we could focus on the advanced features of Open Multithreaded Transactions.

In Figure 22 we present a feature diagram of our extension and show which combinations of aspects were resolved with conflict resolutions. The fact that almost every new aspect depends directly or indirectly on *Collaborative* illustrates that the main principle of OMTT is collaboration. The only exception to this dependency is the aspect *Blockable*. Together with the aspects *Pausable*, *Terminatable*, *OutcomeVotable* and *OutcomeVoting* it forms a dependency chain that shows that RAM's inter-aspect mechanisms allow fine-grained separation that facilitate reuse. The *OpenMultithreadedTransaction* aspect is the heart of our extension and depends directly or indirectly on every other aspect of the framework and its extension[1]. In order to give the reader a complete view of our extended case study we present all aspects and dependencies in a single feature diagram in Figure 23.

4.1 ASPECTS

Our Open Multithreaded Transaction model contains twelve aspects that package the main functionality and structure in small units that can be reused in other environments and frameworks.

1 This is not a smell for bad design as in some other software systems but a proof of the fact that our model contains no unused or superfluous functionality.

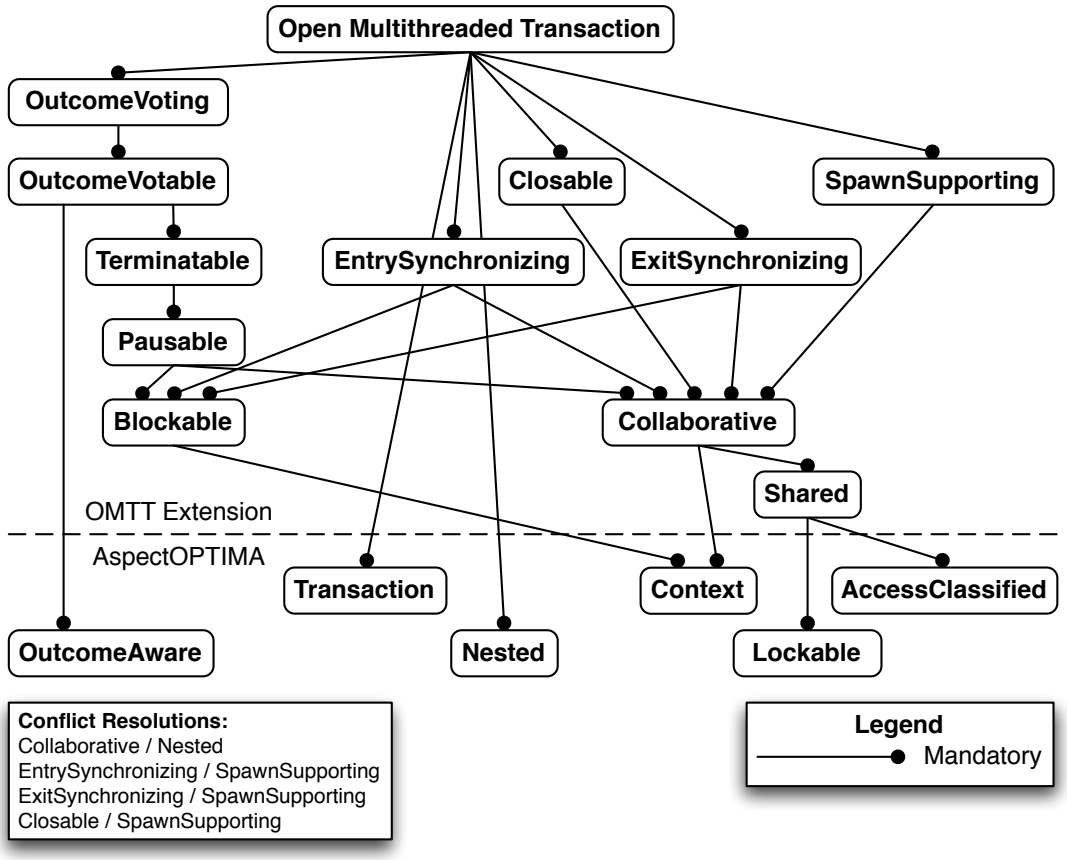

Figure 22: A feature diagram of AspectOPTIMA showing all aspects that were added for Open Multithreaded Transactions or that are directly reused.

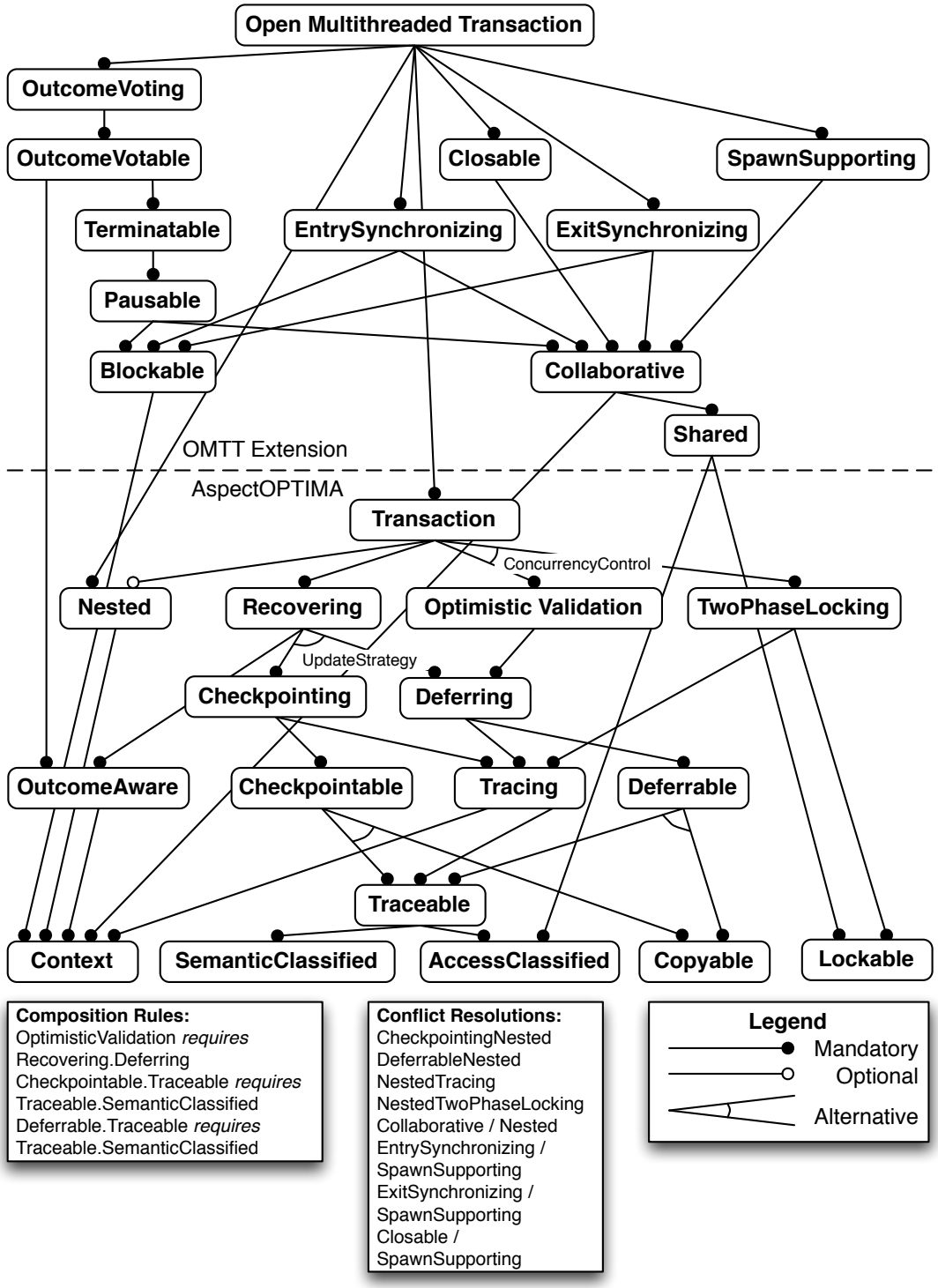

Figure 23: A feature diagram of AspectOPTIMA showing all aspects for Open Multithreaded Transactions and common single-threaded transactions.

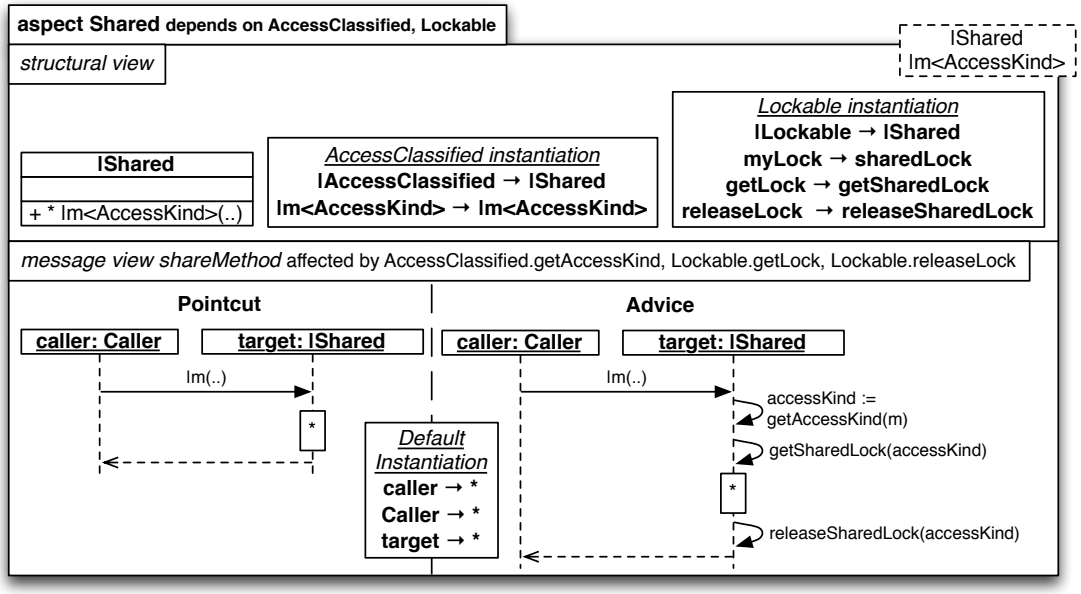

Figure 24: The *Shared* aspect automatically acquires and releases locks based on the access kind immediately before and after a participant accesses a shared object.

4.1.1 *Shared*

Concurrent access on shared objects needs to be restricted even for participants of the same transaction.

Shared objects of multithreaded transactions may be accessed simultaneously so they need to be protected from concurrent modifications to guarantee consistency. To achieve this the *Shared* aspect as shown in Figure 24 reuses the aspects *AccessClassified* and *Lockable* from the original AspectOPTIMA framework.

In the message view *shareMethod* we advise every call to a method that is bound to the mandatory instantiation parameter *Im* of the class *IShared*. Before we execute the unchanged behavior we retrieve the access kind of the advised method and acquire a lock that we release immediately after we executed the original method content. The obtained *sharedLock* differs from the *lock* of the *TwoPhaseLocking* aspect as we rename the association and corresponding methods in *Shared*. Because of this renaming it is possible to bind *IShared* and *ITwoPhaseLocked* to a common class in a reusing aspect without conflicts.

4.1.2 *Collaborative*

Collaboration is the central notion of Open Multithreaded Transactions.

The *Collaborative* aspect is presented in Figure 25 and it provides core functionalities to all other aspects as collaboration is the main principle of Open Multithreaded Transactions. It allows a context to be associated with multiple participants and provides facilities for joining already existing contexts through the methods *joinContext* and *isAllowedToJoin*.

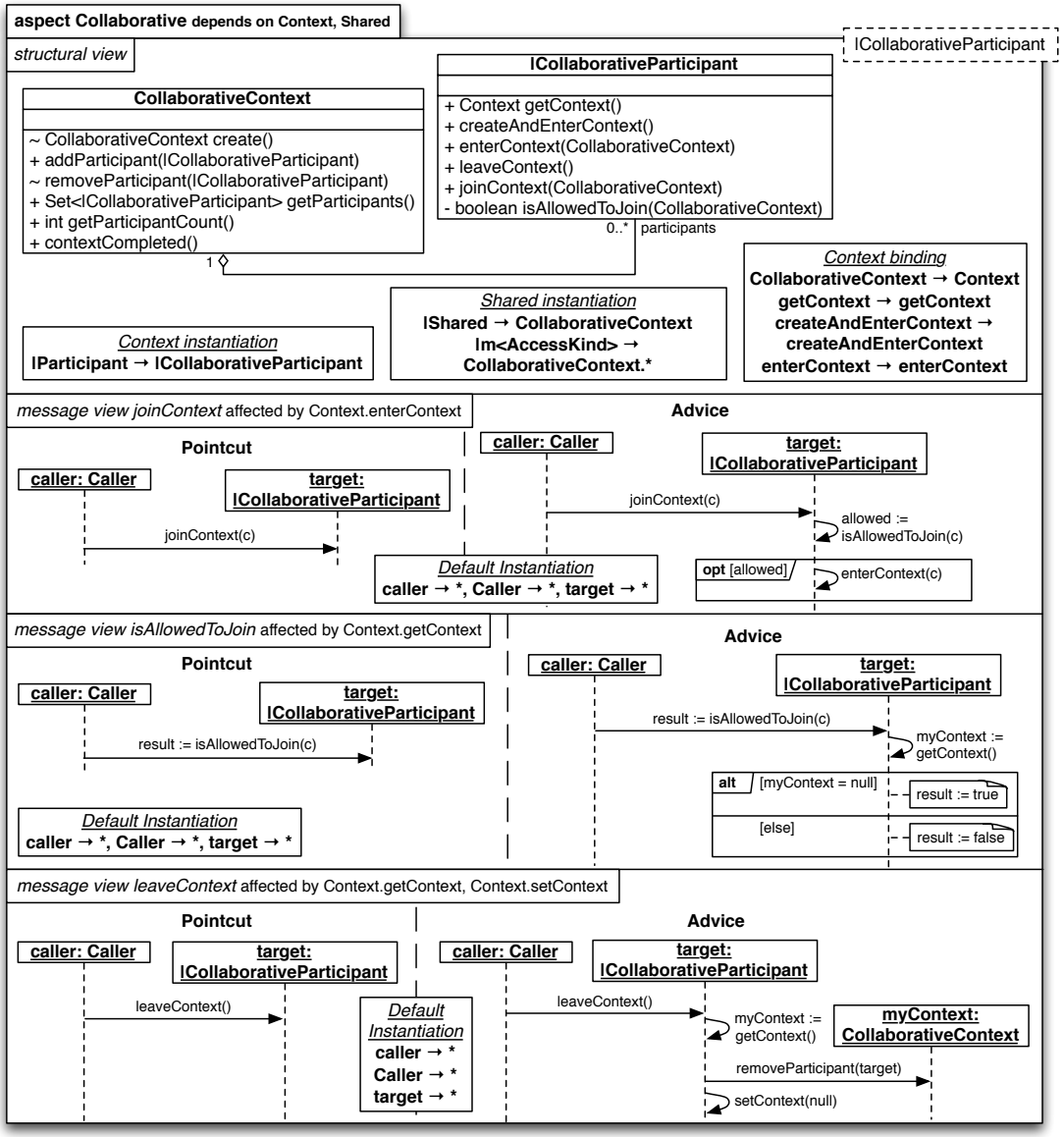

Figure 25: The *Collaborative* aspect permits multiple participants in one context and makes it possible to join existing contexts.

The message view of *joinContext* uses the result of a call to the method *isAllowedToJoin* as a condition for entering the context. We can consider the message view of *isAllowedToJoin* a default implementation that can be changed conveniently if another behavior is desired. In the modeled standard case a participant can only join a context if it is not already participating in a context. The message view for *leaveContext* is identical to the message view of *Context.leaveContext* except for the omitted call to the marker method *contextCompleted*. This is a correct modification as a context with multiple participants is not necessarily completed once a participant left.

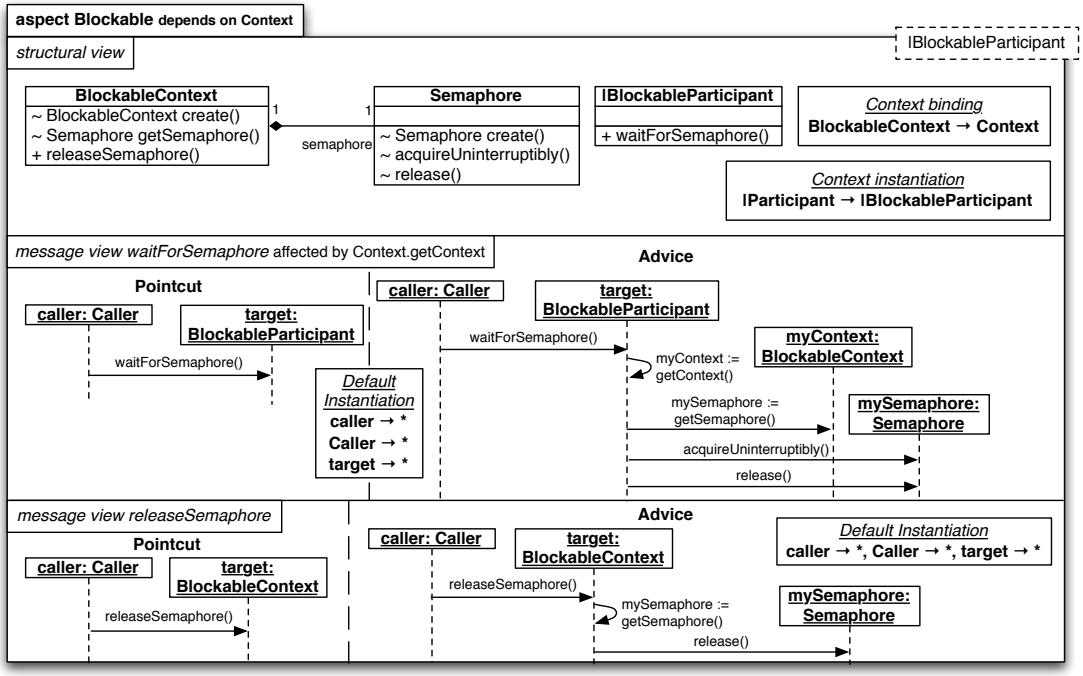

Figure 26: The *Blockable* aspect associates a context to a semaphore and allows participants to wait until they acquire it.

4.1.3 *Blockable*

Synchronization using a semaphore as a general strategy to block and release participants.

The *Blockable* aspect shown in Figure 26 introduces the general principle of blocking a participant of a context through a semaphore that is associated to the context of the participant. Under what circumstances a participant is blocked and how it is released again is not defined in the *Blockable* aspect itself but in reusing aspects. For this reason the method *waitForSemaphore* is detailed in a message view but never called within the aspect. In the message view the context of the participant is retrieved, its semaphore is obtained, and the methods *acquireUninteruptibly* and *release* are called. We release the semaphore immediately after we obtained it in order to give other participants the possibility to acquire the semaphore as well. This means that a single permit can release multiple participants.

In order to release the semaphore independent of a possible acquisition we provide the method *releaseSemaphore* in a separate message view. It simply obtains the associated semaphore and calls the method *release* on it.

4.1.4 *Pausable*

Pausing and continuing a context and all of its participants.

The aspect *Pausable* (Figure 27) augments the *Context* aspect with possibilities to pause and continue contexts and their participants arbitrarily by reusing the *Blockable* aspect. To achieve this a boolean variable *paused* is introduced into the class *Pausable-*

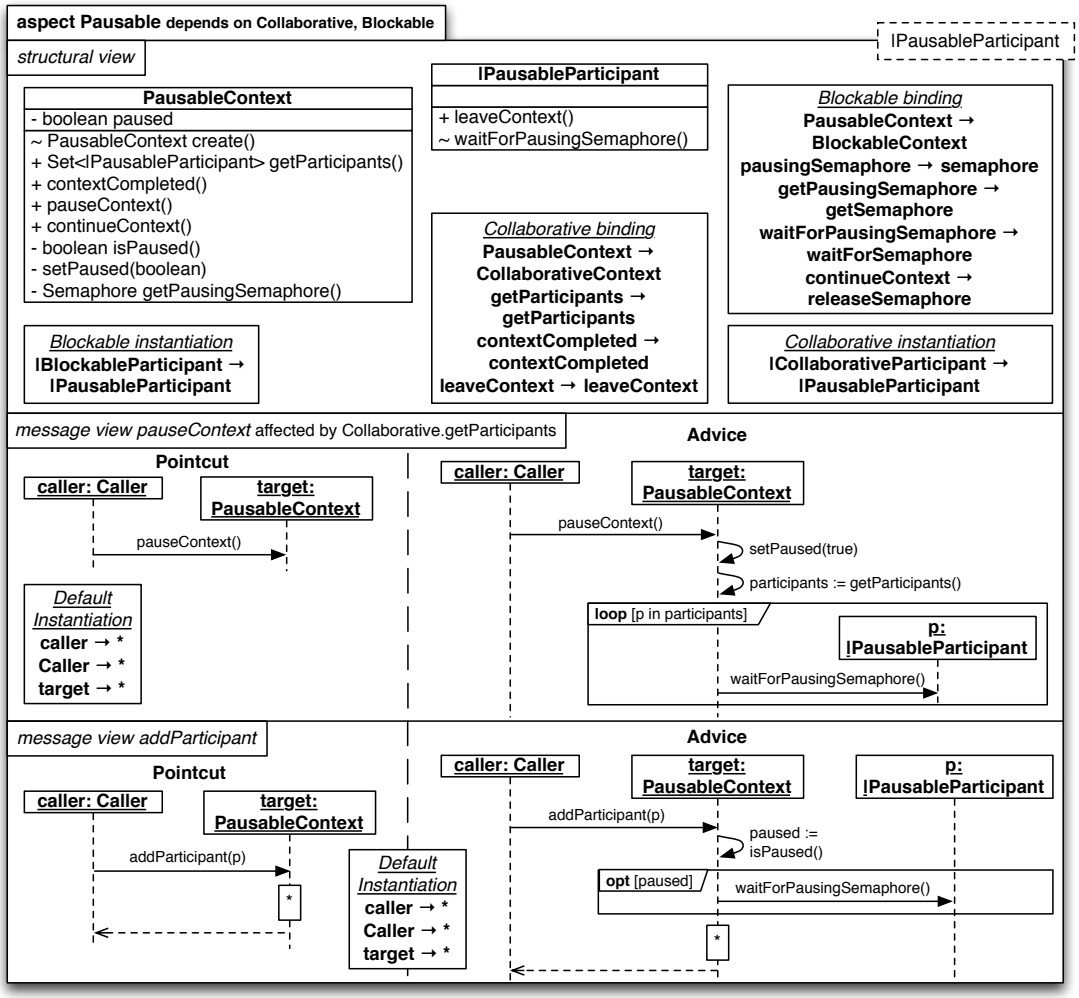

Figure 27: The *Pausable* aspect provides the functionality to pause a context and all its participants.

Context along with methods that allow us to use and modify this variable. The methods *pauseContext* and *continueContext* are also introduced into *PausableContext*, the later being directly bound to the *releaseSemaphore* method of the *Blockable* aspect. The method *pauseContext* however is detailed in a message view: when a context gets paused we simply mark it as paused and make every participant wait for the pausing semaphore. When a context is continued the method *releaseSemaphore* that is bound to *continueContext* releases the semaphore and all participants are unblocked as they all acquire and release the semaphore.

In order to make sure that participants that are added to a paused context get paused too we advise the existing method *addParticipant* in a message view. Whenever we add a participant we check whether the context is paused before we continue with the original behavior. If the participant should be added to a paused context, we make the participant wait for the semaphore in order to pause it too.

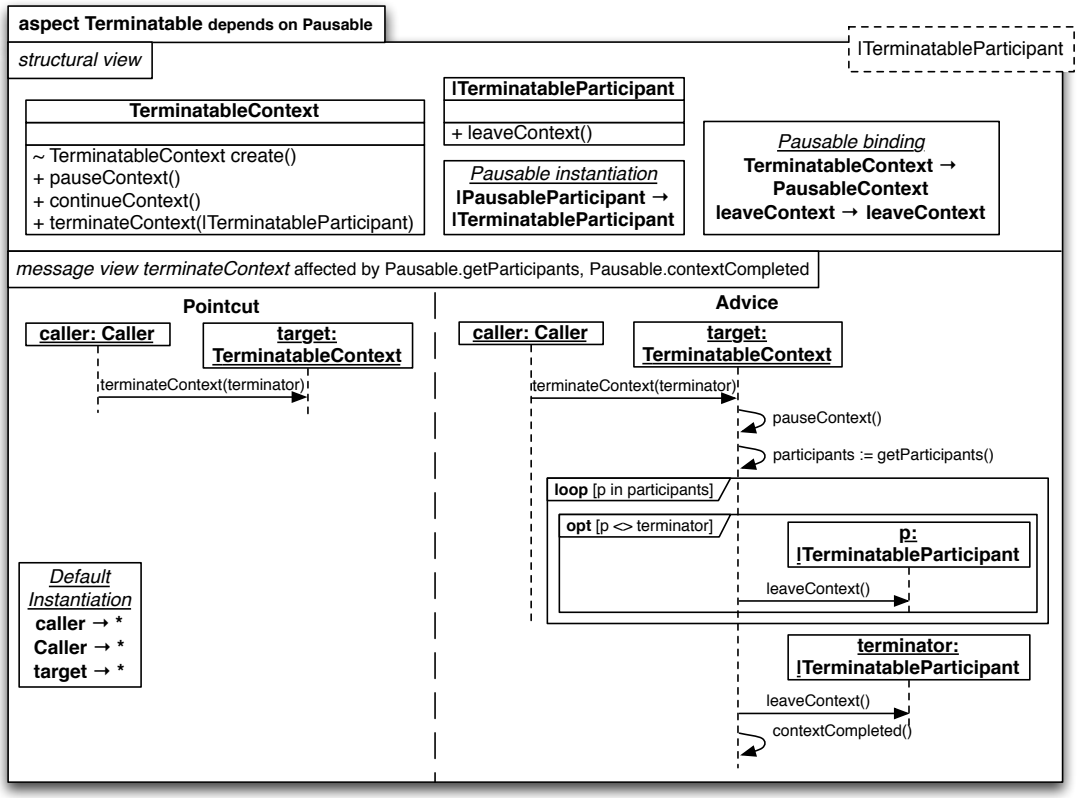

Figure 28: The *Terminatable* aspect allows the immediate termination of a context by making all participants leave it.

4.1.5 *Terminatable*

Terminate a context by forcing all participants to leave.

The *Terminatable* aspect shown in Figure 28 makes us of *Pausable* in order to provide the functionality to force the termination of a context and all its participants. For this purpose we declare the method *terminateContext* in *TerminatableContext* as shown in the structural view. The message view for *terminateContext* illustrates that we terminate all participants before we consider a context terminated and call *contextCompleted*. In order to be able to make all other participants leave we exclude the terminator from our loop and make it leave the context as last participant.

4.1.6 *OutcomeVotable*

Participants can vote and agree on a common outcome.

OutcomeVotable (Figure 29) depends on *Terminatable* and the *OutcomeAware* aspect of the original AspectOPTIMA framework in order to make it possible to vote on the outcome of a transaction upon leaving its context. The aspect automatically terminates a context once its outcome is decided. In the structural view we added the methods *registerVote*, *alreadyVoted*, *computeOutcome* and *isOutcomeDecided* to the class *OutcomeVotableContext*.

Furthermore, we included an association *votes* that maps each *OutcomeVotableParticipant* to an *Outcome*.

The message view for the method *voteAndLeaveContext* replaces the behavior provided by *OutcomeAware*. It illustrates that we register the provided vote at the associated context prior to leaving the context. The functionality of *registerVote* is to add the vote to the *votes* map and to check whether the outcome is already decided. If it is decided, we compute the outcome, set the outcome of the context accordingly and terminate the context.

Reusing aspects can find out whether a participant already voted or not by calling the method *alreadyVoted*. It returns the result of a call to the *containsKey* method on the associated *votes* map in order to decide whether the participant has already been mapped to a vote. If a reusing aspect wants to use a sophisticated voting mechanism he has to advise the method *computeOutcome* in order to replace our default message view. We assume a postive outcome and iterate over all registered votes. If we encounter a negative vote we assign a negative outcome to our return variable. This means that the default voting strategy requires an unanimous positive vote for a positive outcome.

4.1.7 *OutcomeVoting*

Participants that did not vote yet should automatically vote upon leaving.

The aspect *OutcomeVoting* as shown in Figure 30 makes sure that every participant that leaves a context automatically votes on its outcome by using the functionality of *OutcomeVotable*. The concise structural view shows that we added only one method *getDefaultVote* that provides us with the default voting behavior.

The modification of *leaveContext* as shown in the corresponding message view retrieves the context of the participant and determines whether the participant already voted. If this is not the case, the *getDefaultVote* method is called and the returned vote is registered prior to leaving the context. As a participant that leaves without voting should implicitly abort the corresponding transaction we define in a message view for *getDefaultVote* that a new negative *BinaryOutcome* should be returned.

4.1.8 *ExitSynchronizing*

To maintain isolation participants need to leave a context simultaneously.

ExitSynchronizing (Figure 31) introduces the possibility to synchronize the exit of a context by blocking all exiting participants until the last participant tried to exit. For this purpose we add the integer variable *blockedParticipantCount*, corresponding methods, and a method *isLastExitingParticipant* to the *ExitSynchronizingContext* class. We introduce the methods *getExitSemaphore* and *releaseExitSemaphore* into this context class and we introduce the method *waitForExitSemaphore* into |*ExitSynchronizingParticipant*.

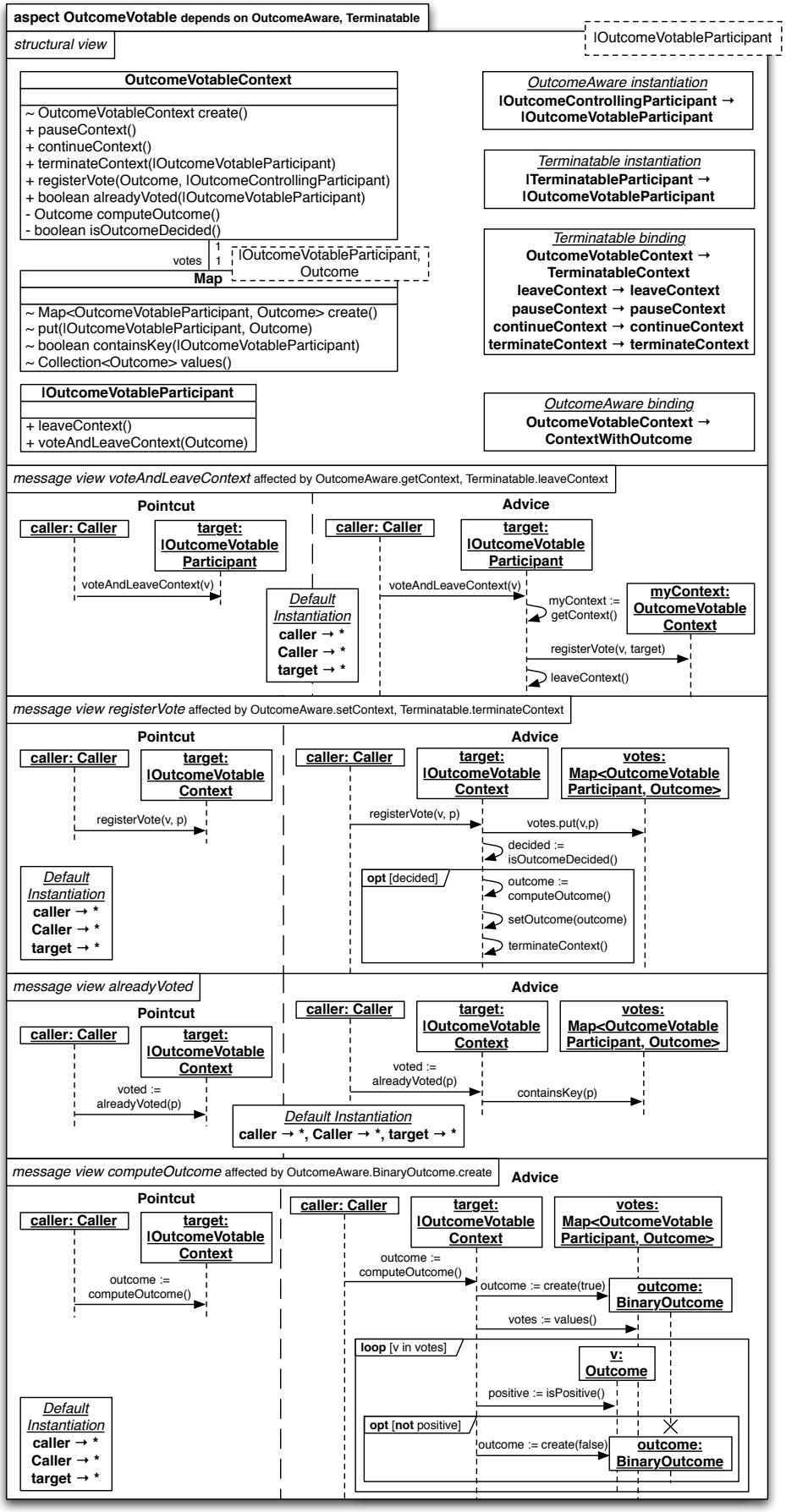

Figure 29: The *OutcomeVotable* aspect gives participants of a context the possibility to vote on an outcome and to determine an overall outcome from these votes.

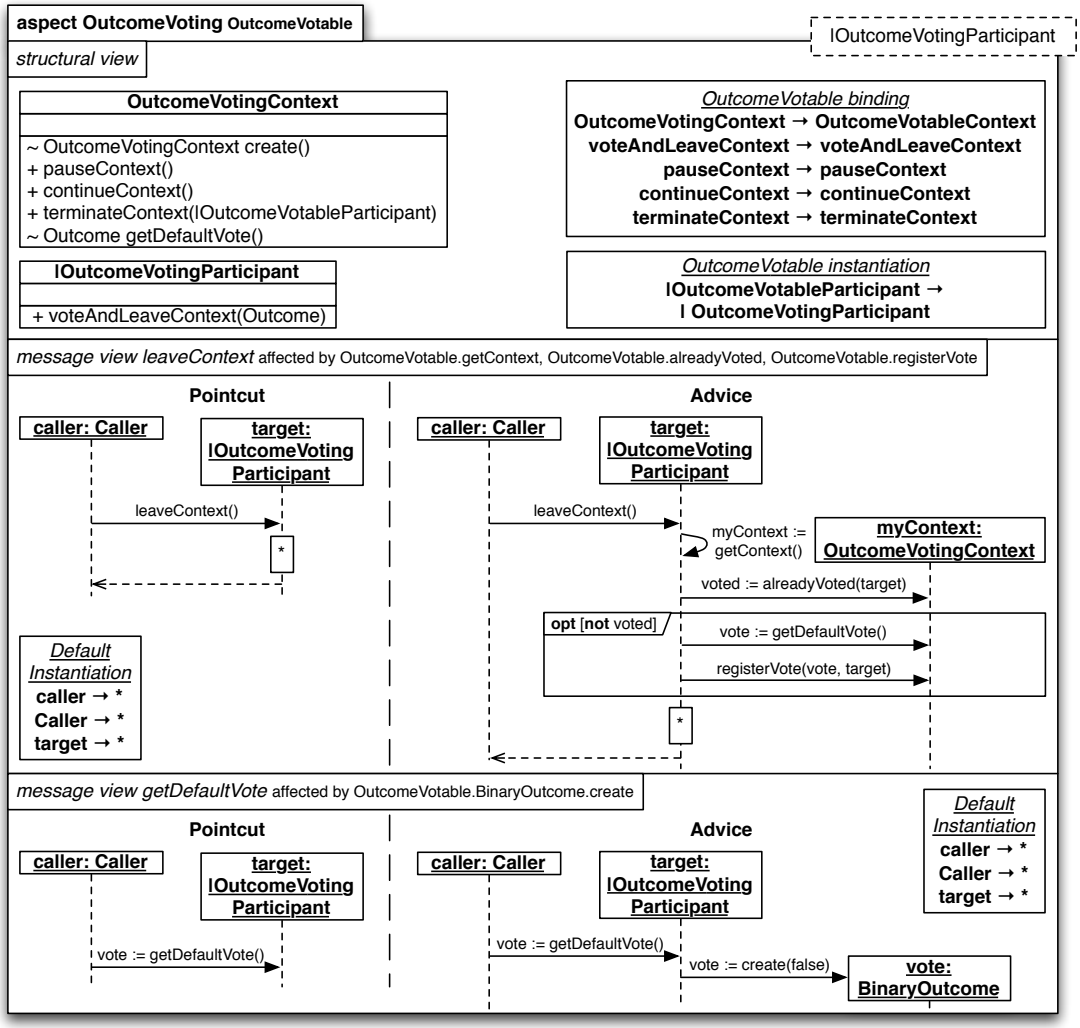

Figure 30: The *OutcomeVoting* aspect automatically votes on an outcome using a default vote if a participant leaves without voting.

The message view for *leaveContext* adds a call to the new method *waitOrReleaseBeforeLeaving* before the original method content. We created a new method instead of inserting its content directly into the message view of *leaveContext* in order to have the possibility to skip the newly added behavior in reusing aspects. Additional information to this decision is given in Section 4.2.2 where we describe a conflict resolution aspect that uses the new method to skip the waiting behavior for spawned participants.

Within the message view of the method *waitOrReleaseBeforeLeaving* we retrieve the context of the target participant, increase the number of blocked participants and determine whether we should block or release the participant by calling *isLastExitingParticipant*. If the participant is the last participant that tries to leave the context we release the associated semaphore thus terminating the waiting period of all other participants. If the current participant is not the last to leave the context we block it by calling *waitForExitSemaphore*. The point of computation after these alternatives is only reached when the participant either released the semaphore or was released by another participant during the execution of *waitForExitSemaphore*. We decrease the number of blocked participants in both cases.

The method *isLastExitingParticipant* assumes that the current participant is not the last as it initializes the return variable *last* to *false*. After this we retrieve the number of blocked participants as well as the number of participants in general. If the number of blocked participants is at least as big as the overall number of participants, the value of the return variable is changed to *true*.

4.1.9 *EntrySynchronizing*

Block contexts until a minimal number of participants is reached.

The aspect *EntrySynchronizing* as modeled in Figure 32 adds functionality to the *Collaborative* aspect that is similar to *ExitSynchronizing* but blocks participants that enter a context until a given number of minimal participants is reached. Methods and variables that are analogous to those introduced and bound in *ExitSynchronizing* are declared in the structural view and detailed in the two message views *waitOrReleaseBeforeEntering* and *isLastEnteringParticipant*. In addition to this, an integer variable *minParticipantCount* and a corresponding getter and setter method are introduced into the class *EntrySynchronizingContext*.

Instead of advising the method *leaveContext* we advise the method *enterContext* in a message view in order to call the method *waitOrReleaseBeforeEntering* after the original method content of *enterContext*. The behavior of the method *waitOrReleaseBeforeEntering* is completely analogous to the method *waitOrReleaseBeforeExiting* of the aspect *ExitSynchronizing*.

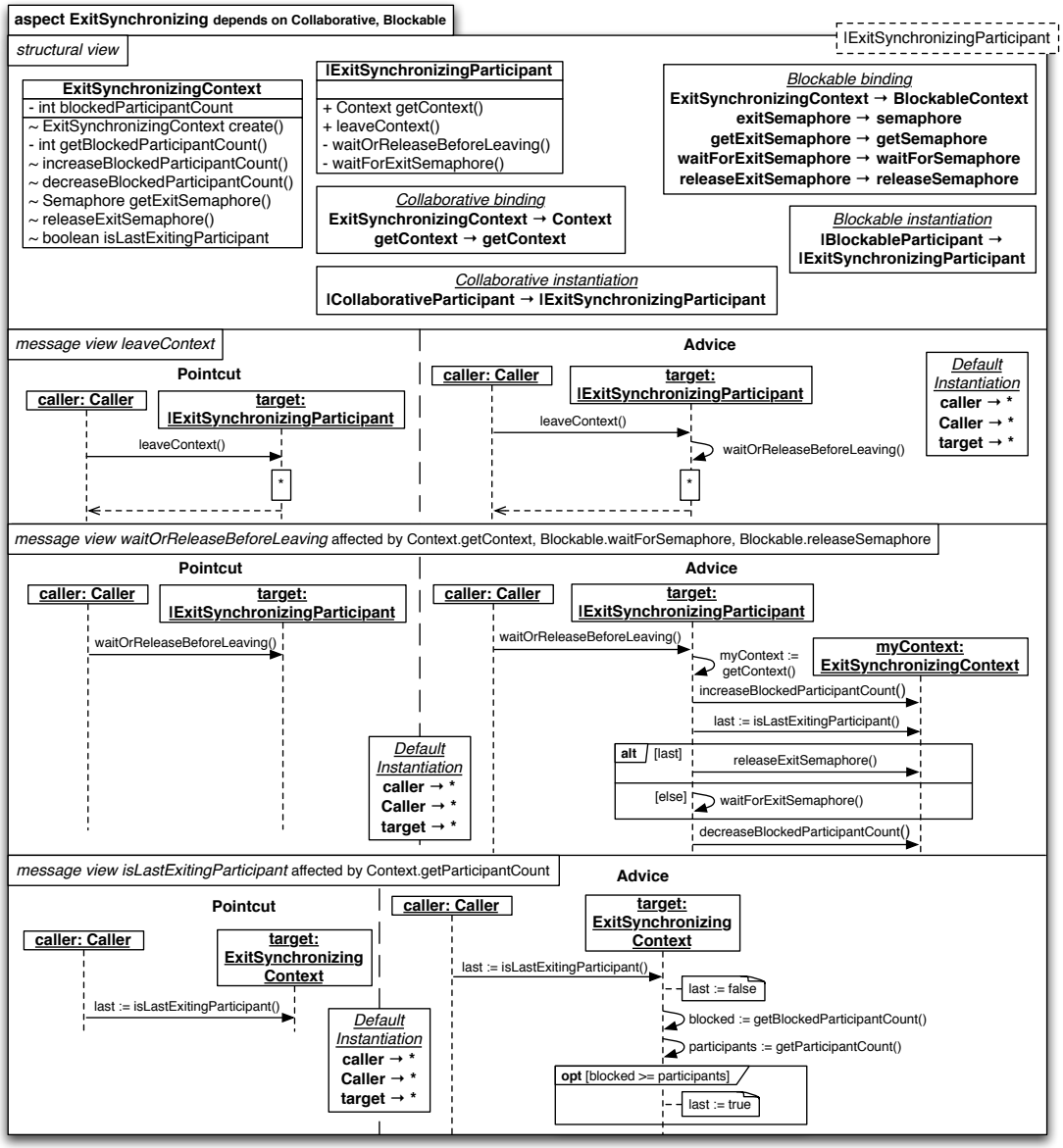

Figure 31: The *ExitSynchronizing* aspect blocks leaving participants until the last participant left.

However, the message view of *isLastEnteringParticipant* is not completely analogous to that of *isLastExitingParticipant*. Instead of comparing the number of waiting participants with the overall number of participants we check whether more participants than the minimal number of participants are waiting.

4.1.10 *Closable*

Refuse participants if a context is closed explicitly or automatically.

The *Closable* aspect (Figure 33) extends the aspect *Collaborative* and introduces a possibility to close a context as a mean of preventing participants from joining it. Furthermore the notion of an upper bound of participants is introduced such that the context is automatically closed once this bound is reached. In the structural view we introduce a boolean variable *closed* and an integer variable *maxParticipantCount* into the class *ClosableContext*. By initializing this variable to *Integer.MAX_VALUE* we define that per default a context will not be closed automatically. Apart from getter and setter methods for these variables we introduce a method *isLastAllowedParticipant* into the class *ClosableContext*.

The existing method *joinContext* is advised in a message view such that its original behavior is only executed if a call to *isClosed* returns *false*. This simple advice prevents participants from joining closed contexts. In a message view for the method *addParticipant* we specify that we check whether we added the last participant that is still allowed to join after the execution of the original behavior of *addParticipant*. If this is the case we automatically close the context by calling *setClosed* with the boolean constant *true* as parameter.

The newly defined method *isLastAllowedParticipant* is similar to the method *isLastEnteringParticipant* of the *EntrySynchronizing* aspect and it is detailed in a message view. We assume that the recently added participant was not the last by creating a boolean return variable that we initialize to *false*. Then we obtain the overall number of participants and the maximal number of participants that is allowed for this context. If both sizes are the same, we change the value of our return variable to *true*.

4.1.11 *SpawnSupporting*

Participants can spawn new participants that automatically become siblings.

The aspect *SpawnSupporting* as shown in Figure 34 gives a context participant the possibility to spawn further participants that automatically participate in its context thereafter. We introduce an integer variable *spawnedParticipantCount* and its methods *getSpawnedParticipantCount* and *increaseSpawnedParticipantCount* into the class *SpawnSupportingContext*. Additionaly, the methods *createNewParticipant* and *spawnParticipant* and a boolean variable

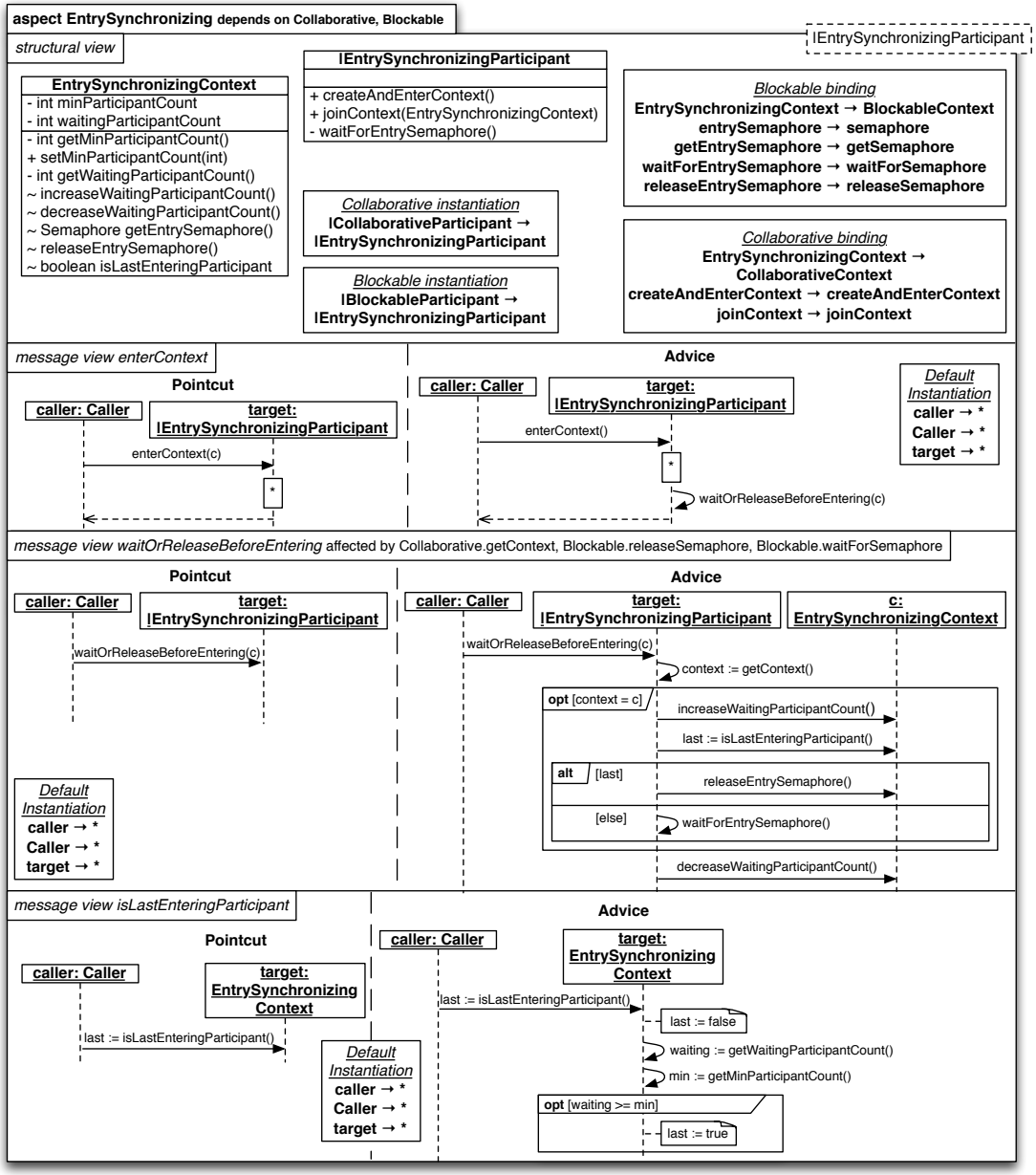

Figure 32: The *EntrySynchronizing* aspect blocks entering participants until a number of minimally required participants is reached.

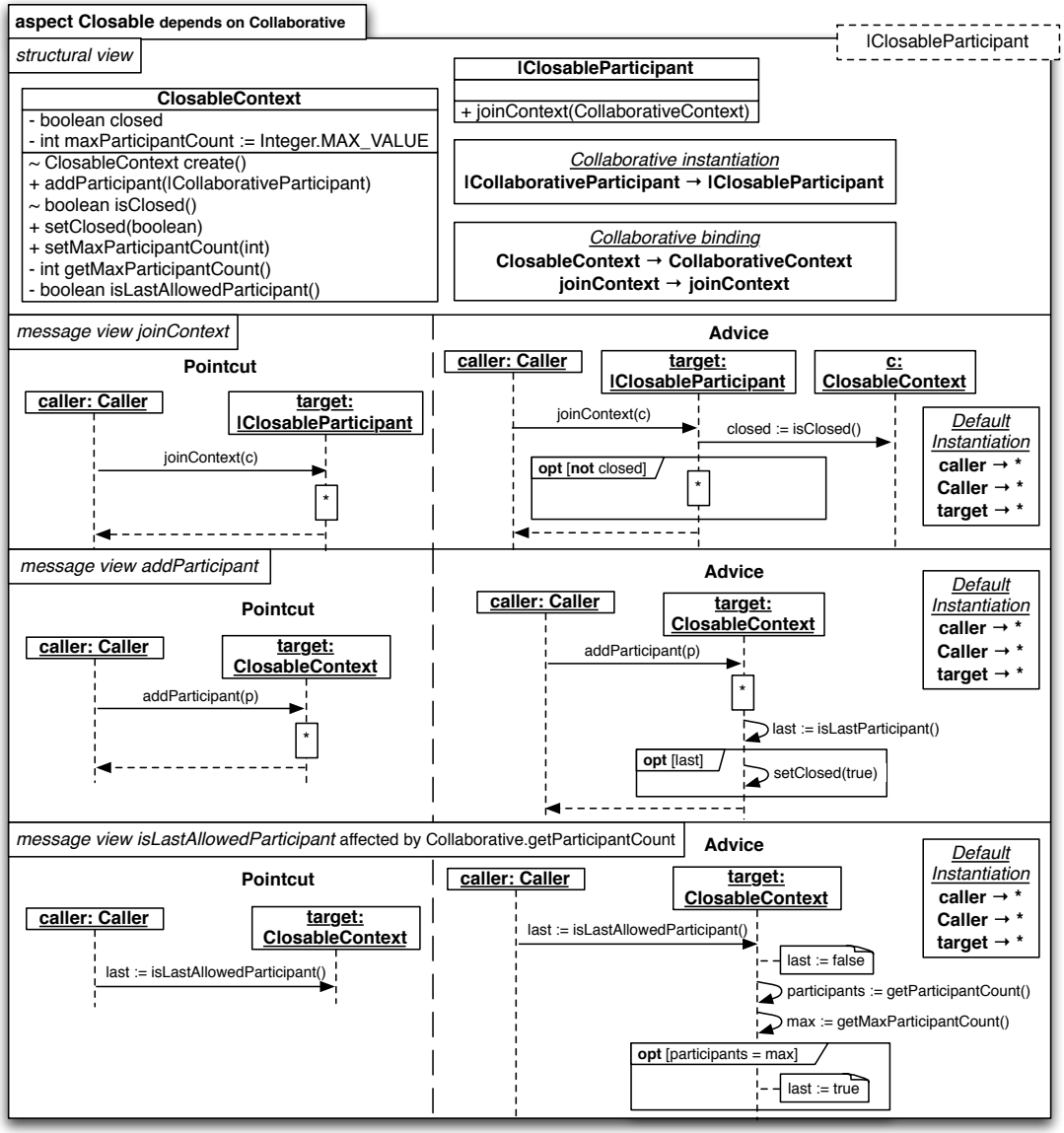

Figure 33: The *Closable* aspect allows explicit and implicit closure of contexts by specifying a maximal number of participants.

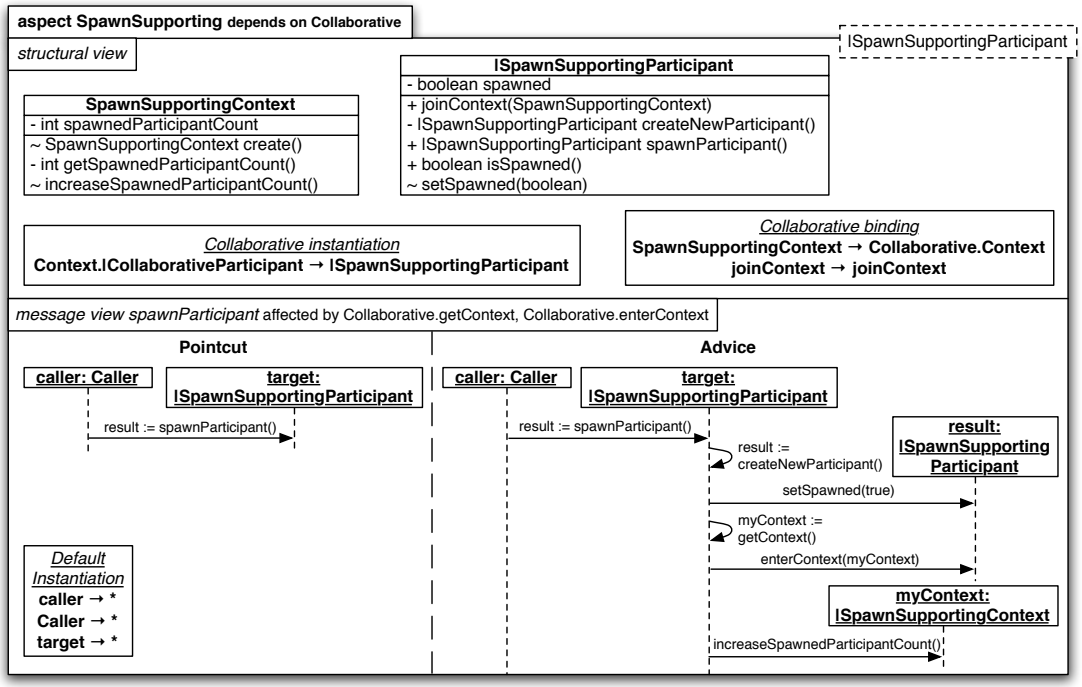

Figure 34: The *SpawnSupporting* aspect allows the creation of new participants that are automatically joining their creator's context.

spawned with its getter and setter methods are introduced into the class *SpawnSupportingParticipant*.

The detailed behavior of spawning a participant is illustrated in the message view of *spawnParticipant*: after creating a new participant we mark it as spawned and retrieve the actual context of the spawning participant. We make the newly spawned participant enter it and increment the number of spawned participants.

4.1.12 *OpenMultithreadedTransaction*

Combining all other aspects in a single interface.

The *OpenMultithreadedTransaction* aspect as modeled in Figure 35 combines the functionality provided by all other aspects of our framework into one aspect and contains no own additional logic or structure. In its structural view only the methods that should be available to the user of the framework are exposed and linked using binding directives.

4.2 CONFLICT RESOLUTIONS

In Section 2.3.2 we introduced RAM's conflict resolution aspects that detect and resolve conflicts automatically using interference criteria that specify under which circumstances conflict resolutions are instantiated. Our Open Multithreaded Transactions model contains three conflict resolution aspects that coordinate the interaction between two existing aspects each.

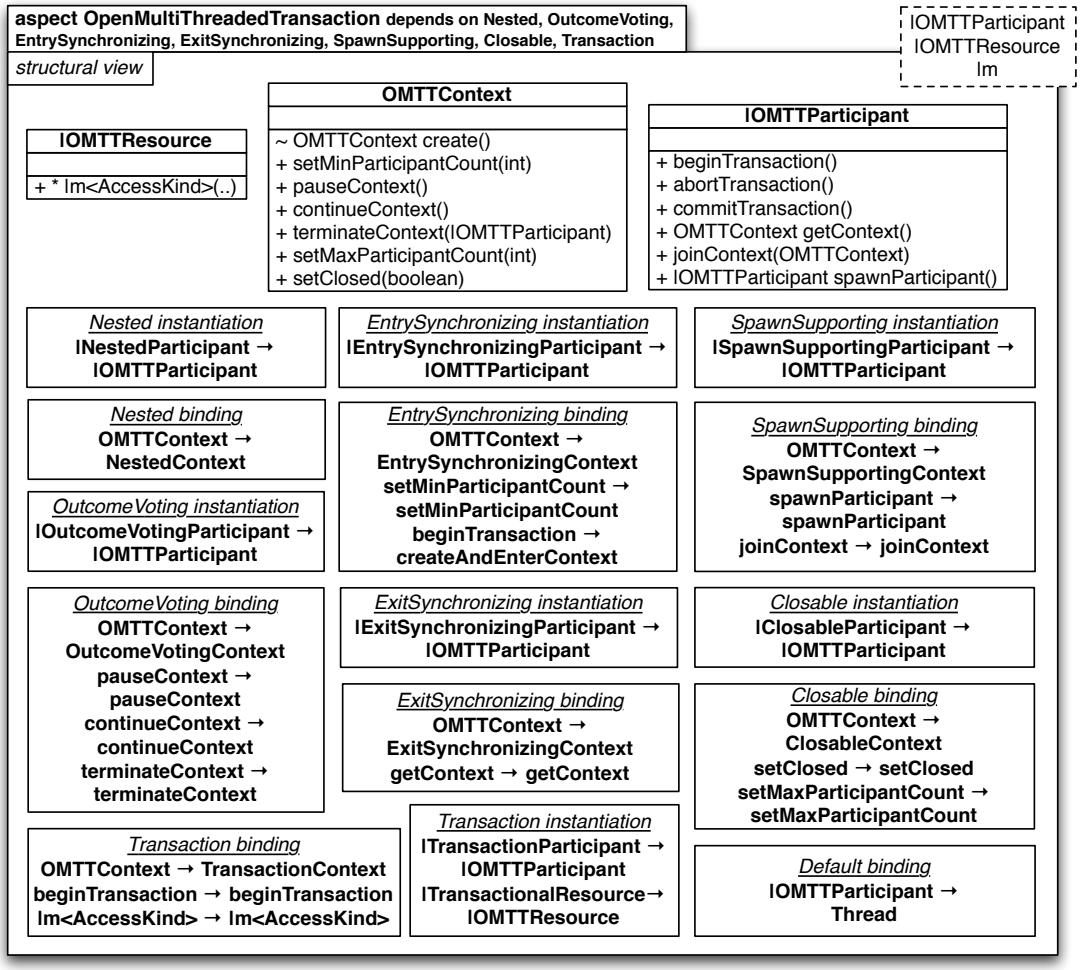

Figure 35: The *OpenMultithreadedTransaction* aspect combines the functionality of all other aspects and exposes their methods.

4.2.1 *Collaborative / Nested*

Participants are only allowed to join descendants of their current context.

The conflict resolution aspect *Collaborative / Nested* that we present in Figure 36 automatically applies whenever an object of the class *CollaborativeContext* is also an instance of the class *NestedContext*. It replaces the message view for *Collaborative.isAllowedToJoin* in order to allow a participant that is already associated to a context to join another context if the context that should be joined is a direct or indirect parent of the currently associated context.

The content of the message view for *isAllowedToJoin* begins with the initialization of a result variable to *false* and the retrieval of the associated context. If no context is associated, the result variable is immediately set to *true*. Otherwise, the parent of the context that should be joined is retrieved and a while loop is used in order to iterate over its ancestors until the context that is currently associated to the participant is found. If such an ancestor context is found, we know that we try to join a child of the current context and change the result variable to *true*. Otherwise, we keep on climbing up the context hierarchy as long as a parental context that differs from the current context exists. If the loop terminates at a context that has no parent context without ever inspecting the context that should be joined in the hierarchy, then the result variable that still evaluates to *false* is returned because a context that is no descendant of the current context was passed as an argument.

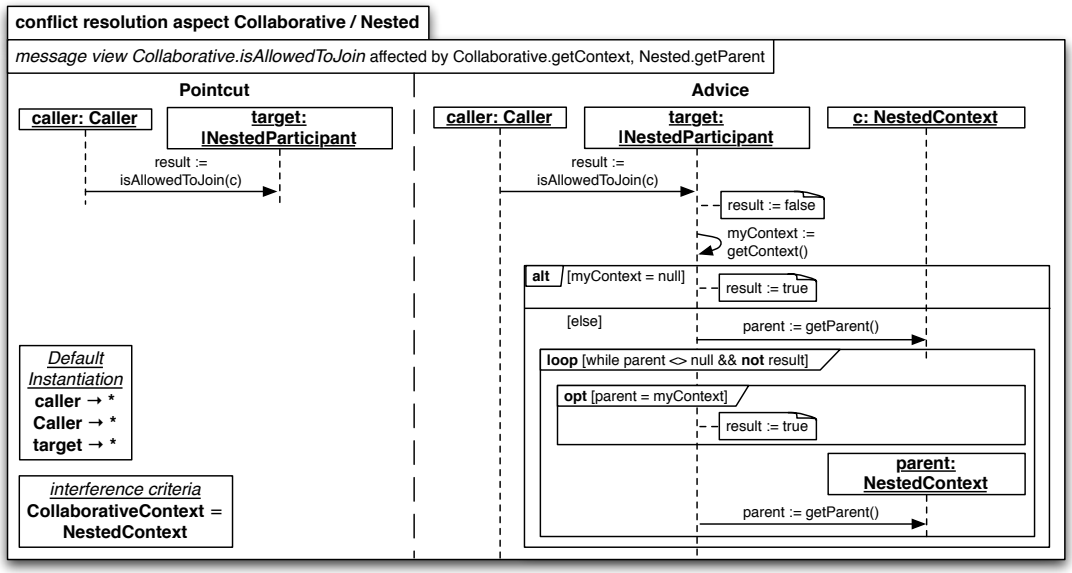

Figure 36: The conflict resolution aspect for *Collaborative / Nested* allows participants to join a context if the current context is an ancestor of the context that should be joined.

4.2.2 *ExitSynchronizing / SpawnSupporting*

Do not synchronize spawned participants upon leaving as they terminate afterwards.

In order to allow spawned participants to leave a context without being blocked in presence of the *ExitSynchronizing* aspect we modeled a conflict resolution aspect *ExitSynchronizing / SpawnSupporting* as shown in Figure 37. The message view for the method *waitOrReleaseBeforeLeaving* is modified in order to check whether a participant is spawned. Only for participants that have not been spawned we proceed with the original blocking behavior such that spawned participants are able to leave a context without beeing blocked.

As indicated in Section 4.1.8 we defined a separate method *waitOrReleaseBeforeLeaving* even if it is only used once in order to model the skipping of the blocking behavior concisely. RAM would also give us the possibility to include the blocking behavior directly into the method *leaveContext* and to work with a pointcut that contains the blocking behavior and replaces it with nothing in case of a spawned participant. As this would lead to an unnecessary duplication of the blocking behavior and a far more bigger conflict resolution aspect we decided to create the new method *waitOrReleaseBeforeLeaving*.

The message view for the method *isLastExitingParticipant* redefines this method in order to subtract the number of spawned participants from the overall number of participants during the comparison with the number of blocked participants. If the sum of blocked and spawned participants equals the overall number of participants we know that every participant that was not spawned is currently blocked. This means that we can change the result variable to *true* in order to indicate that the last participant tried to leave the context.

4.2.3 *Closable / SpawnSupporting*

Exclude spawned participants from the maximal number of allowed participants.

The conflict resolution aspect *Closable / SpawnSupporting* that we present in Figure 38 excludes spawned participants from the condition for the automatic closing of a context. This is achieved by a redefinition of the method *isLastAllowedParticipant*. Similarly to the *isLastExitingParticipant* method of the *ExitSynchronizing / SpawnSupporting* conflict resolution aspect, we retrieve the number of spawned participants and subtract it from the number of overall participants. The resulting number is compared with the maximal number of allowed participants and the result is only set to *true* if they are equal. This means that a context is automatically closed when the number of participants that have not been spawned reaches the maximal number of allowed participants.

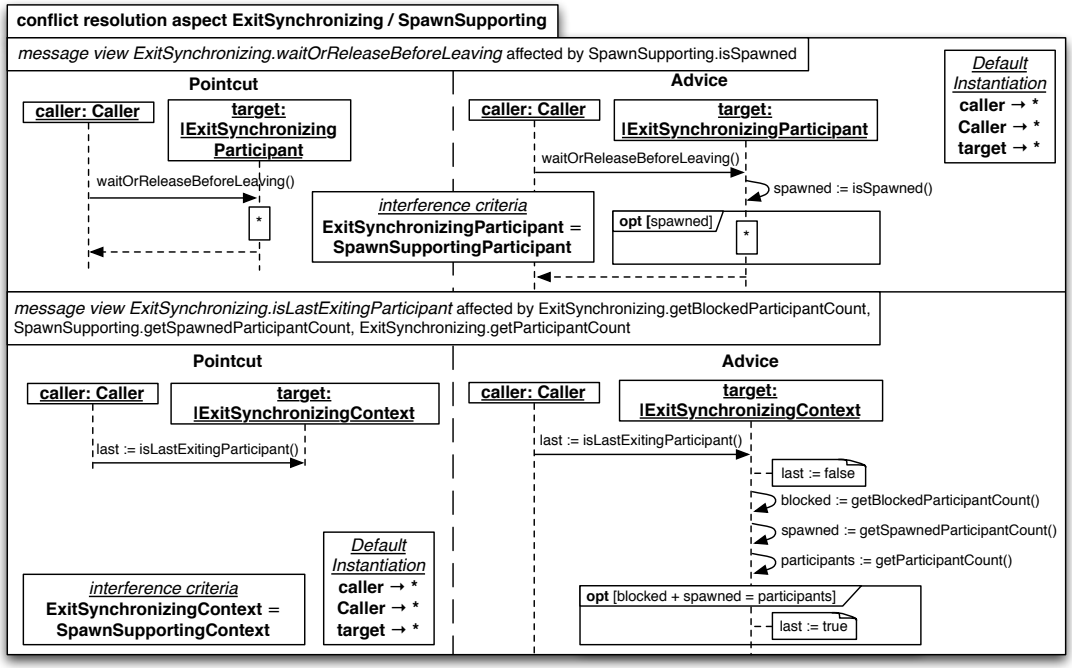

Figure 37: The conflict resolution aspect for *ExitSynchronizing / SpawnSupporting* excludes spawned participants from exit synchronization as they automatically terminate upon leaving.

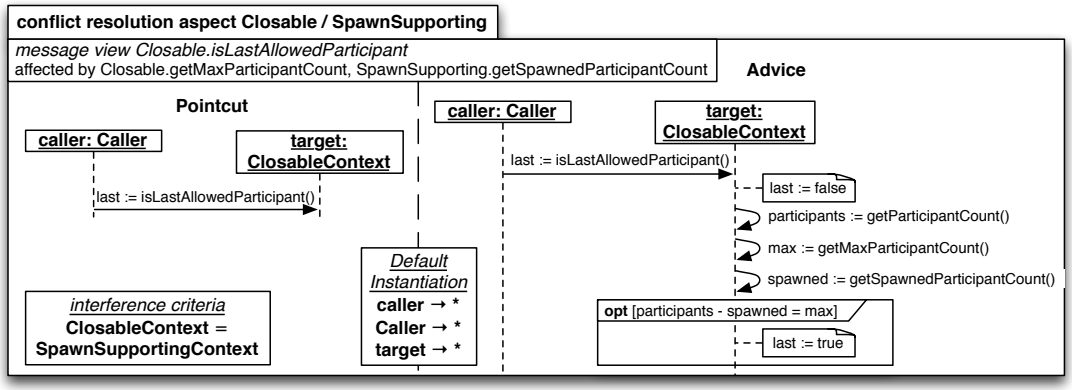

Figure 38: The conflict resolution aspect for *Closable / SpawnSupporting* determines if the maximal number of allowed participants is reached without counting spawned participants.

5 MAPPING REUSABLE ASPECT MODELS TO ASPECTJ

A general purpose mapping that can be used as a draft for a future code generator.

The main purpose of the models that we presented in the two previous chapters is to serve as a case study that helps us to define a mapping from Reusable Aspect Models to AspectJ. In order to ease the understanding and analysis of our mapping we will describe it in detail using examples taken from the AspectOPTIMA case study. Although we applied our mapping manually we hope that it can serve as a starting point for the implementation of a code generator. Wherever possible we tried to prefer a more general solution that can be used for other models as well over an adapted solution that makes use of properties of this specific case study. Therefore the code that we obtained by applying our mapping has not to be seen as an attempt to implement this specific case study but as an illustrating example of our general purpose mapping.

The next section presents the overall structure of the mapped code and the techniques that we used to obtain it. Subsequently we will describe the mapping along the different views of the modeling technique Reusable Aspect Models.

5.1 PRINCIPLES & OVERALL STRUCTURE

5.1.1 *Principles*

The main goals during the development of our mapping where to

- preserve the identity and modularity of modeled concerns
- reduce the necessity of manual implementation efforts
- produce code that can be reused with minimal effort
- maintain the traceability of concerns
- lay the foundations for an evolvable and reversible mapping that could one day be used for round-trip engineering

5.1.2 *Overall Structure*

RAM aspects as Java packages.

As every aspect in RAM corresponds to a package of classes we decided to maintain this structure within the code too. As a starting point a main package for every RAM project is created. Within the main package of every project we create the two

packages `aspects` and `conflictresolutions`. For every ordinary RAM aspect we create a subpackage in the package `aspects` bearing the name of the aspect in lowercase letters and containing all artifacts of the RAM aspect. The artifacts of conflict resolution aspects are located in a subpackage of `conflictresolutions` whose name is the concatenation of both involved aspects in alphabetic order and lowercase letters.

Directly in the main package we create three AspectJ aspects `AnnotationInheritance`, `ConfigurationEnforcement`, and `AspectPrecedence`. The latest defines precedence rules by listing all aspects in the order of reuse starting with the aspect at the root of the directed acyclic reuse graph thus giving it highest precedence. The role and content of the two other aspects will be discussed in Section 5.2.1.6 and Section 5.4.

5.2 ORDINARY ASPECTS

Reusable Aspect Models consist of ordinary aspects enclosing three different types of views and of conflict resolution aspects that have the same structure and an additional feature that activates them if their interference criteria are met.

5.2.1 *Structural View*

The structural view is the core of each RAM aspect as it contains the classes of the aspect with its attributes, methods and associations. It is based on class diagrams of the Unified Modeling Language and supports additional features such as declaration and initialization of methods or classes as mandatory instantiation parameters, and binding of elements across aspects.

5.2.1.1 *Complete Classes*

If a class has a name that does not start with an "|" and its method section lists a constructor it is considered a complete class in RAM. A constructor is a method named `create` that returns its enclosing type.

Unobtrusive and automatic reuse of Java classes and interfaces.

REUSING THE JAVA LIBRARY In order to allow modelers to reuse Java's library classes without polluting the model with artifacts that are specific to this language and of little value for other target languages we decided to use an automatic reuse mechanism: We maintain a list of supported library classes and interfaces (shown in Figure 92 in the appendix) and associate every interface to a default implementation class. For every complete class in the model we check whether it has a name that is identical to the name of a Java class or interface on the library list.

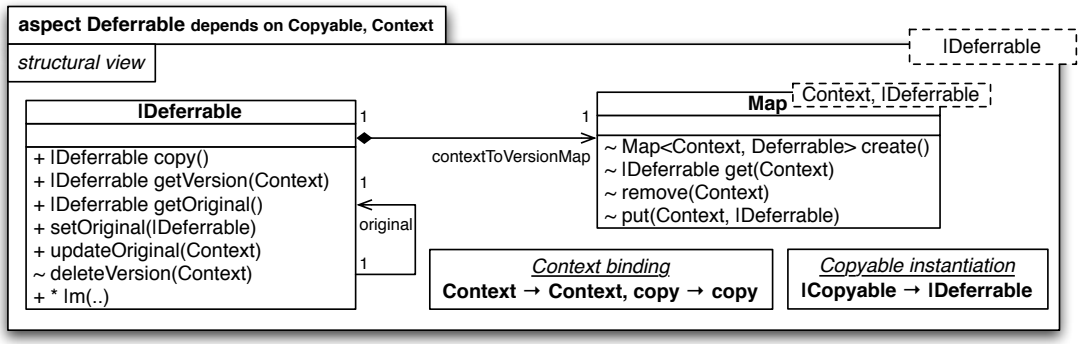

Figure 39: Structural view of the *Deferrable* aspect showing the unobtrusive reuse of JAVA's class `Map`.

If this is the case we inspect the methods mentioned in the structural view and compare them to the methods of the JAVA library class or interface. In case of an interface we match the constructor methods with the constructors of the default implementation that is linked to this interface.

The complete class is mapped to this library class or interface if the modeled class contains only methods with signatures that match methods of the library class or interface. Parameters that are defined at the modeled class using the template parameter visualization of the UNIFIED MODELING LANGUAGE need to match the type parameters of the library class or interface. If we discovered a mapping to an interface, then we use the associated default implementation class whenever a constructor is called but all associations or method variables have the general type of the interface in order to keep the code as independent as possible from the chosen default implementation class.

An example where we automatically reuse the JAVA interface `java.util.Map` together with its default implementation `java.util.HashMap` is shown in Figure 39 and Figure 40. The methods `get`, `remove`, and `put` match the method signatures of `java.util.Map` and the constructor matches a constructor in the corresponding default implementation class `java.util.HashMap`. Note that the mapping of the template parameters `Context` and `Deferrable` to the type parameters `K` and `V` is taken into account during the signature matching process.

Library classes and interfaces are also automatically reused if an argument or return type of a method refers to a class that is neither modeled in the enclosing RAM aspect nor in any other RAM aspect of the same project. In such a case we use the first class or interface with the same name on our library list. An example for this is the class `java.lang.reflect.Method` which is used in the aspect *Traceable* as an argument for the method `createTrace` and as an argument for the constructor of the class `Trace`. The message view of this method is also discussed in

```
public interface Deferrable extends Copyable {
  // public method signatures from structural view
  Deferrable getVersion(Context context);
  Deferrable getOriginal();
  void setOriginal(Deferrable original);
  void updateOriginal(Context context);
}
aspect DeferrableAspect {
  // bind annotations to mandatory instantiation parameters
  declare parents : @DeferrableClass * implements Deferrable;
  // attributes & associations from structural view
  private Map<Context, Deferrable> Deferrable.contextToVersionMap =
   new HashMap<Context, Deferrable>();
  private Deferrable Deferrable.original = null;
  // declared methods without message views
  public Deferrable Deferrable.getOriginal() {
    // auto-generated getter implementation
    return this.original;
  }
  public void Deferrable.setOriginal(Deferrable original) {
    // auto-generated setter implementation
    this.original = original;
  }
  ...
}
```

Figure 40: Mapped code for the structural view of the complete class `Deferrable` (see Figure 39) showing interface methods, the type hierarchy modification, fields, and default implementations for methods without message views.

Section 5.2.3.5 in order to explain concrete calls to constructors along the Figures 53 and 54.

Interfaces and inter-type declarations even for complete classes.

MAPPING USER CREATED CLASSES If a *complete class* in the aspect model is not recognized as an instance of an existing JAVA class, a JAVA implementation is generated from scratch. The straightforward idea of mapping the complete class to a standard JAVA class is not sufficient, as it is possible that the modeled class has to be merged with other classes as an effect of binding directives. If a complete class would be a standard JAVA class we would be unable to implement such a merge as JAVA does not support multiple inheritance.

But as it is possible to make a JAVA class implement multiple interfaces we can use an indirect introduction mechanism employing interfaces and inter-type declarations in order to implement the merging of classes: for every complete class of the model we create a new public JAVA interface with the same name and an ASPECTJ aspect within the file of this interface. In this aspect we introduce fields and methods into the interface using ASPECTJ's inter-type declaration mechanism as described in the following sections. How these interfaces make it possible to merge classes when we map binding directives to code is discussed in detail in Section 5.2.1.7.

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface DeferrableClass {
  // empty
}
```

Figure 41: JAVA annotation for the mandatory instantiation parameter `Deferrable` (see Figure 39) restricted to classes with a `Target(ElementType.TYPE)` annotation.

As interfaces cannot be instantiated in JAVA we create a public class that implements the corresponding interface for every complete class of a RAM aspect. The name of this implementation class is the name of the modeled class with an additional "Impl" appended to it. Implementation classes are only needed to provide instantiation facilities and are empty except for possible constructors that we discuss in detail in Section 5.2.1.5.

A special case of user created complete classes are enumerations. For every modeled enumeration we create a public JAVA enumeration that simply lists the enumeration literals that are given in the model in uppercase letters.

5.2.1.2 *Incomplete Classes*

Incomplete classes need to be instantiated or bound.

In RAM classes that have no constructor are considered incomplete and need to be completed during the weaving process. This can either be done by prepending an "|" to their name in order to mark them as mandatory instantiation parameters or by including a binding directive that binds them to a complete class. In both cases we skip the library reuse matching that we described above and implement incomplete classes almost the same way we implement complete classes that have not been matched to a library class or interface. The only difference is that we omit the creation of an implementation class as incomplete classes cannot be instantiated.

If an incomplete class is marked as a mandatory instantiation parameter we create a JAVA annotation that has the same name and the word "Class" appended to it. Figure 41 presents such an annotation for the mandatory instantiation parameter `Deferrable`. Note that we make sure that only classes are instantiated to this parameter by annotating the annotation itself with `@Target(ElementType.TYPE)`.

We use annotations internally in order to map instantiation directives and externally to give users the possibility to instantiate parameters by annotating their classes. Instantiation directives are discussed in detail in Section 5.2.1.6 and Chapter 6 explains how to use the mapped code.

```
aspect ClosableContextAspect {
  ...
  // attributes & associations from structural view
  private boolean ClosableContext.closed = false;
  private int ClosableContext.maxParticipantCount = Integer.MAX_VALUE;
  ...
}
```

Figure 42: Mapped code for attributes and relations of the `ClosableContext` class (see Figure 33) showing default and manual initialization of resulting fields.

5.2.1.3 *Attributes*

Attributes of classes that are mentioned in the structural view of a RAM aspect are implemented as plain Java fields that are introduced into the interface that corresponds to the class using AspectJ's inter-type declaration. Access modifiers $+$, $\sim$, and $-$ are canonically mapped to `public`, `protected`, and `private`. But as we are convinced that it is good object-oriented style to declare fields private you will find no example of the other two access modifiers for fields in our case study.

In order to reduce manual implementation refinements to the required minimum we provide modelers with the possibility to initialize simple attributes by appending ":=" to an attribute identifier. The resulting initialization statement is the unmodified string that follows ":=" in the model and therefore we can support initializations like `Integer.MAX_VALUE` as used in the `ClosableContext` class shown in Figure 42.

5.2.1.4 *Associations*

Java fields and HashSets for single and multiple associations.

Associations between classes in Reusable Aspect Models are implemented using Java fields whose names correspond to the given role names and whose types depend on the specified multiplicity. For this reason our mapping can only yield correct code for associations with explicit multiplicities and at least one role name. If an association has only one role name we interpret it as unidirectional and introduce a field with the same name into the interface that corresponds to the class at the other end of the association. A second role name for the other direction means a bidirectional association and results in additional code that is obtained exactly the same way.

If the multiplicity that corresponds to a role is specified as "0..1" in a RAM aspect, then the type of the resulting field is simply the type of the associated class and we initialize the field explicitly to `null`. A multiplicity of "1" results in the same type but the field is now initialized using the parameterless constructor of the associated class. If an association has a multiplicity of "0..*" or

"1..*" the type of the corresponding field is a `java.util.Set` that is parameterized using the type of the associated class.

Our mapping could easily be extended to support multiplicities of type "n" or "m..n" by creating n fields in the first case and a `HashSet` whose `add` method is advised in order to preserve multiplicity constraints. As this is a straightforward extension that was not necessary for the AspectOPTIMA case study we will not discuss it in detail. Note that all fields that correspond to associations are declared using the `private` access modifier in order to prevent unauthorized access.

5.2.1.5 *Methods*

Methods that are mentioned in the structural view of a Reusable Aspect Model but have no own message view can be mapped to two types of Java methods: If their functionality can be derived from their name we create a complete default implementation. Otherwise we only create method-stubs that need to be completed manually after code generation. In both cases we mention a method in the Java interface that corresponds to its enclosing type if its access modifier is `public`.

Methods with signatures that follow common patterns can be implemented automatically.

DEFAULT IMPLEMENTATIONS FOR COMMON METHODS The probably most common type of methods whose functionality can be completely inferred from their name are getter and setter methods for attributes or associations. If the name of a parameterless method in RAM is "get" followed by the capitalized name of an attribute or association of the enclosing type and its return type is the type of this attribute or associated class we consider it a getter method. For such methods we create a default implementation that returns the value of the corresponding field. Note that for attributes of type boolean a getter method starts with "is" instead of "get". A method whose name is "set" followed by the capitalized name of an attribute or association of the enclosing type, that takes exactly one parameter that matches the type of this attribute or association and whose return type is void is considered a setter method. It's implementation is a single statement that assigns the value of the passed argument to the corresponding field.

For attributes of type int we support two more special method types to ease the manipulation of the corresponding values. A method whose name starts with "increment" or "decrement" followed by the capitalized name of an int attribute of the enclosing type and whose return type is void is implemented as a single statement that increases or decreases the corresponding value by one. We use various methods of this kind in our Open Multithreaded Transaction model in order to keep track of the number of participants that fulfill a certain property. Their use shrank

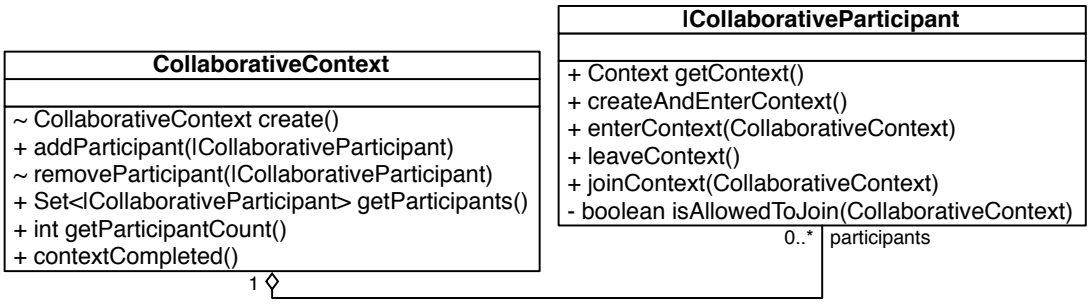

Figure 43: Extract of the structural view of the *Collaborative* aspect showing an association with multiplicity > 1 and corresponding modification and retrieval methods.

our message views as they make it possible to express these operations with one instead of three messages.

Associations with multiplicity > 1 can have three more method types for which we define automatic implementations that ease the addition, removal, and counting of elements. If a method returns nothing, takes exactly one argument of the same type as an associated class with multiplicity > 1 and its name is "add" followed by the capitalized name of the association role without the last letter being a "s", then we consider it an adder method. Such methods are implemented with a single call to the `add` method on the `Set` that corresponds to the association. The same pattern with a "remove" instead of an "add" describes remover methods that are implemented with a single call to the `remove` method on the corresponding `Set`. A method that takes no arguments, returns an int and whose name is "get" followed by the capitalized name of the role of an association with multiplicity > 1 that is in turn followed by the string "Count" is considered a counter method. The implementation of these methods directly returns the result of a call to the `size` method of the `Set` that corresponds to the association.

Note that the pattern for getter methods that we described above is modified slightly for associations with multiplicity > 1: The return type of such method is not the type of the associated class itself but a `Set` that contains elements of this type. Figure 43 and Figure 44 demonstrate how an association with multiplicity > 1 is implemented and show the default implementation for the corresponding adder, remover, getter and counter methods.

Implementation of field initializing constructors.

SPECIAL ROLE OF CONSTRUCTORS Constructor methods have a special syntax in REUSABLE ASPECT MODELS and JAVA so that they require a specific mapping. In RAM a method can be identified as a constructor when it is named `create` and returns its enclosing type. In JAVA, however, the return type of a constructor is implicit and must not be mentioned but the method name needs to be equivalent to the name of the enclosing type.

```
public interface CollaborativeContext extends Context {
  // public method signatures from structural view
  void addParticipant(CollaborativeParticipant participant);
  Set<CollaborativeParticipant> getParticipants();
  int getParticipantCount();
}
aspect CollaborativeContextAspect {
  // attributes & associations from structural view
  private Set<CollaborativeParticipant> CollaborativeContext.participants =
   new HashSet<CollaborativeParticipant>();
  // declared methods without message views
  public void CollaborativeContext.addParticipant(CollaborativeParticipant participant) {
    // auto-generated adder implementation
    this.participants.add(participant);
  }
  void CollaborativeContext.removeParticipant(CollaborativeParticipant participant) {
    // auto-generated remover implementation
    this.participants.remove(participant);
  }
  public Set<CollaborativeParticipant> CollaborativeContext.getParticipants() {
    // auto-generated getter implementation
    return this.participants;
  }
  public int CollaborativeContext.getParticipantCount() {
    // auto-generated counter implementation
    return this.participants.size();
  }
  ...
}
```

Figure 44: Mapped code for the structural view of the complete class `CollaborativeContext` (see Figure 43) showing the implementation of an association with multiplicity 0..∗ and the corresponding modification and retrieval methods.

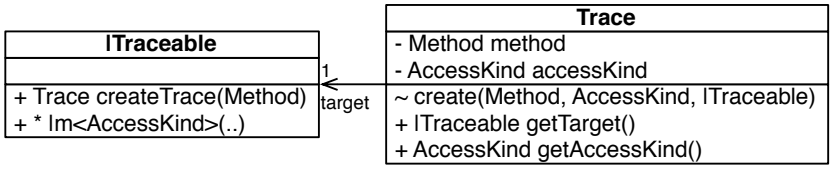

Figure 45: The *Trace* class taken from the *Traceable* model contains fields and a constructor that initializes them.

```
public interface Trace {
  ...
}
aspect TraceAspect {
  ...
  // declared methods without message views
  ...
  TraceImpl.new(Method method, AccessKind accessKind, Traceable target) {
    // auto-generated constructor using fields
    this();
    this.method = method;
    this.accessKind = accessKind;
    this.target = target;
  }
  ...
}
```

Figure 46: Intertype declaration of a constructor of the class `Trace` (see Figure 45) that initializes fields with given parameters.

We implement a parameterless constructor by adding an empty JAVA constructor to the implementation class that corresponds to the modeled class. Note that this in only redundant if the constructor is modeled with a public access modifier as JAVA's implicit default constructor is public.

Constructors that take arguments are matched against the attributes of the enclosing class. If a constructor takes only arguments that match the types of attributes or associated classes and at most one argument for each attribute and associated class we create a default implementation that initializes the corresponding fields with the values of the arguments. As these fields might be private we need to define the constructor in the aspect that corresponds to its enclosing type and not in the implementation class[1]. An example of such a constructor is shown in Figure 45 and the resulting code that initializes the fields of the modeled attributes is presented in Figure 46.

METHODS AS MANDATORY INSTANTIATION PARAMETERS

Methods that are marked as mandatory instantiation parameters yield no direct implementation but a JAVA annotation that can be used to instantiate the parameter. Analogous to the annota-

1 For fields and methods that were introduced using ASPECTJ's inter-type declaration mechanism the access modifiers refer to the defining aspect and not to the enclosing type.

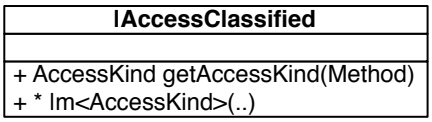

Figure 47: The class *AccessClassified* from the aspect of the same name showing the use of methods as parameterized mandatory instantiation parameters.

```
@Target({ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface AccessClassifiedMethod {
  AccessKind value();
}
```

Figure 48: JAVA annotation for the mandatory instantiation parameter method in the class `AccessClassified` (see Figure 47) showing the use of annotation elements for parameters.

tion for parameter classes that we described in Section 5.2.1.2 we create an annotation with the name of the enclosing class followed by "Method". We use the name of the enclosing class as the parameter names are mostly single letters that reveal little information. If a class contains more than one mandatory instantiation parameter method we need to use the name of the method itself or enumerate the parameters but this case never occurred in the ASPECTOPTIMA case study. We restrict the target type using the annotation `@Target(ElementType.METHOD)` in order to make sure that only methods can be marked using such annotations.

If the parameter itself is parameterized we add corresponding methods for each parameter to the JAVA annotation. An example of such a method parameter is shown in Figure 47 and Figure 48.

METHOD STUBS FOR REMAINING METHODS If a method has no separate message view and is neither a constructor, nor a mandatory instantiation parameter nor a method whose implementation can be derived from its signature we introduce an empty method stub. If the method has a return type we return the default constant of this type in order to keep the generated code compileable. To help the programmer spot such incomplete methods stubs we add a todo comment that marks this method stub as incomplete.

5.2.1.6 *Instantiation Directives*

Type and annotation inheritance implement parameter instantiation.

RAM's instantiation directives are used to assign classes or methods to mandatory instantiation parameters and are implemented by type hierarchy modifications or annotation inheritance. Let us consider an aspect A that depends on an aspect B and whose structural view contains a class C and an instantiation directive $D \rightarrow C$ where D is a mandatory instantiation parameter of B.

This directive is mapped to an inheritance relation between *D* and *C* such that *D* extends *C*.

In order to treat classes that are indirectly used as a value of a mandatory instantiation parameter by means of an instantiation directive the same way as classes that were directly used as a parameter by means of an annotation we inherit annotations according to instantiation directives. Therefore the directive $D \rightarrow C$ from our previous example results in a statement `declare @type: @DClass : @CClass` that tells the AspectJ compiler to automatically add the annotation `CClass` to each class that has been annotated using `DClass`. Annotation declarations like this are located in the AspectJ aspect `AnnotationInheritanceAspect` in the main package of the project.

If a parameterized method parameter |*m*<*ParamType*> in a class *D* is instantiated using another parameterized method parameter |*n*<*ParamType*> in a class *C*, then we use a similar mechanism to inherit the annotation and its parameter values. For every possible parameter value `v` we add the AspectJ statement `declare @method : (@DMethod(v) * *.*(..)) : @CMethod(v);` to the `AnnotationInheritance` aspect. Note that this means that we can only generate correct code for mandatory method instantiation parameters that are itself parameterized with a finite and enumerable parameter or not parameterized at all.

5.2.1.7 *Binding Directives*

Complete classes or methods can be bound to other classes or methods and are implemented through type inheritance or delegating methods. An aspect *A* that depends on an aspect *B* and whose structural view contains a class *C* and a binding directive $C \rightarrow D$ where *D* is a class in *B* means that the two classes *C* and *D* should be merged. As all structure and logic is introduced into the corresponding interfaces it is sufficient to define that *C* extends *D*. If a method *m* of *C* is bound to a method *n* of *D* we need to consider two possible cases: If *m* and *n* have the same name the binding already took place as *C* inherited all methods of *D*. In case of different names we implement the method *m* with a single call to *n* while preserving possible parameters.

5.2.2 *State View*

RAM's state views allow a modeler to define an invocation protocol to which the modeled message views must conform. As code generation usually occurs after a model checking step one might argue that all message views for which code should be generated already conform to the invocation protocol and therefore state views need not to be implemented. This is only partially true as a

system may produce invalid message sequences at runtime even if all message views conform to the invocation protocol.

Nevertheless state views are not necessary to realize a system with the functionality that was modeled in REUSABLE ASPECT MODELS and therefore we decided to exclude them from our general purpose mapping. Despite that, we believe that it would be possible and beneficiary to implement state views as additional runtime checks and we will sketch our idea for a possible mapping in the following. As a first step we would need to implement the data structures and transition mechanism of a general purpose state automaton. This infrastructure would need to be introduced into every object whose message invocations are restricted by a state view. As RAM's state views support parallel states we would require an implementation that allows us to pass from a set of states into another set of states. Using the specific information of each state view we would need to advise every execution of the methods that are used as transition labels. Before the execution of the original method we would need to verify that the object on which the method was invoked resides in an allowed state and then we would make it pass into the target state of the corresponding transition in the state view.

5.2.3 *Message View*

The behavior of methods that are declared in structural views of REUSABLE ASPECT MODELS is defined in message views that are mapped to various ways of defining and modifying JAVA methods.

In the following sections 5.2.3.1 to 5.2.3.4 we discuss the different settings in which message views can be defined and what structural implications this has for the implementation of their content. The detailed mapping for single messages and constructs within a message view is explained in Section 5.2.3.5.

5.2.3.1 *Defining New Methods*

New methods are injected into the interface of the enclosing class.

If a method of a class is detailed in a message view and a method with the same signature has not been declared in a class that is directly or indirectly merged with this class as a result of binding or instantiation directives, then we call this method newly defined. Such methods are implemented using ASPECTJ's inter-type declaration mechanism within the ASPECTJ aspect that corresponds to the interface of the enclosing class. The return type and parameter types for the introduced method are obtained from the corresponding structural view and the parameter names are mentioned in the message view's pointcut.

RAM allows arbitrary message patterns in the pointcut part of a message view but when methods are newly defined the pointcut is always of the following form: a single message labeled with the name of the method and names for passed arguments in round brackets that is sent from the lifeline of a caller object to an object of the type that is declaring the method. The sender and receiver objects may have arbitrary names but in the AspectOPTIMA case study we stuck to the names *caller* and *target*.

A default instantiation directive *caller* → *, *Caller* → *, *target* → * means that the advice will apply for callers of arbitrary type and in cases where the caller and target might be named differently. In order to be able to refer to the caller and target object in messages we need to give them names and cannot simply write e.g *:* for the caller. We could specify the type of the caller directly as * in the lifeline box as we do not refer to it later on but the concrete name *Caller* gives us the possibility to instantiate the type differently in a reusing aspect by writing *Caller* → *SomeType*. Such a restriction on the caller type would be implemented with a `call` pointcut in combination with a `this` pointcut in AspectJ .

5.2.3.2 *Advising Existing Methods*

Methods can be advised before, around, or after the original behavior.

The modification of a method that has been defined earlier in a Reusable Aspect Model is mapped to classical AspectJ code using a `before`, `around`, or after `advice`. In the pointcut of the message view such a modification is modeled the same way a first definition is modeled but an additional wildcard box on the lifeline of the target and a return message are added. The wildcard box represents all message sequences that are initiated by the target between the call and return of the method and therefore stand for the behavior of the method before the advice applies.

Depending on the use of this wildcard box in the advice part of the message view we map the modeled sequence to different types of dynamic AspectJ advice. If the original behavior occurs in between new messages or not at all we use an `around` advice. A sequence that starts with the wildcard box and is followed by new messages is mapped to an `after` advice. Finally a sequence that ends with the placeholder for the original messages is implemented using a `before` advice.

All advice types are implemented using an AspectJ advice declaration signature of the form `adivcePart : pointcutPart contextCollection`. In case of an `around` advice the advice part of the signature starts with the return type of the advised method. The name of the advice type follows after that and with the sequence `(AdvisingType targetName, ArgType argName, ...)` we conclude the advice part by declaring the objects that are needed in the context collection part. `AdvisingType` is the type of the class that inherits but advises the method and `targetName` is

the name for the concrete object of this type that was used in the model. The list of argument types is retrieved from the structural view and the argument names are retrieved from the message label in the pointcut part of the message view.

The pointcut part of the advice signature specifies that we want to advise the execution of the method whose name is given in the header of the message view. The exact form is `execution(returnType DeclType.mName(..))` which restricts the advice to the exact method in consideration. The return type can be retrieved from the corresponding structural view and the name of the declaring type can be discovered by searching through all classes that were directly or indirectly merged with the class that lists the method in the structural view. The sequence `(..)` after the name of the method means that we do not restrict the type and number of arguments in this part of the signature.

In the last part of the advice signature we collect context information in order to use it in the advice body. This part takes the form `&& target(targetName) && args(argName, ...)` , binds the variables that we defined in the first part, and restricts the application of the advice to targets of the advising type.

The wildcard box is an element that may only occur in message views that advise existing methods or methods that are mandatory instantiation parameters. Therefore we explain its implementation before we discuss the implementation of other elements in Section 5.2.3.5. If this placeholder for the original method behavior appears in the advice part on the same lifeline as it appears in the pointcut part we create a `proceed` statement that is parameterized with all arguments of the advice without any change. It is however possible to model the wildcard box on the lifeline of another object than the target object but this new target for the original behavior needs to be of the same type as the original target object. In such a case we produce the same `proceed` statement but replace the variable representing the original target with the variable of the object on which the wildcard box was placed. An example of such a replaced wildcard box is shown in the context of mandatory instantiation parameters in the next section using Figure 49 and Figure 50.

5.2.3.3 *Advising Mandatory Instantiation Parameters*

Advising mandatory instantiation parameter methods means advising unknown existing methods.

The advisory of a method that is a mandatory instantiation parameter is modeled and implemented in a very similar way to that for the advisory of existing methods. The only difference is that we cannot know in advance what name, return type and parameters such a method will have. Therefore the structural pattern for the pointcut and advice part are the same except for the fact that the label of the message that corresponds to the advised method does not list the name of the arguments but the

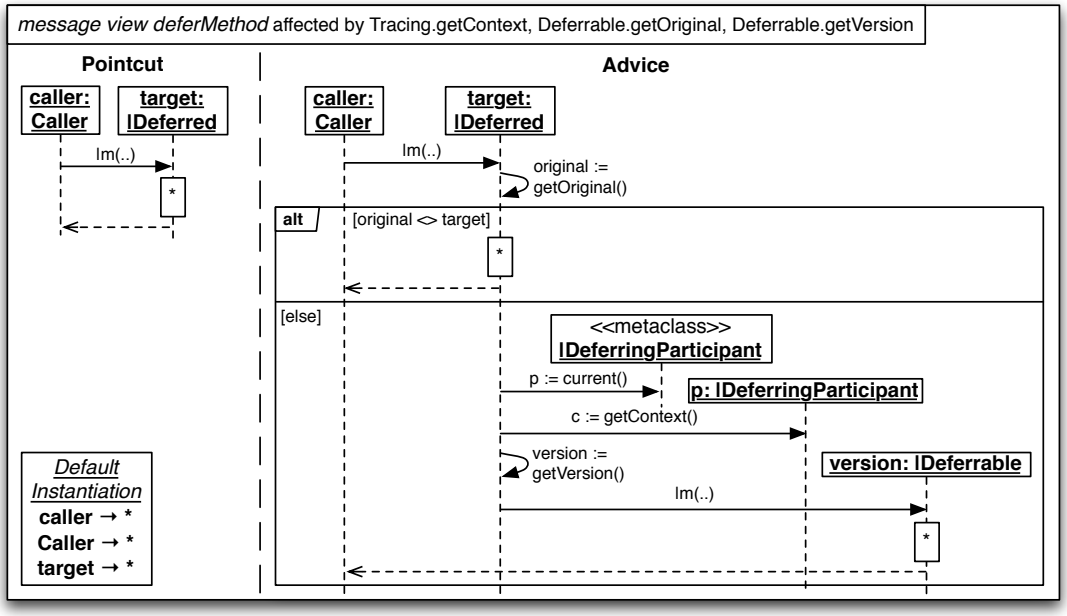

Figure 49: The message view *deferMethod* from the *Deferring* aspect advising a mandatory instantiation parameter.

wildcard characters `..` that represent all arguments regardless of their number, type or order.

The structure of the mapped implementation of such an advice is similar but the parts that were concerned with arguments are eliminated and the pointcut part changes. No longer do we mention the return type, declaring type, name and arguments of a concrete method. Instead, we specify that we want to advise every method that is marked with the annotation `AdvisingTypeMethod` that corresponds to the mandatory instantiation parameter by writing `execution(@AdvisingTypeMethod * *.*(..))`.

An example for a message view that advises a mandatory instantiation parameter is taken from the *Deferring* aspect and shown in Figure 49. The model for deferred methods and the mapped code presented in Figure 50 also demonstrate the use of a wildcard box as a placeholder for the original method behavior. Note that the advice is always located in the aspect that corresponds to the class in which the method is inherited and advised. In order to preserve the explanatory name of the message view we annotate the advice using the annotation `AdviceName`.

5.2.3.4 *Mapping Behavior Involving Generic Types*

Whenever a method parameter makes use of a generic type we cannot use our mapping with annotations, as AspectJ does not support pointcuts with type parameters. For this reason we resort to abstract aspects that can be parameterized with a type. We wrap the advice that corresponds to a message view with type parameters in an abstract aspect and define an abstract pointcut

```
public interface Deferred extends Deferrable, Traced {
  ...
}
aspect DeferredAspect {
  ...
  // methods shown in message views
  @AdviceName("deferMethod")
  Object around(Deferred target):
   execution(@DeferredMethod * *.*(..)) && target(target) {
    Deferrable original = target.getOriginal();
    if (original != target) {
      return proceed(target);
    }
    else {
      ...
      return proceed(version);
    }
  }
}
```

Figure 50: Extract of the mapped code for the message view *deferMethod* (see Figure 49) showing the annotation based advice signature and parameterized proceed statements.

that is used in the advice. Instead of binding parameters to such message views with annotations, we need to define concrete aspects that extend the abstract aspect, provide values for the type parameters and instantiate the abstract pointcut in order to specify which method(s) should be bound. This means that the code of models with generic types, in contrast to all other generated code, cannot be used in legacy systems where it is impossible to insert AspectJ code.

An example use of this technique can be shown with the *Nested* aspect, which makes it possible for transaction contexts to be nested inside each other. *Nested* defines a behavioral template with a generic type in a message view *addChildrensResults* as shown in Figure 51. The model executes the unmodified behavior of the advised method, and then proceeds to iterate over all children contexts in order to apply the parameter method recursively to them. The result of each recursive invocation is added to the overall result and returned.

In the code that corresponds to this message view (Figure 52) we specify the type parameter `T`, a type that extends `Collection` (line 1), such that every instantiation of our abstract pointcut (line 2) will not need to contain it. It is sufficient to use `T` as return type of our `around` advice (line 3) and wherever it is used for variables. We retrieve the currently executing method from AspectJ's join point information (line 4) in order to invoke it recursively on the children objects (line 5). The `invoke` method can throw exceptions if it is used improperly so we need to wrap it in a `try-catch` block (line 6) even if our code is generated in a way that ensures that no reflection exceptions are ever thrown.

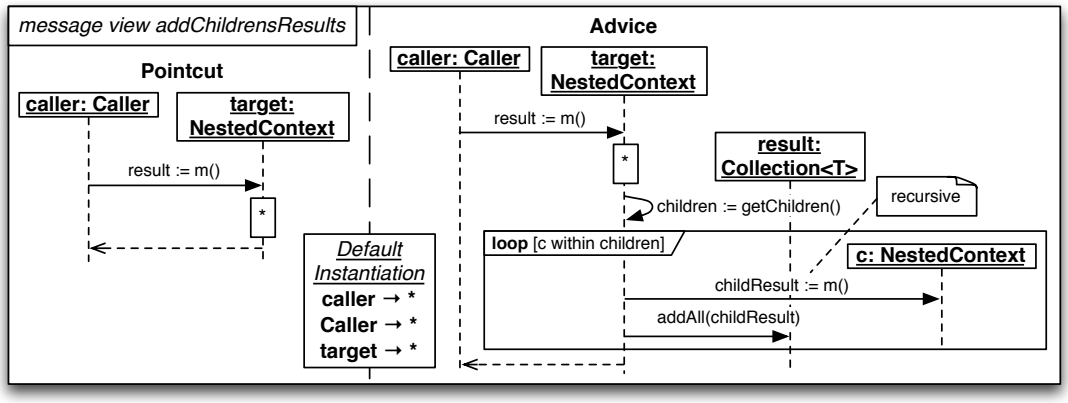

Figure 51: Template message view *addChildrensResults* of the *Nested* aspect showing a non-recursive proceed block and a recursive call for the method parameter *m*.

```
1 public abstract aspect NestedContextAddChildrensResultsAspect <T extends Collection<?>> {
2  public abstract pointcut m(NestedContext target);
   @AdviceName("addChildrensResults")
3  T around(NestedContext target) : m(target) {
    T result = null;
    try {
     result = proceed(target);
     Set<NestedContext> children = target.getChildren();
     for (NestedContext c : children) {
4      Method m = ((MethodSignature) thisJoinPointStaticPart.
       getSignature()).getMethod();
5      T childResult = (T) m.invoke(c, (Object[]) null);
      result.addAll(childResult);
     }
6   } catch (Exception e) {
     if (e instanceof InvocationTargetException)
7       throw new SoftException(e);
     // swallow reflection exceptions
     e.printStackTrace();
    }
    return result;
  }
 }
```

Figure 52: Mapped code for the template *addChildrensResults* (see Figure 51) involving a generic type parameter `T` as return type.

However, the bound method itself may throw exceptions that the reflection API wraps in `InvocationTargetExceptions`. These need to be re-thrown[2].

The use of generic behavior with type parameters cannot be implemented with annotations but leads to the definition of concrete aspects that extend the abstract aspects and instantiate the abstract pointcuts. In order to exemplify the implementation of the use of type parameterized methods we provide an example use of the generic method *addChildrensResults* and present the corresponding code in Section 5.3 in Figure 65 and Figure 66.

5.2.3.5 *Message View Content*

Once the structural framework for a message view has been determined and mapped to AspectJ code we analyze individual messages and constructs within the content area of a message view and map them to code. Most of the elements of RAM's message views are similar to elements of UML sequence diagrams and yield a straightforward mapping to Java code. Nevertheless we need to impose restrictions on the way how elements are used within message views in order to be able to obtain complete and correct code from our mapping.

Implement messages to receiver objects as method invocations on the receiver.

MAPPING MESSAGES In order to be mapped successfully a message view has to adhere to the structural patterns that we discussed in the previous sections by containing only one message from a caller to a target object in the pointcut part and the same message in the advice part. Messages that appear in the advice part after this first message and that are sent from an object a to an object b are implemented with a single Java statement that invokes the message of the same name on the object b using the same arguments. Self-messages require no separate mapping.

Initially the receiver of a message can only be the target object from the pointcut pattern or an object that is associated to this target in the structural view of the aspect that contains the message view. In case of a message to an associated object the base of the method invocation is the variable of the corresponding field obtained either from the special variable `this` in case of a inter-type declaration of the method or from the variable representing the target object in case of an AspectJ advice. Note that associations in RAM are only accessible from classes within the same aspect and therefore an object can not send a message to an object of a class that corresponds to an association outside the actual RAM aspect. By using getter methods with proper names this restriction can be bypassed without the need for further modeling as

2 Since the advised methods do not need to declare to throw the checked `InvocationTargetException`, we must wrap all exceptions in AspectJ's predefined `SoftException`.

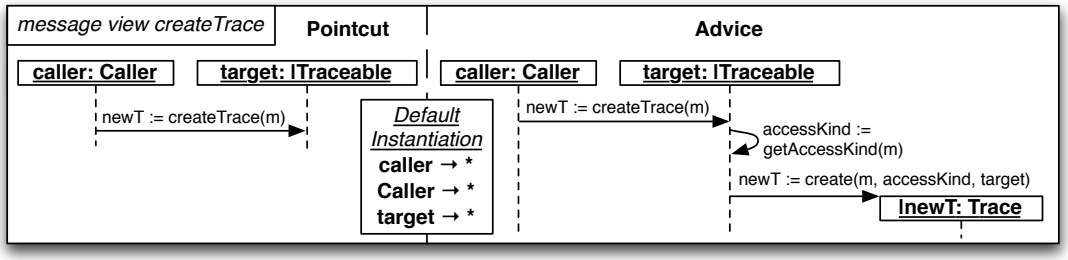

Figure 53: The message view *createTrace* from the *Traceable* aspect showing the creation of new variables and the use of constructors.

getter methods are mapped to complete default implementations as explained in Section 5.2.1.5.

Messages labeled with the name *create* stand for calls to constructors and the type for a resulting constructor call in JAVA is the type of the receiver of the message. If the name of the receiver object is not already in use we create a variable declaration statement that defines a variable of the same name using the receiver type and assign the result of the constructor call to it. This assignment can but does not need to be made explicit in the message view by prepending the name of the receiver object followed by := to the message label.

A label of an arbitrary message represents a variable assignment if it is of the form *var := method(p, ...)* where *var* is an arbitrary identifier and the list of parameters *p, ...* can be of any length. Such a variable assignment is implemented as a JAVA assignment statement whose right side corresponds to the code that we would have obtained by mapping only the part after the := characters. If the used variable identifier does not match a parameter variable of the advised method nor the role name of an available association, then we declare a new JAVA variable using the identifier. The type of the variable is inferred from the right side of the assignment as the return type of the called method. In case of a call to a method that returns a value of boolean type we allow the direct negation of the result prior to the assignment. In the model this is expressed with the keyword *not* following the := characters and in our code we simply insert a `!` before the call.

In Figure 53 we present the message view for the method *createTrace* of the *Traceable* aspect as an example that contains messages that create new variables and call constructors. The implementation of this message view as shown in Figure 54 creates a new variable of type `AccessKind` and assigns the return value of a call to the method `getAccessKind` to it.

RETURN VALUES AND INTERNAL STATEMENTS If a method has a non-void return type, a modeler may omit the optional asynchronous return message at the end of a message view in cases where this is unambiguous. The implementation of such

```
public interface Traceable extends AccessClassified {
  ...
}
aspect TraceableAspect {
  ...
  // methods shown in message views
  public Trace Traceable.createTrace(Method m) {
    AccessKind accessKind = this.getAccessKind(m);
    return new TraceImpl(m, accessKind, this);
  }
}
```

Figure 54: Implementation of the message view *createTrace* (see Figure 53) creating a new variable and calling a constructor.

an implicit or explicit return message is a JAVA return statement involving the result variable that is mentioned in the label of the first message from caller to sender. In cases where the last element of a message view is a note or a message specifying an assignment to the result variable we merge the return statement and variable assignment into a single return statement in order to obtain shorter code that is easier to read.

As it is impossible to express other statements than method calls in sequence diagrams we make use of notes in order to model assignment statements that involve constants in RAM. The implementation of these notes is a simple assignment to the variable with the name on the left side of the ":=" using the value of the constant on the right side. If a variable with the used identifier does not exist yet we declare a new variable and determine the type directly from a list of constants. So far we only used the boolean constants *true* and *false* but a mapping for integer number constants and string constants is equally trivial.

Conditional and alternative combination fragments have direct equivalents in JAVA .

CONDITIONAL AND ALTERNATIVE SEQUENCES

The most simple control flow construct that message views of REUSABLE ASPECT MODELS support are option combination fragments that are directly mapped to equivalent if-statements in JAVA. The code that corresponds to the optional message sequence is placed in the body of the corresponding if-statement. As the conventions for sequence diagrams differ from JAVA syntax we map the boolean operators <> to `!=`, = to `==`, and *not* to `!`. Note that we do not support hidden message calls in conditions of an option combination fragment in order to keep our mapping simple and to encourage what we consider good style.

Very similar elements of message views are alternation combination fragments that are implemented with an initial if- and consequent else-if-statements. Every alternative message sequence is mapped as if it would be outside the fragment and the resulting code is placed in the block of the corresponding if- or else-if-statement. The mapping for conditions is the same as for option

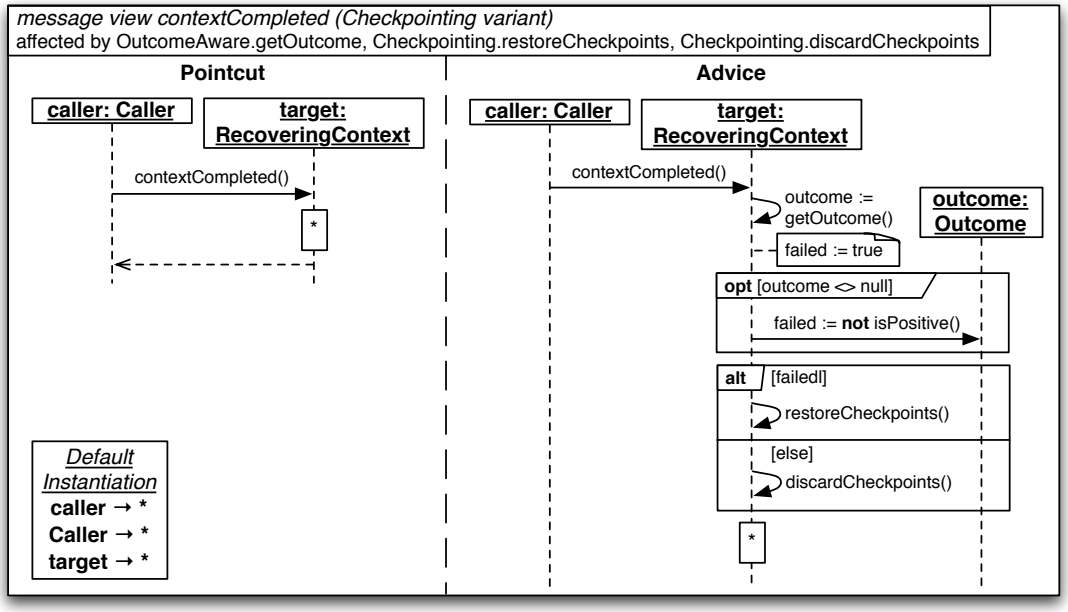

Figure 55: The message view *contextCompleted* for the *Checkpointing* variant of the *Recovering* aspect containing an option and an alternation combination fragment.

combination fragments except for the special *[else]* condition that yields an else-statement instead of an else-if-statement.

In Figure 55 and 56 we present the message view and the implementation for the *Checkpointing* variant of the method *contextCompleted* from the *Recovering* aspect as an example of a method that contains an option and an alternation combination fragment.

Repetitive sequences are implemented with while loops, iterators, and for-each-loops.

LOOP COMBINATION FRAGMENTS The most powerful control flow constructs that can be used in RAM's message views are loop combination fragments that we implement using JAVA's while and for-each loops. The code that we obtain from mapping the content of the loop fragment is always placed into the block of the corresponding JAVA loop.

Our implementation of the condition and initialization part of a loop however differs depending on the structure of the condition that was used in the model. A boolean condition as it can be used in optional and alternation combination fragments is mapped to a plain `while` loop whose conditional expression is obtained in the same way as described above for option combination fragments.

If the condition of a loop contains the keyword *within*, then the modeled loop iterates over the elements of a collection and our implementation needs to contain the corresponding logic for the creation and modification of a pivot element. In order to be able to create a complete implementation of this suggestive condition we need to rely on the assumption that the type of the variable that we should iterate over is a subclass of `java.util.Collection`. Given the possibility to implicitly reuse JAVA library classes as

```
public interface RecoveringContextCheckpointing
 extends RecoveringContext, CheckpointingContext {
  ...
}
aspect RecoveringContextCheckpointingAspect {
  ...
  // methods shown in message views
  before(RecoveringContextCheckpointing context) :
   execution(void Context.contextCompleted()) && target(context) {
    Outcome outcome = context.getOutcome();
    boolean failed = true;
    if (outcome != null) {
      failed = !outcome.isPositive();
    }
    if (failed) {
      context.restoreCheckpoints();
    } else {
      context.discardCheckpoints();
    }
  }
}
```

Figure 56: Implementation of the *Recovering* variant of the message view *contextCompleted* (see Figure 55) showing the mapping of option and alternation combination fragments to if- and else-statements.

described in Section 5.2.1.1 we believe that this restriction is easy to fulfill for users of RAM.

An example of a simple loop combination fragment is given in Figure 57 with the message view of *performUpdate* from the *Deferring* aspect. Every element of the variable *traced* is processed iteratively using a pivot name *t* in order to send a message *updateOriginal* to it. The implementation of this message view using a for-each loop is shown in Figure 58.

For loop combination fragments whose condition is not composite we use JAVA's compact for-each loop as it produces the leanest and most readable code. The collection that is to be iterated over is stated directly after the *within* keyword in the

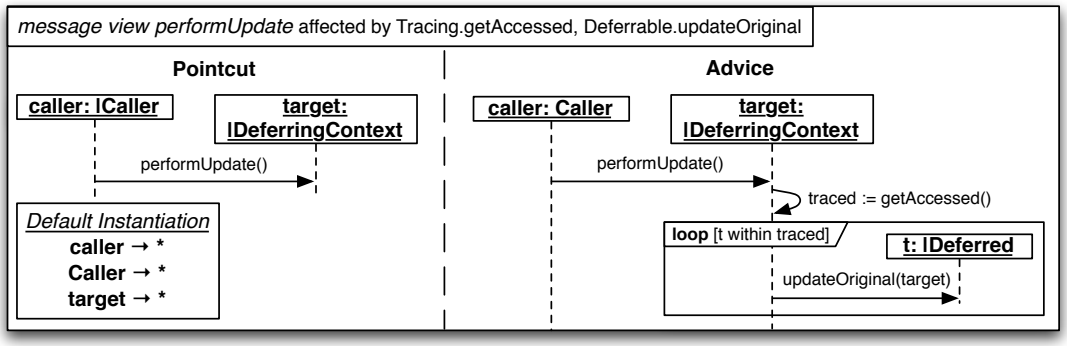

Figure 57: The message view *performUpdate* of the *Deferring* aspect containing a loop combination fragment that iterates over a collection using the keyword *within*.

```
public interface DeferringContext extends TracingContext {
  ...
}
aspect DeferringContextAspect {
  // methods shown in message views
  public void DeferringContext.performUpdate() {
    Collection<Traced> traced = getAccessed();
    for (Traced t : traced) {
      ((Deferrable) t).updateOriginal(this);
    }
  }
}
```

Figure 58: Code for the method *performUpdate* of the *Deferring* aspect (see Figure 57) demonstrating the use of for-each loops to implement loop combination fragments involving *within*.

modeled condition and occurs last in the resulting JAVA loop but provides the type for the pivot variable. This pivot type forms the beginning of our for-each-loop condition followed by the name of the pivot element that is mentioned in the condition before the *within*. The variable name of the collection that is to be iterated over appears last after a separating colon.

In case of a composite loop condition that combines a *within* iteration with boolean expressions we cannot use JAVA's simple for-each-loop. Instead, we need to work with a classical iterator and a while loop. Our code begins with the declaration of an iterator variable that is parameterized with the element type of the collection and initialized by calling the `iterator` method on the collection. The following while-loop has a compound condition that combines a call to the `hasNext` function of the iterator with the boolean expression of the loop combination fragment. We start the block of the while loop with the declaration of the variable for the pivot element and directly assign the return value of a call to the `next` function on the iterator to it. The code that corresponds to the content of the loop combination fragment is inserted immediately after this statement.

Figure 59 presents the message view for the method *wasAccessed* in the *Nested / Tracing* conflict resolution aspect that contains a loop combination fragment with a compound condition. This condition *c within children && not accessed* expresses that we want to iterate over all elements *c* of the variable *children* as long as the value of the variable *accessed* is *false*. The resulting code is shown in Figure 60 and involves an iterator variable whose information is used for the loop condition and binding of the pivot element.

ENSURING TYPE SAFETY In RAM the type of an object in a message view might be more specific than the return type of the method that provided the object. In our implementation we

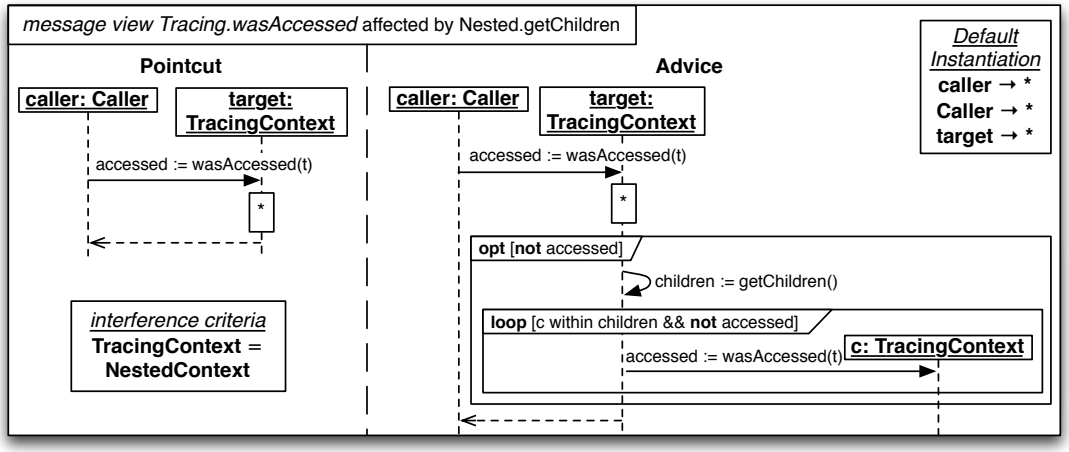

Figure 59: Message view for the method *wasAccessed* in the *Nested / Tracing* conflict resolution aspect comprising a loop combination fragment with a compound condition.

```
privileged aspect TracingContextAspect {
  ...
  // methods shown in message views
  boolean around(NestedContext target, Traced t) :
   execution(boolean TracingContext.wasAccessed(..)) && target(target) && args(t) {
    boolean accessed = proceed(target, t);
    if (!accessed) {
      Set<NestedContext> children = target.getChildren();
      Iterator<NestedContext> childrenIterator = children.iterator();
      while (childrenIterator.hasNext() && !accessed) {
        NestedContext c = childrenIterator.next();
        accessed = ((TracingContext) c).wasAccessed(t);
      }
    }
    return accessed;
  }
  ...
}
```

Figure 60: Code for the *wasAccessed* method in the *Nested / Tracing* conflict resolution aspect (see Figure 59) using an iterator to implement a loop with a compound condition.

enforce these types with explicit type conversions. If the lifeline of an object specifies a type that differs from the return type of the method that returned the object, we insert a downcast to the type that is mentioned for the lifeline between the corresponding call and assignment.

In case of *within* loops that specify a different type for the pivot element than the iterated collection we prepend an inline cast to the modeled type to every method call on the pivot element. The current status of the AspectJ compiler does not allow inter-type introduction on generic classes whose type parameter is also generic. For this reason we could not realize our plan to parameterize every type such that methods of reused aspects will correctly return the more specific types of the reusing aspect. We hope that further improvements on the AspectJ compiler will make it possible to use inter-type introduction on generic classes with generic types as this would allow us to avoid these typecasts.

5.3 CONFLICT RESOLUTION ASPECTS

Conflict resolution aspects are privileged aspects with interference criteria.

In order to automatically detect and resolve conflicts between reused aspects RAM supports the definition of conflict resolution aspects. These aspects have all features of ordinary aspects, but they cannot be instantiated. Instead, they are automatically applied if their *interference criteria* are met. To support that, our mapping needs to generate code for conflict resolution aspects that is only executed when the corresponding criteria hold.

In the AspectOPTIMA case study all interference criteria are of the form *ClassA* = *ClassB*. Such a criterion captures cases in which the binding and instantiation directives of the aspects that define the involved classes and the directives of a reusing aspect resulted in a merge of both classes.

An additional feature of conflict resolution aspects is that they are given access to methods, classes and associations with package access modifier for both involved aspects. This means for example that message views of a conflict resolution aspect have direct access to associated objects and can contain calls to methods that would not be visible from plain aspects. Our implementation achieves this by prepending the keyword `privileged` to every AspectJ aspect that results from a RAM conflict resolution aspect.

5.3.1 *Structural View*

Our AspectOPTIMA case study contains no conflict resolution aspect with a structural view, but our mapping would be able to implement such views with an interference criterion of the form $A = B$ without further complications. In a preparatory step we would need to define an interface `AB` that stands for the merged

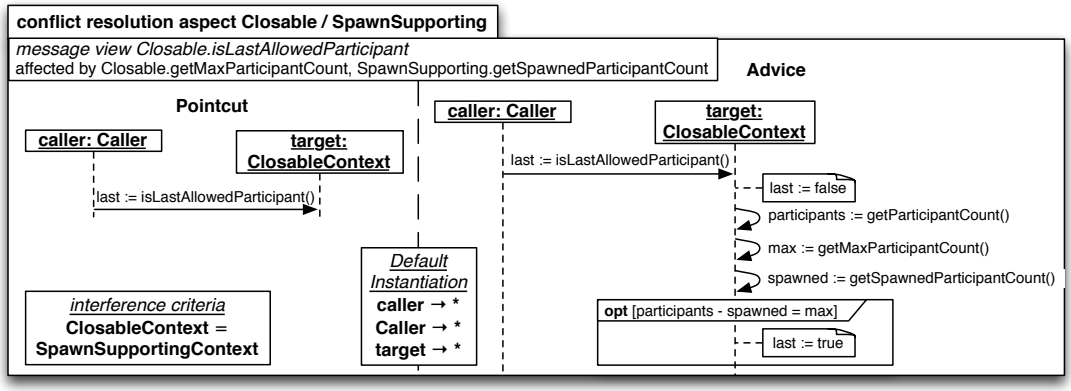

Figure 61: The conflict resolution aspect *Closable / SpawnSupporting* shows the use of methods of both merged classes on a target of type *ClosableContext*.

type and we would make every class that is an instance of both merged classes implement this interface with a type hierarchy modification of the form `declare parents: (@AClass @BClass *) implements AB`. The content of the structural view could then be implemented almost the same way as described in Section 5.2.1 with the only difference that the new interface `AB` would be used instead of `A` or `B`.

5.3.2 *Message View*

The implementation of message views of conflict resolution aspects follows the same rules as described in Section 5.2.3 for message views of plain aspects with slight modifications. We will discuss these modifications for three different structural circumstances in which message views can occur and present the special case of a conflict resolution aspect with type parameters.

5.3.2.1 *Structural Circumstances*

Let us now consider a message view of a conflict resolution aspect with an interference criterion *ClassA = ClassB* that advises an existing method (see Section 5.2.3.2) that is initially declared in one of the merged classes, say *ClassA*, and unavailable in the other, say *ClassB*. In such a case we specify in the advice part of the corresponding aspect signature that the target object is of type *ClassB*. The result is that we only advice the execution of the method in a case where *ClassA* provided the method and the target is an instance of *ClassB*. This means that the target is an instance of both classes. An example for such an interference criterion guaranteeing advice signature is shown in Figure 62 for the *Closable / SpawnSupporting* conflict resolution aspect that we present again in Figure 61.

```
privileged aspect ClosableContextAspect {
  ...
  // methods shown in message views
  boolean around(SpawnSupportingContext target) :
   execution(boolean ClosableContext.isLastParticipant()) && target(target) {
    boolean result = false;
    int participants = target.getParticipantCount();
    int max = ((ClosableContext) target).getMaxParticipantCount();
    int spawned = target.getSpawnedParticipantCount();
    if (participants - spawned == max) {
      result = true;
    }
    return result;
  }
}
```

Figure 62: Code for the conflict resolution aspect *Closable / SpawnSupporting* (see Figure 61) showing the interference criteria guaranteeing target variable of type `SpawnSupportingContext` and an inline typecast for *getMaxParticipantCount*.

In case of a message view with the same interference criterion *ClassA = ClassB* that advises an existing method that is not initially declared in one of the merged classes, we need to add a further pointcut to the corresponding advice signature in order to ensure the interference criterion. This is necessary because AspectJ forces us to specify the declaring type in the pointcut part of the advice as the type of the lowest class in the type hierarchy that declared the method. An advice that only specifies that the target object is of type *ClassA* for example would apply in every case where a class that extends the declaring class is merged with *ClassA*. Therefore we append a pointcut of the form `if(target instanceof ClassB)` to the advice signature, thus restricting our advice to the cases that meet the interference criterion.

The last circumstance of a message view of a conflict resolution aspect is the definition of a completely new method. As such a method does not need to be specified in a structural view if it is only reused in the defining conflict resolution aspect, we need to retrieve the structural information from the defining method view and from a message view that contains a call to the new method. The declaring class for our new method is the class of the target of the corresponding message in the pointcut part of the message view. From a call to the new method in another message view we obtain the parameter types as the types of the concrete arguments of that call. As the method is only used within this conflict resolution aspect we are guaranteed to find such a call. If this would not be the case the new method would never be used and could be ignored. An example of such a definition of a new method is *copyAndMapNewVersion* of the conflict resolution aspect *Deferrable / Nested* as shown in Figure 63. The code of the corresponding method, presented in Figure 64,

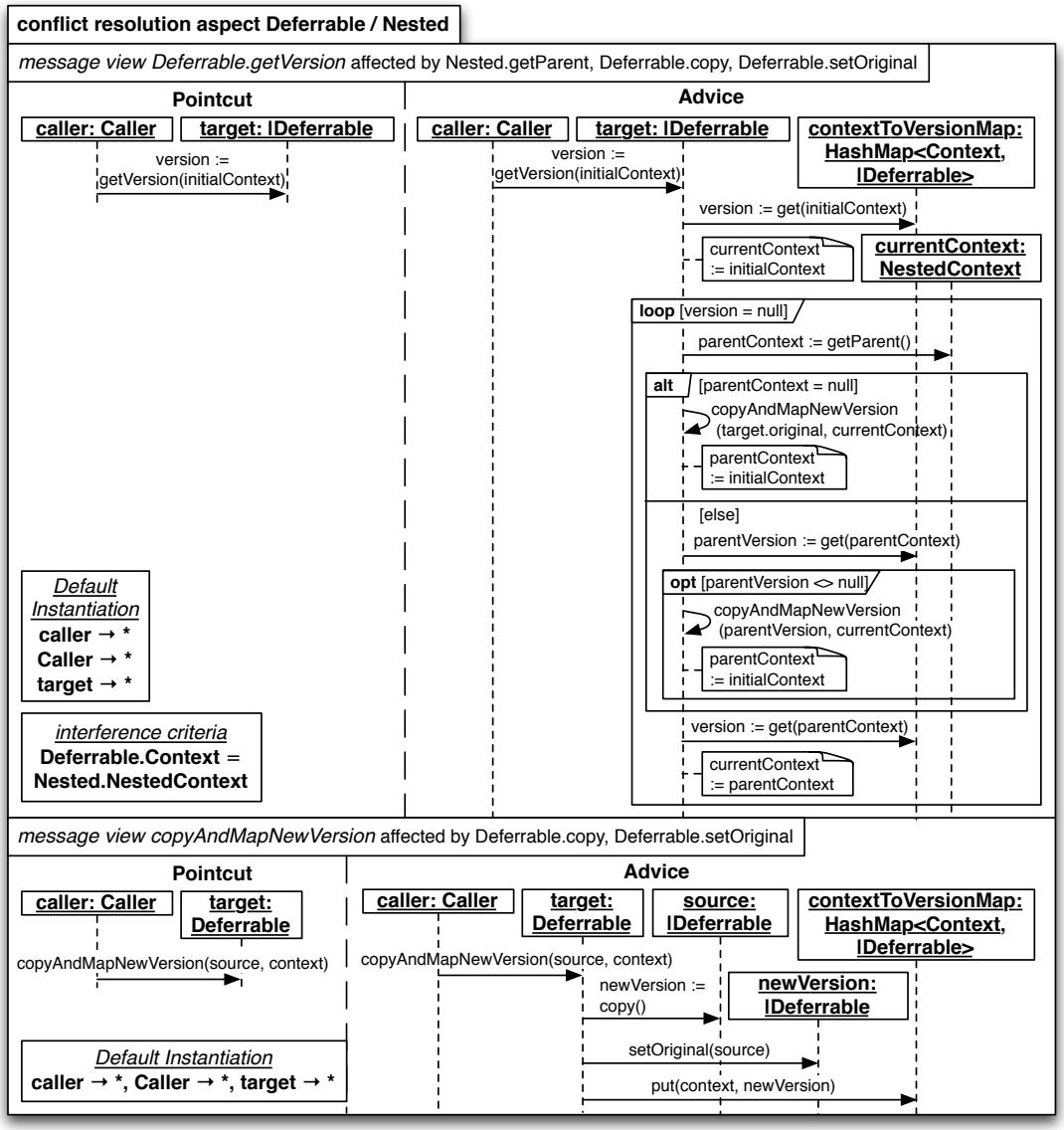

Figure 63: The definition and use of a new method *copyAndMapNewVersion* in the conflict resolution aspect *Deferrable / Nested*.

demonstrates the direct access to the field that corresponds to the *contextToVersionMap* association of the *Deferrable* aspect.

5.3.2.2 *Conflict Resolutions with Type Parameters*

The restriction to use abstract and extending pointcuts and aspects for conventional aspects with type parameters also applies to conflict resolution aspects. In Section 5.2.3.4 we explained this special case using the method template method *addChildrensResults* from the the aspect *Nested*. We will now use the conflict between the aspects *Nested* and *Tracing* in order to show a message view of a conflict resolution aspect that uses the generic template method of our prior example.

```
privileged aspect DeferrableAspect {
  ...
  // methods shown in message views
  ...
  private void Deferrable.copyAndMapNewVersion(Deferrable source, Context context) {
    Deferrable newVersion = (Deferrable) source.copy();
    newVersion.setOriginal(source);
    this.contextToVersionMap.put(context, newVersion);
  }
}
```

Figure 64: Code for the *copyAndMapNewVersion* method of the conflict resolution aspect *Deferrable / Nested* (see Figure 63) showing privileged access to the association field *contextToVersionMap*.

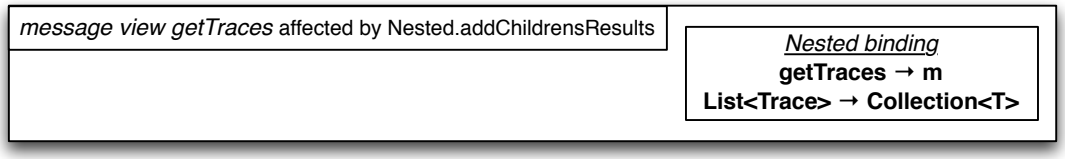

Figure 65: The message view *getTraces* from the *Nested / Tracing* conflict resolution aspect binding the template *addChildrensResults*.

Our conflict resolution aspect *Nested / Tracing* was modeled in order to make traces that were gathered in a child context also available to the parent context. To achieve the desired effect, the conflict resolution aspect applies the template behavior `addChildrensResults` (Figure 51) to the method `getTraces` (Figure 65). The results is that all traces of children contexts are added to the traces of a context before the result is returned.

To achieve the same effect in AspectJ , our mapping produces a concrete aspect that extends the abstract aspect (Figure 52) as shown in Figure 66. Its purpose is to assign the return type parameter to `List<Trace>` and to instantiate the abstract pointcut `m` to the method `getTraces`. The conflict criteria `TracingContext = NestedContext` is also visible in the generated code: in the pointcut definition we intercept executions of the `getTraces` method of a `TracingContext`, but the declared target type is `NestedContext`.

5.3.2.3 *Message View Content*

The content of a message view that is contained in a conflict resolution aspect is mapped to AspectJ code the same way a message view of a plain aspect would be mapped, except for one additional rule: Whenever a message is passed to an object of

```
aspect GetTracesAddChildrensResultsAspect extends AddChildrensResultsAspect<List<Trace>> {
  public pointcut m(NestedContext target) :
   execution(List<Trace> TracingContext.getTraces()) && target(target);
}
```

Figure 66: Mapped code for the method *getTraces* (see Figure 65) extending the abstract aspect for *addChildrensResults*.

one of the merged classes that corresponds to a method that is not available at this class we insert an inline typecast to the other merged class into the corresponding method call statement. The conflict resolution aspect *Closable / SpawnSupporting* that already served as an example in Section 5.3.2.1 also demonstrates the use of such inline typecasts as shown in Figure 62.

5.4 CONFIGURING PRODUCT LINES

Due to its support for optional and alternative aspects, REUSABLE ASPECT MODELS allow a modeler to create a line of software products with varying configuration. When we map a model with support for different configuration possibilities to code we need to account for every possible variation.

A RAM aspect that supports different variants is modeled the same way as other aspects except for three little modifications: The list of reused aspects can contain elements of the form *(AspectA xor AspectB)* to denote alternatives and elements of the form *opt AspectC* to denote optional aspects. In both cases a message view can be defined only for a certain configuration by appending *(AspectV variant)* to the name of the message view where *AspectV* stands for the aspect that is used in this variant of the aspect.

In case of an alternative variant of the form *AspectA xor AspectB* we create a JAVA enumeration that lists both possibilities for convenient reuse in all involved aspects. The name of this enumeration is the name that was given to this alternative in the feature diagram of the project and the enumeration literals bear the name of the reused aspects. As an example of an aspect with an alternative variant we present the structural view of the *Recovering* aspect in Figure 67. The binding and instantiation directives for the alternative aspects *Checkpointing* and *Deferring* are only executed in the variant that uses the corresponding aspect. As this alternative was named *UpdateStrategy* in the feature diagram of ASPECTOPTIMA (Figure 2), the resulting enumeration bears the same name. We present it in Figure 68.

An optional variant requires no special structure that is reused, and binding and instantiation directives involving an optional aspect are only applied if the corresponding variant is chosen. In both variation cases, alternative and optional, we need to determine for every class whether its structure or behavior change due to the variation. Such a change is effected by different instantiation or binding directives or variant specific message views for methods of a class. If we observe a changed structure or behavior for a class we create an additional interface that extends the original interface of that class and has the name of the aspect that causes the variation appended to its name. Furthermore, we account for the variation in the annotation that corresponds to the

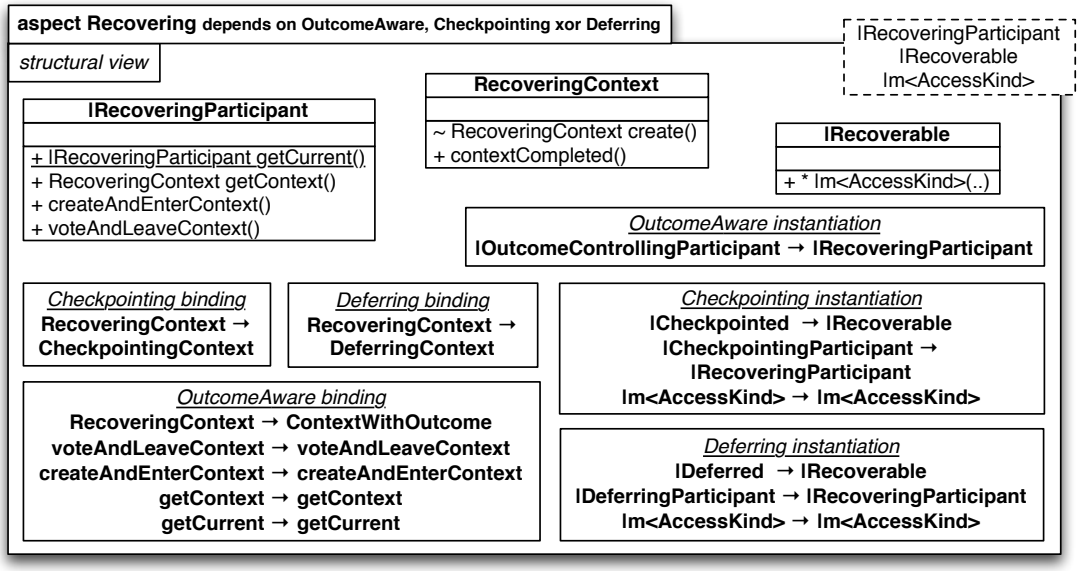

Figure 67: Structural view of the *Recovering* aspect with an alternative for reused aspects and corresponding binding and instantiation directives.

```
public enum UpdateStrategy {
  // enumeration literals for reused aspects
  CHECKPOINTING {
    @Override
    public String toString() {
      return "Checkpointing";
    }
  },
  DEFERRING {
    @Override
    public String toString() {
      return "Deferring";
    }
  };
}
```

Figure 68: Enumeration for the *UpdateStrategy* alternative of the *Recovering* aspect (see Figure 67) preserving the original literal names in the `toString` method.

```
@Target( { ElementType.TYPE })
@Retention(RetentionPolicy.RUNTIME)
public @interface TransactableClass {
  UpdateStrategy updateStrategy();
  ConcurrencyControl concurrencyControl();
}
```

Figure 69: Annotation for the class *Transactable* of the *Transaction* aspect (see Figure 17) showing parameters for an inherited and newly defined alternative.

```
@Target( { ElementType.TYPE })
@Retention(RetentionPolicy.RUNTIME)
public @interface TransactionParticipantClass {
  UpdateStrategy updateStrategy();
  ConcurrencyControl concurrencyControl();
  boolean nested();
}
```

Figure 70: Annotation for the *Transaction* aspect's *TransactionParticipant* class (see Figure 17) containing parameters for two alternatives and an optional reuse.

class by parameterizing it. In case of an alternative the parameter has the corresponding enumeration as type and in case of an optional aspect we add a boolean parameter that evaluates to true if the aspect is chosen.

As variations are preserved in reusing aspects we need to apply the same technique to classes that are merged with varying classes in reusing aspects. This means that for every class we need to account for every possible configuration that results in a different structure or behavior for this class. An example that demonstrates these effects is the *Transaction* aspect as it inherits the alternative *UpdateStrategy* from the reused *Recovering* aspect, defines a new alternative *ConcurrencyControl*, and depends optionally on the *Nested* aspect. The code for the annotation for the class *Transactable* is shown in Figure 69. Note that both alternatives have an effect on this class, but the optional *Nested* aspect has no effect on it. In contrast to this the class *TransactionParticipant* is affected by the *Nested* option too and therefore contains an additional boolean parameter as shown in Figure 70.

Our annotation inheritance technique that we discussed in Section 5.2.1.6 is also affected by the variation parameters. When we declare an annotation on an element that is marked with an annotation that involves variation parameters we need to preserve this information for the newly added annotation. An example of this is the configuration for the alternative *UpdateStrategy* that needs to be passed on from the annotation of a *Transactable* object of the *Transaction* aspect to the annotation that marks it as *Recoverable* for the *Recovering* aspect as shown in Figure 71.

All code that results from the variation independent part of a structural view and from message views that do not list a special

```
// Transaction instantiates Recovering
declare @type :
 @TransactableClass(updateStrategy = UpdateStrategy.CHECKPOINTING) * :
  @RecoverableClass(UpdateStrategy.CHECKPOINTING);
declare @type :
 @TransactableClass(updateStrategy = UpdateStrategy.DEFERRING) * :
  @RecoverableClass(UpdateStrategy.DEFERRING);
...
// Transaction instantiates TwoPhaseLocking xor OptimisticValidation
declare @type :
 @TransactableClass(concurrencyControl = ConcurrencyControl.TWOPHASELOCKING) * :
  @TwoPhaseLockedClass;
declare @type :
 @TransactableClass(concurrencyControl = ConcurrencyControl.TWOPHASELOCKING) * :
  @TwoPhaseLockedClass;
declare @type :
 @TransactableClass(concurrencyControl = ConcurrencyControl.OPTIMISTICVALIDATION) * :
  @OptimisticallyValidatedClass;
declare @type :
 @TransactableClass(concurrencyControl = ConcurrencyControl.OPTIMISTICVALIDATION) * :
  @OptimisticallyValidatedClass;
```

Figure 71: Extract from the `AnnotationInheritance` aspect that forwards configuration information for `Transactable` objects (see Figure 17) to the `Recoverable` annotation.

variant is introduced into the variation independent interfaces. These general interfaces extend all interfaces that correspond to bound or instantiated classes of directives that are valid for all variations. An example for invariant methods are *beginTransaction* etc. of the *Transaction* aspect. As variation independent methods may use methods that are not inherited from a variation independent aspect but inherited in every variation we need to make these methods available in the general interface. We achieve this by making the general interface extend all interfaces that correspond to classes that are merged with the class in consideration in all possible variants. An example for such indirectly invariant interfaces are *TracingParticipant* and *OutcomeAwareParticipant* that are extended by the variation independent interface *RecoveringParticipant* as *OutcomeAware* is an aspect that is reused in every variation and *Tracing* is an aspect that is reused in both alternatives *Checkpointing* and *Deferring*.

Code that results from variation specific message views is introduced into the special interfaces of that variant. The mapping of instantiation and binding directives that are variation specific is also applied to these special interfaces. An example of a variation specific method is *contextCompleted* from the *Recovering* aspect.

To ensure that a user or modeler can never choose both options of an alternative we create a special AspectJ aspect named `ConfigurationEnforcement` in the main package of every project. In this aspect we use AspectJ's facilities to declare weaver errors at compiletime and specify that involved classes can never be marked with annotations that correspond to both alternatives.

```
aspect ConfigurationEnforcementAspect {
  // declare runtime errors if xor exclusions are violated
  declare error : execution(* (@CheckpointedClass @DeferredClass *).*(..)) :
   "The configuration parameter \"update strategy\" of the |\"Recovering\" aspect " +
   "is an exclusive or! You cannot use both options simultaneously!";
  ...
  // declare runtime errors if composition rules are violated
  declare error : execution(* (@OptimisticallyValidatedClass @CheckpointedClass *).*(..)) :
   "The value \"Checkpointing\" for the configuration parameter \"update strategy\" is " +
   "not allowed in combination with the value \"OptimisticValidation\" " +
   "for the configuration parameter \"concurrency control\"!";
}
```

Figure 72: ConfigurationEnforcement aspect containing weaver error declarations for configurations that are forbidden due to alternatives or composition rules.

An extract from this `ConfigurationEnforcement` aspect that also ensures composition rules that are mentioned in the feature diagram is shown in Figure 72.

5.5 MULTIPLE REUSE OF ASPECTS WITH DIFFERENT BINDING

So far we only considered cases of reuse where classes were bound to reused classes exactly once. During the design of a system cases can occur where a reused class is merged more than once with an existing class due to instantiation or binding directives. This is only useful if the names of the attributes, associations and methods of this reused class are renamed in at least all but one reusing aspect.

TWO MAPPING APPROACHES In order to support these multiple reuses we need to generate code that gives us the possibility to rename fields and methods and to use these fields and methods accordingly. We developed two mapping approaches that can be applied independent of each other for different aspects within a single system. The first approach simply duplicates the code of the reused aspect with new names for fields and methods whereas the second approach does not duplicate modeled logic but uses JAVA'S REFLECTION API to access renamed fields and methods. We do not believe that one of both approaches has unbeatable advantages over the other but that both should be used in different circumstances in order to generate the best code within each context.

The duplication approach has the disadvantage that modeled logic is duplicated such that implementational refinements of this logic need to be performed at multiple places. This leads to higher maintenance efforts and a higher probability of failure. A big advantage of this approach however is that no additional overhead is introduced as no expensive calls to methods of the JAVA REFLECTION API are used.

The code that we obtain from the reflection approach is leaner and easier to update and maintain than the duplicated code of the first approach. But we pay this advantage with lower performance due to reflective access to methods and fields. The instantiation or binding directives that made it necessary to use one of these approaches however have a more direct equivalent in the reflection approach than in the duplication approach.

A future code generator should offer a modeler the possibility to decide for every aspect that is reused more than once which approach he wants to use. This would ensure that small aspects that are already complete such that their code is unlikely to be changed can be implemented with the cheap duplication approach whereas the logic of bigger aspects that might be changed later on is not spread over the system when the reflection approach is used.

5.5.1 *Duplication Approach*

In our extended AspectOPTIMA case study with support for Open Multithreaded Transactions we decided to implement the multiple reuse of the *Lockable* aspect using the duplication approach. This means that the code that we created for the *Lockable* aspect is used directly in the *TwoPhaseLocking* aspect as it changes no name of any entity. For the *Shared* aspect that uses *Lockable* with different names for methods and attributes we duplicate the original *Lockable* code and use these different names.

In both cases we do not work directly with the interface that corresponds to the *Lockable* class but define new interfaces whose name is *LockableFor* followed by the name of the reusing aspect. The classes *TwoPhaseLocked* and *Shared* extend these specifc interfaces instead of the general *Lockable* interface as the ordinary mapping would require.

The code for the reuse of *Lockable* in *TwoPhaseLocking* is shown in Figure 73. As *TwoPhaseLocking* reuses *Lockable* with unchanged names for attributes and methods the resulting interface extends the *Lockable* interface directly and is empty except for a `declare parents` directive. This AspectJ directive makes sure that every class that implements the *Lockable* as well as the *TwoPhaseLocked* interface automatically implements the interface that we created specially for this reuse of *Lockable*.

The *Shared* aspect however reuses the *Lockable* aspect with different names for the field *myLock* and the methods *getLock* and *releaseLock* as shown in Figure 74. For this reason the resulting *LockableForShared* interface does not extend the *Lockable* interface but contains exactly the same code as this interface except for the changed names and the `declare parents` directive. The corresponding code for this reuse is shown in Figure 75.

```
public interface LockableForTwoPhaseLocking extends Lockable {
  // empty: direct reuse of the default binding for Lockable
}
aspect LockableForTwoPhaseLockingAspect {
  // bind annotations to mandatory instantiation parameter interfaces
  declare parents :
   (@LockableClass @TwoPhaseLockedClass *) implements LockableForTwoPhaseLocking;
  // empty: direct reuse of the default binding for Lockable
}
```

Figure 73: Mapped code for the direct reuse of *Lockable* in *TwoPhaseLocking* (see Figure 15 and 16) with unchanged entity names.

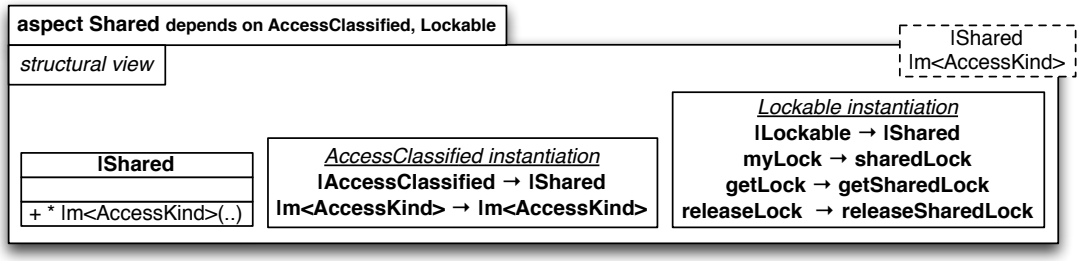

Figure 74: Structural view of the *Shared* aspect containing an instantiation directive that changes the names of an association and two methods of the reused *Lockable* aspect.

```
public interface LockableForShared {
  // public method signatures from structural view
  void getSharedLock(AccessKind accessKind);
  void releaseSharedLock(AccessKind accessKind);
}
aspect LockableForSharedAspect {
  // bind annotations to mandatory instantiation parameters
  declare parents :
   (@LockableClass @SharedClass *) implements LockableForShared;
  // attributes & associations from structural view
  private Lock LockableForShared.mySharedLock = new LockImpl();
  // declared methods without message views
  // empty
  // methods shown in message views
  public void LockableForShared.getSharedLock(AccessKind accessKind) {
    ...
  }
  public void LockableForShared.releaseLock(AccessKind accessKind) {
    ...
  }
}
```

Figure 75: Mapped code for the reuse of *Lockable* in *Shared* (see Figure 74) with a renamed attribute and two renamed methods.

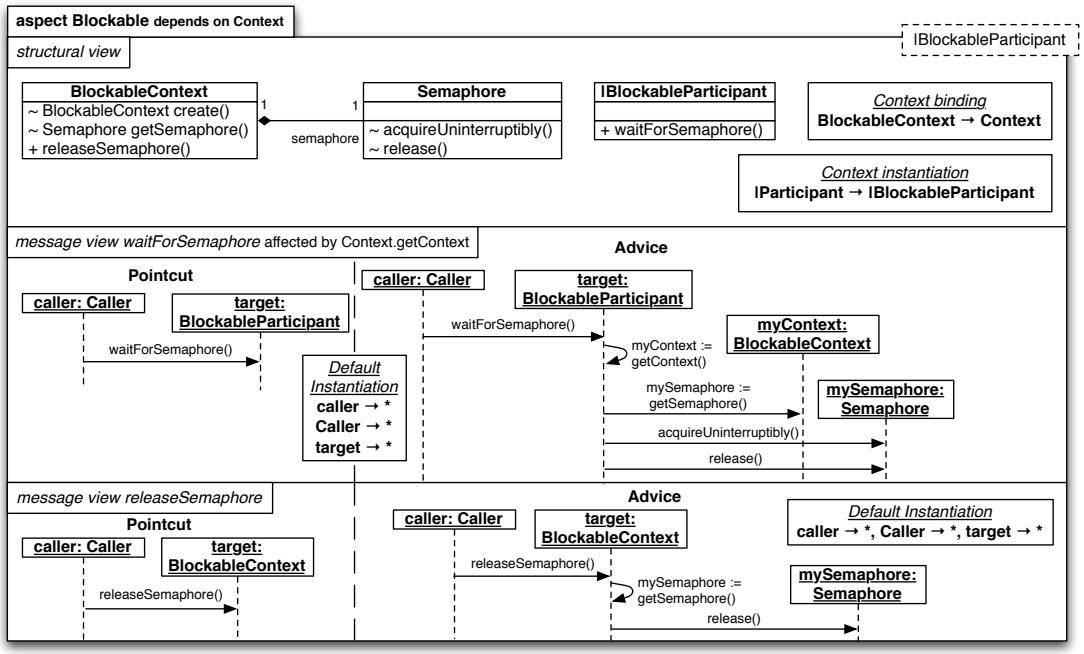

Figure 76: The model for the *Blockable* aspect containing the association *semaphore* and the methods *getSemaphore*, *releaseSemaphore*, and *waitForSemaphore* that are renamed in reusing aspects.

5.5.2 *Reflection Approach*

A more sophisticated approach accesses fields and methods using Java's Reflection API in order to avoid code duplication. It has been applied to the reuses of the *Blockable* aspect of the Open Multithreaded Transaction extension to AspectOPTIMA. The *Blockable* aspect itself is presented again in Figure 76. It is a good example for our general mapping for multiple reuses as it contains an attribute as well as a dependency between methods that needs to be maintained even when they are renamed. The message views of these methods are implemented in static methods of two helper classes `BlockableParticipantMessages` and `BlockableContextMessages`. In Figure 77 we present the code for `BlockableContextMessages` that also contains a helper method to retrieve the getter method *getSemaphore*. Note that instead of calling the `getSemaphore` method directly it is called using the passed binding mapping and Java's Reflection API. This means that no matter how we renamed the *getSemaphore* method in a reusing aspect we will always be calling the right method from within the corresponding *releaseSemaphore* method.

All reuses of *Blockable* are syntactically equivalent such that it is sufficient to discuss the reuse in the *ExitSynchronizing* aspect. Figure 78 presents the structural view of the *ExitSynchronizing* aspect in which the *semaphore* association and three methods of *Blockable* are renamed within a binding directive. We implement this binding with a class that contains nothing but a static field

```
class BlockableContextMessages {
  // static version of methods shown in message views
  static void releaseSemaphore
   (BlockableContext blockableContext, Map<String,String> bindingMap) {
    try {
      Class<? extends BlockableContext> blockableContextClass = blockableContext.getClass();
      Method getSemaphoreMethod = getGetSemaphoreMethod(blockableContextClass, bindingMap);
      Semaphore mySemaphore =
       (Semaphore) getSemaphoreMethod.invoke(blockableContext, (Object[]) null);
      mySemaphore.release();
    } catch (Exception e) {
      // swallow
      e.printStackTrace();
    }
  }
  // helper methods for access to bound fields and methods
  static Method getGetSemaphoreMethod
   (Class<? extends BlockableContext> blockableContextClass, Map<String,String> bindingMap)
   throws SecurityException, NoSuchMethodException {
    String getSemaphoreName = (bindingMap == null) ? "getSemaphore" : bindingMap.get("getSemaphore");
    Method method = blockableContextClass.getDeclaredMethod(getSemaphoreName, (Class<?>[]) null);
    method.setAccessible(true);
    return method;
  }
}
```

Figure 77: Helper class `BlockableContextMessages` containing the method `releaseSemaphore` of `BlockableContext` that is detailed in a message view (see Figure 76) and a helper method for the getter method `getSemaphore`.

`map` that links the names of associations, attributes, and methods to their new names in the reusing aspect. The resulting code for *ExitSynchronizing* is shown in Figure 78.

Similary to the *LockableForShared* interface in the duplication approach that we discussed in the preceding chapter we create an interface *BlockableForExitSynchronizing* for the reuse of *Blockable* within *ExitSynchronizing*. Note that we still need to introduce the field that corresponds to the association that was renamed to *exitSemaphore* and the adjunct getter method[3]. The conceptual difference however is that complete methods that are detailed in message views are no longer introduced. Instead of that we introduce a single method call to the static implementation of the message view in the helper class `BlockableContextMessages` and provide the binding as an argument. The resulting code for *ExitSynchronizing*'s reuse of *Blockable* is shown in Figure 80.

For this short example the length and obscurity of the calls to the methods of Java's Reflection API might not seem worth the effort. Nevertheless we believe that for complicated examples that may require fine tuning after the modeling phase the advantage of having all behavioral logic at a single spot weights out the additional reflection infrastructure.

3 We do not use a general getter method that uses reflection to access the field as this is far more expensive than a single line of code even if it is "duplicated".

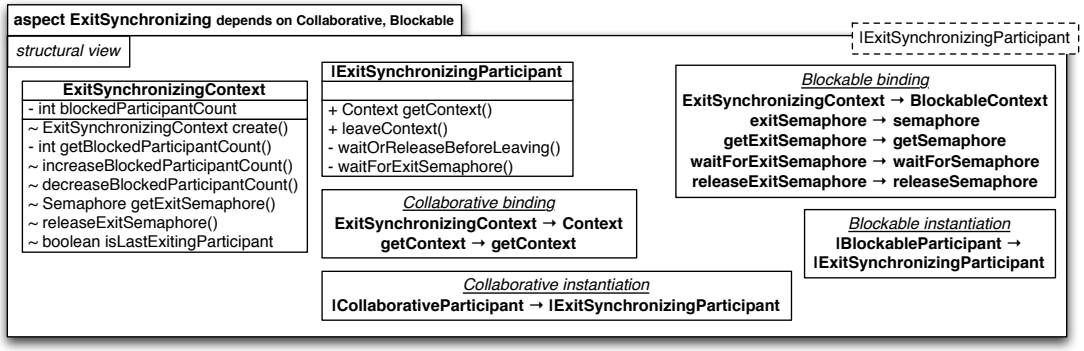

Figure 78: Structural view of the *ExitSynchronizing* aspect containing an binding directive that changes the names of an association and three methods of the reused *Blockable* aspect.

```
class BlockableForExitSynchronizingBinding {
  // mapping from Blockable to ExitSynchronizing fields and methods
  static Map<String, String> map = new HashMap<String, String>();
  // initialize binding upon construction of aspect
  static {
    map.put("semaphore", "exitSemaphore");
    map.put("getSemaphore", "getExitSemaphore");
    map.put("waitForSemaphore", "waitForExitSemaphore");
    map.put("releaseSemaphore", "releaseExitSemaphore");
  }
}
```

Figure 79: Helper class containing the binding from names of associations, attributes, and methods of the *Blockable* aspect to their equivalents in the *ExitSynchronizing* aspect (see Figure 78).

```
public interface BlockableContextForExitSynchronizing extends BlockableContext {
  // public method signatures from structural view
  Semaphore getExitSemaphore();
  void releaseExitSemaphore();
}
aspect BlockableContextForExitSynchronizingAspect {
  // attributes & associations from structural view
  Semaphore BlockableContextForExitSynchronizing.exitSemaphore;
  // declared methods without message views
  public Semaphore BlockableContextForExitSynchronizing.getExitSemaphore() {
    // auto-generated getter implementation
    return this.exitSemaphore;
  }
  // methods shown in message views
  public void BlockableContextForExitSynchronizing.releaseExitSemaphore() {
    BlockableContextMessages.
     releaseSemaphore(this, BlockableForExitSynchronizingBinding.map);
  }
}
```

Figure 80: Code for *ExitSynchronizing*'s reuse of *Blockable* (see Figure 78) showing the introduction of a field, a duplicated getter method and a reflective method for the message view *releaseExitSemaphore*.

5.6 LIMITATIONS & BUGS

5.6.1 *Overriding Methods*

Ensuring that overridden methods are used throughout an around advice.

In JAVA a method that is overriding a method of a superclass is only executed when the static type of the target is the class that overrode the method. Let us consider an aspect that contains a class that overrides a method that is already available as a result of a binding or instantiation directive. This overriden behavior would only take effect if the method is called on a target whose static type reflects the type used in the overriding aspect. However, calls from within methods of reused aspects would not use the type of the reusing aspect and therefore execute the unchanged behavior. To prevent this from happening we have to add an around advice for every overriden method.

We will explain this workaround using the class *CollaborativeContext* from the *Collaborative* aspect. The method `addParticipant` and its analog `removeParticipant` are already defined within the `Context` aspect but with another functionality as they refer to the single association *participant* and not to the manifold association *participants*. This means that we need to make sure that our more specific methods are called whenever the target is of type `CollaborativeContext` or of any of its subclasses. To this end we advise the execution of the existing adder and remover method in `Context` such that they call the corresponding methods in `CollaborativeContext`.

Without this technique the method `enterContext` in the class `Participant` would always call `Context.addParticipant` even for a `CollaborativeContext`. Note that this is not only an ASPECTJ limitation as it would have the same effect on standard classes with member methods. The problem arises from a clash of principles between Aspect-Orientation and strict type systems. Aspects should be oblivious and therefore the behavior of a method that is defined in one aspect may be changed by another aspect. In strictly-typed languages like JAVA however the target and therefore behavior of a method call should be guaranteed during compile time. The question whether or not this is a useful requirement is controversial. However, such cases in which methods with auto-generated default implementations infer with existing methods in superclasses might be rare, but they can easily be handled with the described technique.

5.6.2 *Restrictions on Object Names*

No support for restrictions on caller or target names.

In our model every default instantiation directive of a message view binds the name of the caller and the name of the target of the advised call to the wildcard character *. Any other instantia-

```
void around(CollaborativeContext target, CollaborativeParticipant participant) :
 execution(void Context.addParticipant(..)) && target(target) && args(participant) {
  // auto-generated overriding of existing adder
  target.addParticipant(participant);
}
void around(CollaborativeContext target, CollaborativeParticipant participant) :
 execution(void Context.removeParticipant(..)) && target(target) && args(participant) {
  // auto-generated overriding of existing remover
  target.participants.remove(participant);
}
```

Figure 81: Workaround for the overridden methods `addParticipant` and *removeParticipants* ensuring that the specialized implementation of these methods is called where needed.

tion that restricts the name of the caller or target would not be supported by our mapping as AspectJ offers no possibilities to encode such restrictions.

Nevertheless, we believe that such restrictions could be implemented if they should ever be modeled. A solution could be to work with different interfaces that reflect different names and extend a common interface that is independent of the names. Such a solution would increase the complexity of the implementation significantly. It is, however, questionable how often such restrictions will be modeled as a method normally behaves identically regardless of the name of the caller or target.

5.6.3 *Automatic Information Hiding*

Reused methods are automatically hidden in the model but not in the code.

A central feature of Reusable Aspect Models is its automatic information hiding for reused aspects [1]. Let us consider a public method that is declared in a class of a reused aspect and that becomes available in a reusing aspect due to an instantiation or binding directive. The access modifier of such a method is automatically changed in the reusing aspect to *package* in order to protect it from improper use in aspects that reuse the reusing aspect. If a modeler wants to provide access to a reused method, he needs to expose the method by listing it with a public access modifier in the structural view.

As explained in Section 5.2.1.6 and 5.2.1.7 we implement the merging of classes that results from binding and instantiation directives through inheritance. For this reason all public methods of a reused class become and remain available in reusing classes that were merged with it. That means that our mapping does not support automatic information hiding yet.

We considered modifying our mapping in order to support automatic information hiding, but decided to prioritize the simplicity and understandability of the obtained code. A code generator, however, could give the modeler the possibility to decide

on a per-class basis whether he wants to generate code with or without automatic information hiding for the class in question.

If the modeler would choose to automatically hide methods of reused aspects we would need to change the way we implement instantiation and binding directives. A solution could be to associate an object of a class of a reused aspect with a class in the reusing aspect instead of merging them. As a result we would need to introduce delegating methods into the class of the reusing aspect. These methods would call the corresponding methods on the associated object and could be omitted if they are not exposed with a public access modifier in the structural view of the reusing aspect.

5.6.4 *Removing Functionality or Structure*

Functionality can only be removed indirectly and structure cannot be removed at all.

So far, neither REUSABLE ASPECT MODELS nor our mapping provide direct possibilities to explicitly remove functionality or structure. Due to the pointcut and advice syntax of message views, however, it is possible to indirectly remove behavior by listing it in the pointcut and omitting it in the advice. A concise possibility to do this is to introduce a separate method for the behavior that should be removed and to model a short message view that advises every call to this method. An example for this technique is discussed in Section 4.2.2 for the *ExitSynchronizing / SpawnSupporting* conflict resolution aspect.

If a possibility to explicitly remove behavior or structure should be included into REUSABLE ASPECT MODELS our mapping would need to be updated accordingly.

5.6.5 *Deviations from Common Patterns*

Our mapping imposes additional restrictions on RAM to avoid ambiguity.

At various points our mapping relies on specific syntax that is common but not necessary for REUSABLE ASPECT MODELS . If a model does not follow the slight assumptions and restrictions that we made, especially for the content of message views or interference criteria, it will be hard or even impossible to automatically obtain code using our mapping. Therefore we believe that a code generator that realizes our mapping should explicitly list these restrictions in guidelines or even provide dynamic interface dialogues that explain and resolve possible ambiguity problems. One example could be to allow a modeler to extend the list of supported classes and interfaces from the JAVA library whenever the code generator encounters a class whose methods are not detailed in message views.

6

USING THE ASPECTOPTIMA IMPLEMENTATION

We completed the code that we obtained from the application of our mapping to the AspectOPTIMA case study by adding implementational details that were not modeled and by providing interfaces through which our code can be used in existing environments or in new projects. In order to test our implementation we created test cases for each aspect and two descriptive examples that demonstrate how to use our code.[1]

In the following section we present two interfaces and an example for the part of AspectOPTIMA that supports conventional transactions and in the succeeding section we show how to use AspectOPTIMA's Open Multithreaded Transactions. We tried to evaluate the comprehensiveness of our mapping by measuring how much code was obtained from our mapping and how much code was added or modified afterwards and we present the results in Chapter 8.

6.1 ASPECTOPTIMA

The part of AspectOPTIMA that provides facilities for conventional transactions can be used through two interfaces. An annotation interface allows users of the framework to annotate their existing source code using the annotations provided by AspectOPTIMA without calling a single method of the framework. Additionaly a programmatic interface is provided that allows users to explicitly begin, commit, or abort transactions.

6.1.1 *Annotation Interface*

If a user does not want to change existing source code he has the possibility to annotate his code using the Java Annotations that are automatically created by our mapping. The annotated source code can still be compiled using a conventional Java compiler and the resulting behavior will be identical to the behavior that would have been produced from the source code without annotations. However, if a user compiles the annotated source code with an AspectJ compiler and provides a reference to the AspectOPTIMA source code the transaction support will automatically be woven into the existing code.

1 The complete model and implementation for AspectOPTIMA is available at www.cs.mcgill.ca/~joerg/SEL/RAM.html.

Although a user could use partial functionality by annotating his code with annotations that correspond to reused aspects we assume that in most cases it will be sufficient to use the annotations provided by the *Transaction* aspect. These annotations are `TransactableClass` for classes whose fields should be saved and restored for transactions, `TransactableMethod` for methods that represent operations that should be transacted, and `TransactionParticipantClass` in case a user wants other classes than `java.lang.Thread` to be a participant of a transaction. All annotations contain configuration parameters in order to account for different update strategy and concurrency control variants.

In addition to these annotations that we obtained automatically from our mapping we provide two more annotations that ease the use of the framework. `AutoTransactedClass` can be used similarly to `TransactableClass` in order to annotate classes whose content should be saved and restored. The difference to `TransactableClass` is that we added an AspectJ instruction that adds a `TransactableMethod` annotation to every method of the class so that all operations on that class are automatically transacted. The second annotation that we added is `AbortTransactionOnThrowing` that takes a class that extends `java.lang.Throwable` as a parameter. If a method is marked with this annotation the corresponding transactions are automatically aborted whenever a `Throwable` object that is an instance of the class that is provided as parameter is thrown.

6.1.2 *Method Interface*

A second possibility to access the functionality provided by the AspectOPTIMA framework for conventional transactions is to use our small API. The class `AspectOPTIMA` in the package `ca.mcgill.sel.aspectoptima.interfacing` provides static methods that begin, commit, or abort a transaction. The different update strategy and concurrency control variants are selected by passing corresponding string constants to the methods. To explicitly begin a transaction that is a nested child of the current transaction the static method `beginChildTransaction` is provided. An extract from the JavaDoc for this class and its methods is shown in Figure 89 in the appendix.

The programmatic interface is an extension to the annotation interface as users are still required to mark classes whose content should be taken into account in transactions with the `TransactableClass` annotation. It extends the possibilities of the annotation interface as it gives users the possibility to begin, abort, or commit transactions independent of existing structure. Users that exclusively use annotations can only begin transactions when an annotated method is called.

```
@AutoTransactedClass(updateStrategy = UpdateStrategy.DEFERRING,
 concurrencyControl = ConcurrencyControl.TWOPHASELOCKING, nested = false)
public class Account implements Serializable {
  private int id;
  private float balance;
  private float limit;
  public Account() { /* needed for java.io.Serializable */ }
  protected Account(int id, float balance, float limit) {
    this.id = id;
    this.balance = balance;
    this.limit = limit;
  }
  protected int getId() {
    return this.id;
  }
  protected float getBalance() {
    return this.balance;
  }
  protected void withdraw(float amount) throws InsufficientFoundsException {
    if (this.balance - amount < limit) {
      throw new InsufficientFoundsException();
    }
    this.balance -= amount;
  }
  protected void deposit(float amount) {
    this.balance += amount;
  }
}
```

Figure 82: The banking example's `Account` class containing fields, getters and setters, and methods to withdraw or deposit money.

6.1.3 *Banking Example*

In order to demonstrate the concrete use of our transaction framework and the interfaces we provide a small example of a simplified banking scenario that supports the transfer of money from one bank account to another. Our `Account` class (shown in Figure 82) consists of an account id, a current balance and an overdraft limit. The only possible interactions with an account are operations to withdraw or deposit money. Note that all methods are automatically transacted using the deffering update strategy and the concurrency control two-phase-locking as we used corresponding parameters in the annotation `@AutoTransactedClass`.

A more fine-grained approach is used in our `Bank` class (shown in Figure 83) that manages accounts and provides a `transfer` method: We only define that the `Bank` class supports transactions by annotating it with `TransactableClass` but we decide on a per-method basis that only the `transfer` method should be transacted by annotating it with `TransactableMethod`. The `AbortTransactionOnThrowing` annotation is an additional feature of our framework that gives the user the possibility to specify that an transaction should automatically be aborted if an exception of a given type is thrown. We use it in order to define

```
@TransactableClass(updateStrategy = UpdateStrategy.DEFERRING,
 concurrencyControl = ConcurrencyControl.TWOPHASELOCKING)
public class Bank implements Serializable {
  private Map<Integer, Account> idToAccountMap = new HashMap<Integer, Account>();
  public Bank() { /* needed for java.io.Serializable */ }
  protected Bank(Collection<Account> accounts) {
    super();
    for (Account account : accounts) {
      this.idToAccountMap.put(account.getId(), account);
    }
  }
  @TransactableMethod(accessKind = AccessKind.UPDATE,
   updateStrategy = UpdateStrategy.DEFERRING,
   concurrencyControl = ConcurrencyControl.TWOPHASELOCKING, nested = false)
  @AbortTransactionOnThrowing(BankingException.class)
  protected void transfer(int sourceId, int destinationId, float amount)
      throws InsufficientFoundsException, AccountNotAvailableException {
    /* this order is unusual but necessary to show
     * that operations get "undone" on abort */
    Account destinationAccount = getAccountForId(destinationId);
    destinationAccount.deposit(amount);
    Account sourceAccount = getAccountForId(sourceId);
    sourceAccount.withdraw(amount);
  }
  private Account getAccountForId(int id) throws AccountNotAvailableException {
    Account account = idToAccountMap.get(id);
    if (account == null) {
      throw new AccountNotAvailableException();
    }
    return account;
  }
}
```

Figure 83: The `Bank` class of our banking example containing method to administrate accounts and to transfer money between them.

that the transaction should automatically be aborted whenever a `BankingException` is thrown and use this class as a superclass for all exceptions that we define ourselves.

To demonstrate how our transaction framework ensures the ACID properties in case of a successful and a failing transfer we provide a small JUnit testcase in Figure 84. In both cases we create two accounts, try to transfer money from the first to the second and use JUnit's assertion mechanism in order to ensure that the balance of both accounts complies to our expectations. In the first case the transfer of 500 monetary units should be successful because both accounts exist and the overdraft limit is not exceeded. The second case, however, tries to transfer 900 monetary units from an account with a balance of 100.10 and a limit of -500 so we expect it to fail. But as we defined in the `transfer` method that the amount should be added to the target account before it gets withdrawn from the source account we might have caused an inconsistent state if we would not have used a transaction framework. The recovery mechanism however makes sure that the `deposit` operation is "undone" after the

```
public class BankingTest {
  @Test
  public void testSuccessfull() {
    Collection<Account> accounts = new ArrayList<Account>(2);
    Account a1 = new Account(1, 100.10f, -500);
    Account a2 = new Account(2, 200.20f, -500);
    accounts.add(a1);
    accounts.add(a2);
    Bank bank = new Bank(accounts);
    try {
      bank.transfer(1, 2, 500);
      assertEquals(-399.90f, a1.getBalance(),0);
      assertEquals(700.20f, a2.getBalance(),0);
    } catch (BankingException e) {
      e.printStackTrace();
    }
  }
  @Test
  public void testFail() {
    Collection<Account> accounts = new ArrayList<Account>(2);
    Account a1 = new Account(1, 100.10f, -500);
    Account a2 = new Account(2, 200.20f, -500);
    accounts.add(a1);
    accounts.add(a2);
    Bank bank = new Bank(accounts);
    boolean insufficient = false;
    try {
      bank.transfer(1, 2, 900);
    } catch (InsufficientFoundsException e) {
      insufficient = true;
    } catch (AccountNotAvailableException e) {
      e.printStackTrace();
    }
    assertTrue(insufficient);
    assertEquals(100.10f, a1.getBalance(),0);
    assertEquals(200.20f, a2.getBalance(),0);
  }
}
```

Figure 84: A JUnit test for our banking example demonstrating a successful and a failing transfer that is automatically undone.

`withdraw` method failed and therefore the balances remain the same after this unsuccessful transaction.

6.2 OMTT EXTENSION

Our extension to the AspectOPTIMA case study that adds support for Open Multithreaded Transaction to the framework needs to be accessed through annotations and a programmatic interface as we did not discover a convincing possibility to spawn or join threads using annotations.

6.2.1 *OMTT Interface*

Analogous to the annotations for conventional transactions that we described in Section 6.1.1 the OMTT part of AspectOPTIMA provides the annotations `OMTTTransactableClass`, `OMTTTransactableMethod`, `OMTTTransactionParticipantClass`, `AutoOMTTransactedClass`, and `AbortOMTTransactionOnThrowing`. The API consists of a class `AspectOPTIMAOMTT` in the package `ca.mcgill.sel.aspectoptimaomtt.interfacing` that provides static methods that begin, join, spawn, commit, or abort an Open Multithreaded Transaction as well as methods that ease the handling of the involved threads. In order to be able to assign new jobs to threads that were spawned for another job we created the class `OMTTThread` that extends `java.lang.Thread`.

As we documented all methods of the APIs for conventional and Open Multithreaded Transactions we omit a detailed explanation in this thesis and refer the interested reader to the corresponding JavaDoc. An extract of this documentation is shown in the appendix in Figure 90 and Figure 91.

6.2.2 *Travel Agency Example*

In order to demonstrate the use of AspectOPTIMA's Open Multithreaded Transaction we created a highly simplified travel agency scenario that presents a user different travel offers for parts of a travel route. The idea of such a system is to reserve seats in the proposed trains or planes for a short time during the booking process in order to avoid cases in which a seat is not available anymore when the booking process is completed. Note that the travel agency system and the banking system that we described earlier are only very small mock-ups that try to explain the functionality of AspectOPTIMA and therefore they are far from being fully functional or complete.

The class `TravelAgency` (shown in Figure 85) provides a method `bookTravel` that allows a user to book a travel from a given point of departure to a given destination on a given date. All code within this method is wrapped into the `run` method of an anonymous class `java.lang.Runnable` and passed to the method `run` of the facade `AspectOPTIMAOMTT` in order to be run in a thread that supports Open Multithreaded Transactions. Within the code itself we begin a transaction, let a `TravelRouter` split the route into several parts and spawn a transaction participant for every route part. These spawned participants make use of a `FlightBroker` in order to find offers for every part of the overall route. Within the called method `findAndAddOffer` the corresponding seats are automatically reserved and will be booked if the transaction is completed or discarded if the transaction is aborted. The decision

```
public class TravelAgency {
  FlightBroker flightBroker = new FlightBroker();
  public void bookTravel(final String departure, final String destination, final Calendar departureDate) {
    Runnable bookTravelRunnable = new Runnable() {
      public void run() {
        try {
          AspectOPTIMAOMTT.beginTransactionInCurrentThread("DEFERRING", "TWOPHASELOCKING" ,true);
          final List<Offer> offerList = new ArrayList<Offer>();
          int routePartCount = 0;
          Future<Integer> splitRouteFuture =
            new FutureTask<Integer>(
              new Callable<Integer>() {
                public Integer call() {
              List<Route> routeList = TravelRouter.getRouteList(departure, destination, departureDate);
              int localRoutePartCount = routeList.size();
              for (final Route r : routeList) {
                Runnable findRoutePartRunnable = new Runnable() {
                  public void run() {
                    if (r.vehicleType.isPlane()) {
                      flightBroker.findAndAddOffer(offerList, r);
                    }
                  }
                };
                AspectOPTIMAOMTT.spawnAndRunTransactionParticipantFromCurrentThread(findRoutePartRunnable);
              }
              return localRoutePartCount;
              }
            }
          );
          AspectOPTIMAOMTT.spawnAndRunTransactionParticipantFromCurrentThread((Runnable) splitRouteFuture);
          routePartCount = splitRouteFuture.get();
          while (offerList.size() < routePartCount) {
            /* wait until all offers are found */
            Thread.sleep(1000);
          }
          System.out.println("We found the following travel offers for you.");
          for (Offer offer : offerList) {
            System.out.println(offer);
          }
          System.out.println("Do you want to confirm these offers and book your travel? \n" +
          "Enter \"yes\" to book. \"no\" to cancel, and \"help\" to get assistance from a representative.");
          BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
          String input = in.readLine();
          if (input.toUpperCase().equals("YES")) {
            AspectOPTIMAOMTT.commitTransactionOfCurrentAndSpawnedThreads();
          }
          else if (input.toUpperCase().equals("HELP")) {
            OMTTThread assistanceThread = getAssistanceThread();
            AspectOPTIMAOMTT.joinCurrentTransactionWithExistingThread(assistanceThread);
          }
          else {
            AspectOPTIMAOMTT.abortTransactionOfCurrentThread();
          }
        } catch (Exception e) {
          e.printStackTrace();
        }
      }
    };
    AspectOPTIMAOMTT.run(bookTravelRunnable);
  }
  ...
}
```

Figure 85: The class `TravelAgency` containing a method that allows a user to find and book offers for a travel.

```
public class TravelAgencyTest {
  @Test
  public void testSuccessfull() {
    TravelAgency travelAgency = new TravelAgency();
    int remainingSeats = travelAgency.flightBroker.remainingSeats;
    travelAgency.bookTravel("Montreal", "Paris", Calendar.getInstance());
    int newRemainingSeats = travelAgency.flightBroker.remainingSeats;
    assertEquals(remainingSeats, newRemainingSeats + 1);
  }
  @Test
  public void testFail() {
    TravelAgency travelAgency = new TravelAgency();
    int remainingSeats = travelAgency.flightBroker.remainingSeats;
    travelAgency.bookTravel("Montreal", "Paris", Calendar.getInstance());
    int newRemainingSeats = travelAgency.flightBroker.remainingSeats;
    assertEquals(remainingSeats, newRemainingSeats);
  }
}
```

Figure 86: A JUnit test for our travel agency example demonstrating a successful and a failing request whose seat reservation is automatically undone.

to accept or decline an offer is made by the user after all offers are presented to him. If he confirmed the offers the transaction is commited, if he rejected the offers the transaction is aborted, and if he asks for help a thread that is associated to a customer support representative will join his transaction.

We tested our travel agency example using a JUnit test case that we present in Figure 86. It contains successful booking request in which the number of remaining seats declines and a failed request in which the number of remaining seats stays unchanged. Note that the decision to accept or reject the offer is made by the user and is not encoded in our interactive test cases.

7 RELATED WORK

Aspect-Oriented Modeling in general and Reusable Aspect Models in particular are still considered recent approaches and the transition from the modeling phase to the implementation phase attracts comparatively little interest. Moreover the AOM community focuses mainly on model-weaving techniques that result in common models that can be implemented in languages that do not support aspect-orientation. These circumstances probably contribute to the fact that we are not aware of more than one publication that examined in detail how constructs of an Aspect-Oriented Modeling approach can be mapped to constructs of an aspect-oriented programming language.

7.1 MAPPING THEME/UML TO ASPECTJ

In 2002 Clarke and Walker [6] presented a detailed mapping from one of the most know Aspect-Oriented Modeling techniques Theme/UML [2] to AspectJ. Theme/UML is an asymmetric and more complex AOM approach than RAM, but it shares some common points with RAM and thus lead to a similar mapping. The main difference, however, is that the code for Theme/UML is less flexible when seen from a user perspective as AspectJ's recent support for Annotations could not be used yet.

Analogous to our mapping, pattern classes are implemented with interfaces and non-template operations become methods that are introduced in these interfaces. If a template operation has no supplementary behavior it is mapped to abstract methods in contrast to our mapping. Template operations with additional behavior are mapped to abstract pointcuts. That means that template methods without supplementary behavior need to be bound by implementing the corresponding abstract method with a delegating call. This is more verbose than marking existing methods with annotations but it allows solutions that adapt to incompatible signatures. Abstract pointcuts that result from template operations with supplementary behavior put no restrictions on arguments, but they require the definition of concrete pointcuts for bound methods. This means that aspects that are modeled with Theme/UML using such operations and that are implemented according to this mapping cannot be used in pure Java projects. For the reuse of systems that are modeled with RAM and implemented according to our mapping this is not the case as Annotations are pure Java.

The decisions behind the mapping for Theme/UML result in code that imposes stronger restrictions on reusing projects than code that is obtained with our mapping. Nevertheless, the mapped code for Theme/UML may be more concise and readable than our mapped code. Efforts that improve the readibilty and reuse of our code like the definition of a Domain-Specific-Language are discussed in Chapter 8.

The mapping for Theme/UML also appeared in the book *Aspect-Oriented Analysis and Design* by Clarke and Baniassad [5]. A first implementation and an extension that targets CaesarJ are discussed in an unpublished work by Jackson et al. [12]. The mapping to CaesarJ, however, is very similar to the mapping to AspectJ and it has to address various problems that arise from properties of CaesarJ but it gave us no further insights on general questions that need to be addressed when a mapping to an aspect-oriented programming language is developed.

7.2 OTHER WORK ON ALL-ASPECTUAL MAPPINGS

What impact the decision on an implementation strategy for Aspect-Oriented Modeling approaches has on maintenance has been analysed by Hovsepyan et al. [11]. One of the results of this work is the insight that the use of approaches that target aspect-oriented languages "results in smaller, less complex and more modular implementation".

Some other work is concerned with the generation of aspect-oriented code but focuses on other concerns. Groher et al. [7] for example describe an experimental aspect-oriented extension to the Unified Modeling Language and give an outlook on the generation of AspectJ code. It is unfortunate that their work is lacking a detailed description of the mapping from model artifacts to code elements.

An interesting domain-specific approach for agent-based systems is described by Kulesza and colleagues [21]. They use a Domain-Specific Language (DSL) that is based on xml schemas in order to describe aspects of agents and they mention a code generator for their DSL. Although this work looks very promising it is lacking generality in comparison to the general-purpose approach of RAM and provides no insights into the concrete mapping from the DSL to aspect-oriented code.

Beier and Kern [3] describe a general approach that generates code from UML models that were augmented with aspect information. Their idea to use code templates in order to generate code for these aspects is an interesting approach for user defined structures, but it involves more human interaction than a mapping from a fixed set of modeling constructs to code.

7.3 OTHER WORK ON ASPECT-ORIENTATION

Our mapping makes use of various Aspect-Oriented Programming techniques and patterns that have been described previously. One example is the introduction container pattern that was presented by Hanenberg and Costanza [9]. Other researchers inspired our mapping and the principles behind it or backed up our decisions with empirical studies like Hoffman and Eugster that showed that explicit interfaces facilitate reuse [10].

Related Aspect-Oriented Modeling approaches had an impact on the way we made use of RAM's features while we were modeling the AspectOPTIMA case study. Eight of these AOM approaches are evaluated in depth by Schauerhuber et al. [24] together with a conceptual reference model for Aspect-Oriented Modeling in general.

8 CONCLUSIONS & FUTURE WORK

8.1 CONCLUSIONS

This thesis presented a mapping from constructs of Reusable Aspect Models to AspectJ code. We applied our mapping to an extended and updated version of the AspectOPTIMA case study and completed the obtained code manually in order to create a working implementation.

One of our main goals during the development of the mapping was to preserve as much information that is contained in the model during the generation of code. In order to evaluate our performance we marked every line of code that was modified or added after the application of the mapping. In Figure 87 and Figure 88 we show how many lines of code (LOC) were necessary to complete the implementation of each aspect. The vertical axes show the LOC whereas the horizontal axes list each aspect of the basic AspectOPTIMA framework or its Open Multithreaded Transactions extension. It is remarkable that in both projects most aspects needed no further refinements but some technical aspects like *Lockable* or *SpawnSupporting* had to be completed with code that was around 15% to 34% percent of the size of the mapped code. Conflict resolution aspects needed no refinements at all so that we excluded them from our figures. The aspect *Copyable*, however, consists of 36 mapped and 160 manual lines of code that

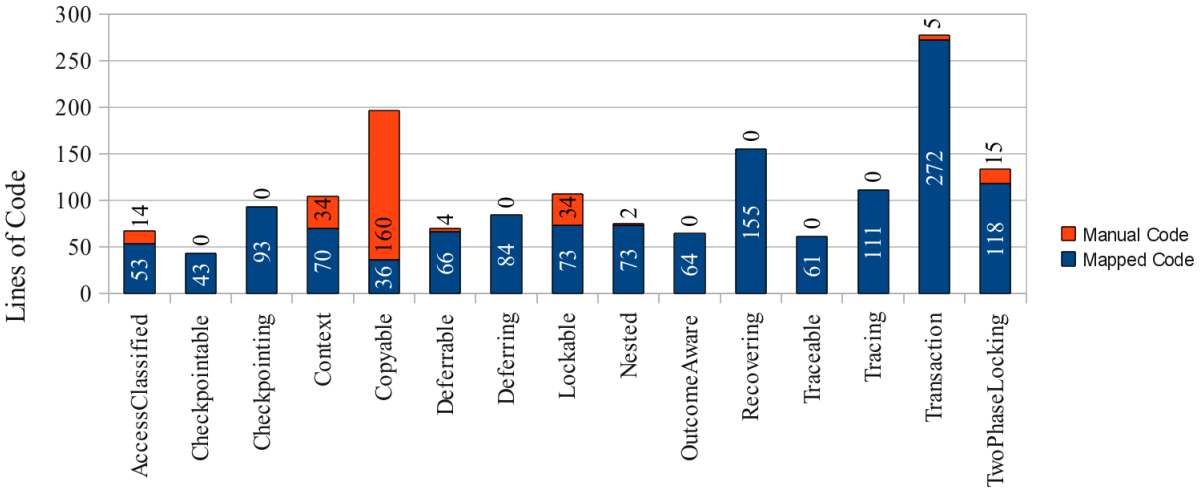

Figure 87: Diagram showing the mapped and manual lines of code (LOC) for every aspect of the AspectOPTIMA case study. The amount of mapped LOC without changes is shown in blue, manually edited or added LOC are shown in orange.

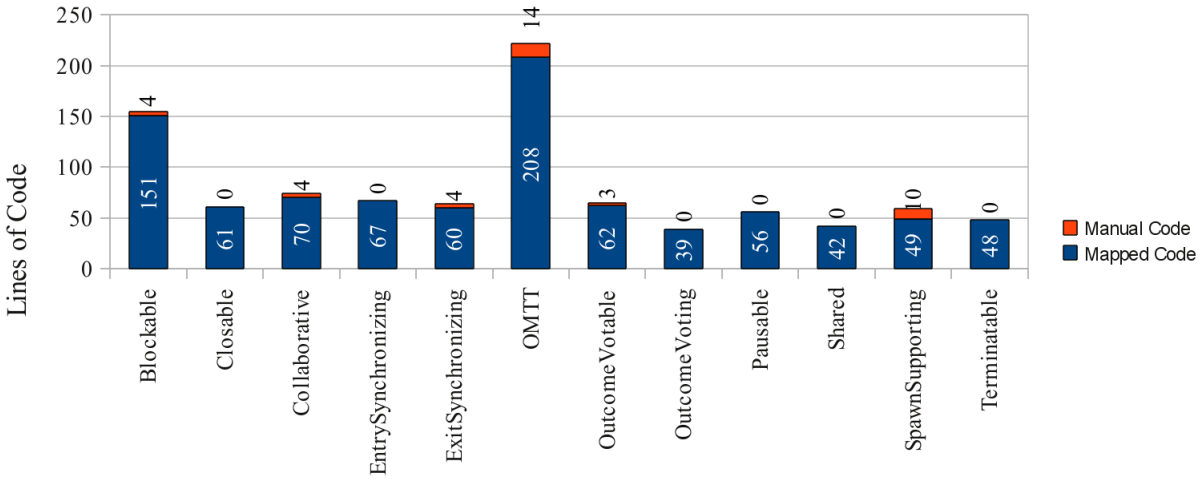

Figure 88: Diagram showing the mapped and manual lines of code (LOC) for every aspect of the OMTT extension for the ASPECT-OPTIMA case study. The amount of mapped LOC without changes is shown in blue, manually edited or added LOC are shown in orange.

probably result from the fact that recursive copies of interlinked objects and their fields can neither be implemented in a few lines nor modeled in a visual modeling language.

All together we measured that 84% of the code for the aspects that model conventional transactions and 96% of the code for Open Multithreaded Transactions was obtained by a rigorous application of the general mapping. That means that only 16% of the final code for conventional transactions and 4% of the final code for OMTTs were manually edited or added after the code generation. But, we have to mention that we updated parts of the model in order to achieve a more complete mapping and that we had to implement interfacing functionality that was not modeled with 415 LOC for conventional transactions and 455 LOC for Open Multithreaded Transactions. Because of this unmodeled interfacing code it is hard to estimate how much manual code will be necessary when our mapping is applied to other models. Nevertheless, the magnitude of the measured numbers gives us hope that a code generator for RAM could lead to significant improvements in efficiency compared to manual implementations.

8.2 FUTURE WORK

To use the full potential of our mapping future work should address the development of a code generator for RAM. Together with improved visualization tools a code generator could make it more attractive for other researchers and developers to use REUSABLE ASPECT MODELS in their projects.

Another future project could include state views into our mapping in order to provide additional runtime checks that make resulting software systems more robust. As these checks would require a mechanism for exception handling, it could be more efficient to address the question, whether a general support for exceptions should be developed for RAM, at the same time. A solution that would allow modelers to define how exceptions can be issued and handled could be of great value for developers of fault tolerant systems. We sketched a possible strategy for state view mapping in Section 5.2.2.

In order to make another feature of REUSABLE ASPECT MODELS available on the implementation level our mapping could be modified in order to support automatic information hiding. As the solution strategy that we described in Section 5.6.3 would lead to more complex code we believe that such an information hiding support should be an optional feature of a future code generator. A more elegant solution that automatically hides reused public methods without obfuscating the code could, however, be applied by default.

Even if we described a mapping that is restricted to ASPECTJ we believe that it could provide valuable insights to target other aspect-oriented programming languages. It would even be possible to develop a Domain-Specific Language (DSL) that hides technical details while providing a textual transcription of REUSABLE ASPECT MODELS. Such a DSL could serve as an intermediate representation from which various implementations for different aspect-oriented languages could be generated. The mapping to such an intermediate language could also be more concise and elegant as the question whether the obtained code is easy to read and maintain would be separated from the question how a certain feature or circumstance can be mapped to executable code. The technical details would be left to the translation process from the DSL to an existing aspect-oriented programming language and therefore the mapping could ignore these details and focus on a compact and readable representation.

Finally our mapping requires thorough testing on various projects in order to obtain a basis for a robust code generator that is of real benefit for possible users. Even if all these possible projects and improvements represent a lot of work we believe that our mapping could be a first step towards solutions that could be of worth to the Aspect-Oriented Modeling community.

APPENDIX

`public static boolean abortTransaction()`
Aborts the transaction that is associated to the currently active thread.
Returns: true if the current transaction was successfully aborted, false in case of any error

`public static boolean beginChildTransaction()`
Begins a nested transaction as a child of the transaction that is associated to the currently active thread using the configuration parameters of the parent transaction.
Returns: true if the corresponding transaction was successfully started, false in case of any error

`public static boolean beginTransaction(java.lang.String updateStrategy, java.lang.String concurrencyControl, boolean nested)`
Begins a transaction using the passed update strategy and concurrency control mechanism. Allows for nested transactions if the parameter nested is true.
Parameters: updateStrategy - "CHECKPOINTING" or "DEFERRING" concurrencyControl - "TWOPHASELOCKING" or "OPTIMISTICVALIDATION" nested - whether nested child transactions should be allowed or not
Returns: true if the corresponding transaction was successfully started, false in case of any error

`public static boolean commitTransaction()`
Commits the transaction that is associated to the currently active thread.
Returns: true if the current transaction was successfully committed, false in case of any error

Figure 89: Extract from the JavaDoc of the class *AspectOPTIMA* providing a functional interface for single-threaded transactions.

`public static boolean abortTransactionOfCurrentThread()`
Aborts the transaction that is associated to the currently active thread.
Returns: true if the current transaction was successfully aborted, false in case of any error

`public static boolean abortTransactionOfForeignThread(OMTTThread t)`
Aborts the transaction of the passed thread t.
Returns: true if the transaction was successfully aborted, false in case of any error

...

`public static OMTTThread beginAndRunTransactionInNewThread(java.lang.Runnable r, java.lang.String updateStrategy, java.lang.String concurrencyControl, boolean nested, int minParticipantCount, int maxParticipantCount)`
Begins an OpenMultithreadedTransaction in a new thread using the passed update strategy and concurrency control mechanism and sets the minimal and maximal number of participants to the provided values. Runs the passed Runnable immediately afterwards in the newly created thread. This returned new OMTTThread provides methods for enqueueing new jobs. Allows for nested transactions if the parameter nested is true. Note that the return value can simply be ignored if the new thread should never execute anything else than the passed Runnable.
Parameters: updateStrategy - "CHECKPOINTING" or "DEFERRING" concurrencyControl - "TWOPHASELOCKING" or "OPTIMISTICVALIDATION" nested - whether nested child transactions should be allowed or not
Returns: the new OMTTThread (extends Thread) in which the transaction was started if it was started successfully, null in case of any error

...

`public static boolean beginTransactionInCurrentThread(java.lang.String updateStrategy, java.lang.String concurrencyControl, boolean nested, int minParticipantCount, int maxParticipantCount)`
Begins an OpenMultithreadedTransaction using the passed update strategy and concurrency control mechanism and sets the minimal and maximal number of participants to the provided values. Allows for nested transactions if the parameter nested is true.
Parameters: updateStrategy - "CHECKPOINTING" or "DEFERRING" concurrencyControl - "TWOPHASELOCKING" or "OPTIMISTICVALIDATION" nested - whether nested child transactions should be allowed or not
Returns: true if the corresponding transaction was successfully started, false in case of any error

...

`public static boolean commitTransactionOfCurrentAndSpawnedThreads()`
Commits the OpenMultithreadedTransaction that is associated to the currently active thread.
Returns: true if the current transaction was successfully committed, false in case of any error

`public static boolean commitTransactionOfCurrentThread()`
Commits the OpenMultithreadedTransaction that is associated to the currently active thread.
Returns: true if the current transaction was successfully committed, false in case of any error

...

`public static boolean joinAndRunTransactionWithExistingThread(java.lang.Runnable r, OMTTThread threadToJoin, OMTTThread joiningT)`
Lets the thread that is passed as joiningThread join the OpenMultithreaded-Transaction of the thread that was passed as threadToJoin. Runs the passed Runnable immediately afterwards in the joining thread.
Parameters: threadToJoin - the thread that is associated to the OpenMultithreadedTransaction that should be joined joiningT - the thread that should join the OpenMultithreadedTransaction of threadToJoin
Returns: true if the corresponding transaction was successfully joined, false in case of any error

Figure 90: Extract (1/2) from the JavaDoc of the class *AspectOPTIMAOMTT* interface for Open Multithreaded Transactions.

`public static boolean joinTransactionWithCurrentThread(OMTTThread threadToJoin)`
Lets the currently executing thread join the OpenMultithreadedTransaction of the thread that was passed as an argument.
Parameters: threadToJoin - the thread that is associated to the OpenMultithreadedTransaction that should be joined
Returns: true if the corresponding transaction was successfully joined, false in case of any error

`public static boolean joinTransactionWithExistingThread(OMTTThread threadToJoin, OMTTThread joiningThread)`
Lets the thread that is passed as joiningThread join the OpenMultithreadedTransaction of the thread that was passed as threadToJoin.
Parameters: threadToJoin - the thread that is associated to the OpenMultithreadedTransaction that should be joined joiningThread - the thread that should join the OpenMultithreadedTransaction of threadToJoin
Returns: true if the corresponding transaction was successfully joined, false in case of any error

`public static <T> void mapKeyToThread(T key, OMTTThread thread)`
Maps the passed key object to the passed thread for futher reference.

`public static <T> OMTTThread getThread(T key)`
Returns the thread that is mapped to the passed key.

`public static boolean run(java.lang.Runnable runnable)`
Runs the passed runnable in a new OMTTThread. This method is the entry point to the OMTT framework and every code that uses OMTT features must have been passed to his method in a runnable or it must have been executed by the framework itself through spawned threads.
Parameters: runnable -
Returns: true if the code was successfully run in a new OMTTThread, false in case of any error

`public static void setMinParticipantCountInCurrentThread(int min)`
Sets the minimal number of participants that is required to begin a transaction to the passed value for the transaction of the current participant.
...
`public static OMTTThread spawnAndRunTransactionParticipantFrom CurrentThread(java.lang.Runnable r)`
Spawns a participant of an OpenMultithreadedTransaction using the transaction that is associated to the currently executing thread. Runs the passed Runnable in the newly spawned thread.
Returns: the new OMTTThread (extends Thread) that is associated to the spawned participant if it was started successfully, null in case of any error

...
`public static OMTTThread spawnRunAndReturnTransactionParticipant FromExistingThread(java.lang.Runnable r, OMTTThread spawningT)`
Spawns a participant of an OpenMultithreadedTransaction using the transaction that is associated to the thread that was passed as an argument. Runs the passed Runnable in the newly spawned thread.
Returns: the new OMTTThread (extends Thread) that is associated to the spawned participant if it was started successfully, null in case of any error

`public static OMTTThread spawnTransactionParticipantFromCurrentThread()`
Spawns a participant of an OpenMultithreadedTransaction using the transaction that is associated to the currently executing thread.
Returns: the new OMTTThread (extends Thread) that is associated to the spawned participant if it was started successfully, null in case of any error

Figure 91: Extract (2/2) from the JavaDoc of the class *AspectOPTIMAOMTT* interface for Open Multithreaded Transactions.

Interfaces:
java.util.Collection → java.util.HashSet
java.util.List → java.util.ArrayList
java.util.Map → java.util.HashMap
java.util.Set → java.util.HashSet

Classes:
java.lang.reflect.Method
java.lang.Thread
java.util.ArrayList
java.util.concurrent.Semaphore
java.util.HashMap
java.util.HashSet
java.util.LinkedList
java.util.Stack

Figure 92: List of JAVA library classes and interfaces that are automatically reused if a complete class with the same name contains only methods with signatures that are specified in the JAVA class or interface.

LIST OF FIGURES

BIBLIOGRAPHY

[1] Wisam Al Abed and Jörg Kienzle. Information hiding and aspect-oriented modeling. In *Proc. of the 14th Int. Workshop on Aspect-Oriented Modeling*, Denver, CO, USA, 2009.

[2] Elisa Baniassad and Siobhan Clarke. Theme: An approach for aspect-oriented analysis and design. In *ICSE '04: Proc. of the 26th International Conference on Software Engineering*, pages 158–167, Washington, DC, USA, 2004. IEEE Computer Society.

[3] Georg Beier and Markus Kern. Aspects in uml models from a code generation perspective. 2002.

[4] Gv̈en Bölükbası. Aspectual decomposition of transactions. Master's thesis, McGill University, Montréal, Québec, Canada, 2007.

[5] Siobhàn Clarke and Elisa Baniassad. *Aspect-Oriented Analysis and Design*. Addison-Wesley Professional, 2005.

[6] Siobhán Clarke and Robert J. Walker. Towards a standard design language for aosd. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 113–119, New York, NY, USA, 2002. ACM.

[7] Iris Groher and Stefan Schulze. Generating aspect code from uml models. In *Workshop on Aspect-Oriented Modeling with UML @ AOSD*, 2003.

[8] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, 1983.

[9] Stefan Hanenberg and Pascal Costanza. Connecting aspects in aspectj: Strategies vs. patterns. In *First AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, Enschede, The Netherlands, 2002.

[10] Kevin Hoffman and Patrick Eugster. Towards reusable components with aspects: an empirical study on modularity and obliviousness. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 91–100, New York, NY, USA, 2008. ACM.

[11] Aram Hovsepyan, Riccardo Scandariato, Stefan Van Baelen, Yolande Berbers, and Wouter Joosen. From aspect-oriented

models to aspect-oriented code?: the maintenance perspective. In *AOSD '10: Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, pages 85–96, New York, NY, USA, 2010. ACM.

[12] Andrew Jackson, Niall Casey, and Siobhán Clarke. Mapping design to implementation. AOSD-Europe-TCD-D111 www.aosd-europe.net.

[13] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, volume 1241, pages 220–242, 1997.

[14] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *ECOOP 2001 â Object-Oriented Programming*, Lecture Notes in Computer Science, pages 327–354, Berlin / Heidelberg, Germany, 2001. Springer.

[15] Jörg Kienzle. *Open Multithreaded Transactions: A Transaction Model for Concurrent Object-Oriented Programming*. PhD thesis, Ecole Polytechnique FÃ©dÃ©rale de Lausanne, 2001.

[16] Jörg Kienzle, Alfred Strohmeier, and Alexander Romanovsky. Open multithreaded transactions: Keeping threads and exceptions under control. *Object-Oriented Real-Time Dependable Systems, IEEE International Workshop on*, 0:197, 2001.

[17] Jörg Kienzle, Jeff Gray, Dominik Stein, Walter Cazzola, Omar Aldawud, and Tzilla Elrad. *11th International Workshop on Aspect-Oriented Modeling*. Springer, Berlin / Heidelberg, Germany, 2008.

[18] Jörg Kienzle, Wisam Al Abed, and Jacques Klein. Aspect-oriented multi-view modeling. In *AOSD '09: Proc. of the 8th ACM international conference on Aspect-oriented software development*, pages 87–98, New York, NY, USA, 2009. ACM.

[19] Jörg Kienzle, Ekwa Duala-Ekoko, and Samuel Gélineau. *AspectOptima: A Case Study on Aspect Dependencies and Interactions*, pages 187–234. Springer, Berlin / Heidelberg, Germany, 2009.

[20] Jacques Klein and Jörg Kienzle. Reusable aspect models. In *Proc. of the 11th Int. Workshop on Aspect-Oriented Modeling*, Nashville, TN, USA, 2007.

[21] Uira Kulesza, Alessandro Garcia, and Carlos Lucena. Generating aspect-oriented agent architectures. In *Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design*, 2004.

[22] Ramnivas Laddad. *AspectJ in Action: Enterprise AOP with Spring Applications*. Manning Publications Co., Greenwich, CT, USA, 2009.

[23] Gunter Mussbacher. *Aspect-Oriented User Requirements Notation: Aspects in Goal and Scenario Models*, pages 305–316. Springer, Berlin / Heidelberg, Germany, 2008.

[24] A. Schauerhuber, W. Schwinger, E. Kapsammer, W. Retschitzegger, M. Wimmer, and G. Kappel. A survey on aspect-oriented modeling approaches. 2007.

DECLARATION

I hereby declare that this thesis and all results presented in it are my original work and have not been submitted in any form to another university or educational instution for any award. Where information was derived from the published or unpublished work of others this has been acknowledged.

Karlsruhe, Germany, September 2010

Max E. Kramer