Reusable Aspect Models

Jacques Klein[1] and Jörg Kienzle[2]

[1] University of Luxembourg, Luxembourg
Jacques.Klein@uni.lu
[2] School of Computer Science, McGill University, Montreal, Canada
Joerg.Kienzle@mcgill.ca

Abstract. This paper presents an approach for specifying reusable aspect models that define structure (using class diagrams) and behavior (using sequence diagrams). The high degree of reusability of the aspect models is demonstrated by modeling the design of 8 inter-dependent aspects of the AspectOptima case study. Based on this experience, several modeling language features that we deem essential to support reusable aspect modeling are identified.

1 Introduction

Aspect-oriented software development (AOSD) techniques aim to provide systematic means for the identification, separation, representation and composition of crosscutting concerns. Aspect-oriented ideas can be applied at any phase and at any level of abstraction during software development. Aspect-oriented modeling (AOM) focuses on modularizing and composing crosscutting concerns within software *models* — models that can be used to describe or analyze properties of a system under development.

It has been shown that aspect-oriented programming techniques can be effectively used to increase code reuse. Even very general concerns such as distribution [1], concurrency [2], persistency [3] and failures [4] have been successfully implemented in an application-independent aspect, and then later composed with different applications. Reuse of *many* aspects within one application, however, has proven to be more challenging, since aspects can have complex dependencies and interactions. Currently, many researchers are working on aspect-oriented programming languages features that could make such reuse easier.

In this paper we propose to take a look at the aspect reuse problem from an aspect-oriented modeling point of view. We believe that the insights gained at the modeling level can provide deeper understanding of the fundamental problems of aspect reuse, and that the solutions we propose might be transferable to the programming language level as well. We demonstrate the validity of our approach by modeling the design of AspectOptima, a complex aspect framework that was created as a case study for studying aspect dependencies and interactions.

The structure of the rest of the paper is as follows: section 2 presents the class diagram and sequence diagram composition approaches that our aspect models are based on as well as a brief overview of AspectOptima; section 3 presents our reusable aspect models; section 4 applies our modeling approach to 8 aspects of the AspectOptima case study; section 5 discusses the reusability of our models and presents modeling language features that we deem essential to support reusable aspect modeling; section 6 presents related work and the last section concludes and mentions directions of future research.

2 Background

2.1 Weaving of Class Diagrams

The symmetric model composition technique proposed by France et al. [5,6] supports merging of model elements that present different views of the same concept. The model elements to be merged must be of the same syntactic type, that is, they must be instances of the same metamodel class. An aspect view may also describe a concept that is not present in a target model, and vice versa. In these cases, the model elements are included in the composed model.

Currently, their composition tool focuses mainly on the merging of class diagrams. The signature of a class consists only of its name, and thus the attributes and operations can be used to define different views of the class. Attributes and operations match if and only if they have the identical syntactic properties. Associations match if they have the same role names at their association ends.

2.2 Weaving of Sequence Diagrams

In [7], Klein et al. propose a semantic-based weaving of scenarios, where the weaving is based on the dynamic semantics of the models used. In [7], the scenarios are modeled with Message Sequence Charts (MSCs), but since the semantics of UML 2.X Sequence Diagrams (SDs) are largely inspired by MSCs, the scenario weaver can be easily adapted to SDs [8].

An aspect in this approach is defined as a pair of SDs: one SD serves as a pointcut (specification of the behavior to detect), and one serves as an advice (representing the expected behavior at the join point). Similarly to AspectJ, where an aspectual behavior can be inserted "around", "before" or "after" a join point, an advice may complement the matched behavior or replace it with a new behavior. When an aspect is defined with sequence diagrams, some advantages related to sequence diagrams are preserved. In particular, it is easy to express a pointcut as a sequence of messages.

2.3 AspectOPTIMA

AspectOptima [9] is an aspect framework that implements the ACID properties (atomicity, consistency, isolation and durability) of transactional objects. Its latest version [10] defines 28 individually reusable aspects that can be combined in different ways to implement various transaction models, concurrency controls and recovery strategies. We are going to use example aspects from AspectOptima to demonstrate the reusability of our aspect models.

3 Reusable Aspect Modeling

In our approach, an aspect can define *structure* using the class diagram composition approach presented in section 2.1, as well as *behavior* using the sequence diagram composition approach presented in section 2.2. It should be noted here that we think of aspects as *concerns* that are reused in potentially many places throughout one application (or several different ones). Therefore, any functionality that is *reusable* is modeled as an aspect in our approach.

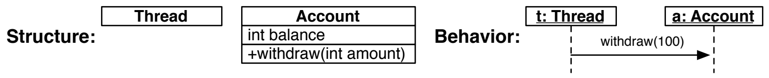


Fig. 1. A Simple Application Model

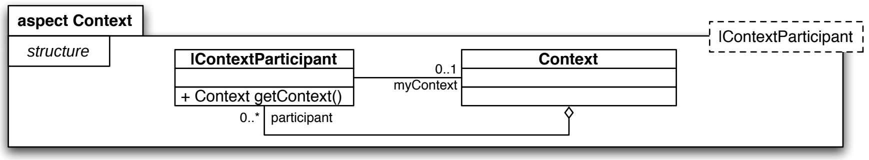


Fig. 2. The *ContextParticipant* Aspect Model

3.1 Model Instantiation

Similar to the Theme/UML approach of Clarke et al. [11] or the composition technique proposed by France et al. [5,6], we use UML templates with template parameters to keep our aspect models as generic as possible. Instantiating a (generic) aspect model involves binding the aspect model's template parameters to target model-specific elements. The resulting *context-specific aspect* model can then be composed with a target model. The template parameters that need to be bound are textually identified with a preceding vertical bar, and graphically depicted by a dotted rectangle in the top right corner of the aspect model. Pattern-matching techniques can be used to specify one-to-many bindings to target model elements.

3.2 Aspect Model Examples

Fig. 1 defines a simple application model of a bank, where a *Thread* instance *t* calls the *withdraw* method of an *Account* instance *a*. Fig. 2 shows the aspect model of *Context*, a simple AspectOptima aspect that defines a permanent association between a (to be determined) class and the class *Context*. The class *|ContextParticipant* is a template parameter of the aspect model. To bind the class *|ContextParticipant* to the *Thread* class in the application model, the designer has to instantiate the template element as `|ContextParticipant` → `Thread`. As a result, the *Thread* and the instantiated *|ContextParticipant* classes are composed to yield a new *Thread* class that has an associated *Context* instance *myContext* as well as a `getContext()` method.[3]

Fig. 3 shows the aspect model of *Copyable*, an aspect that provides the functionality of creating a duplica of an object. This is a more elaborate example, since the aspect model defines behavior in form of a sequence diagram. For each message `clone()` between a *Caller* and a *Copyable* instance, the sequence diagram weaver adds behavior that creates a new *Copyable* instance and copies the state of the original object to the copy.

[3] The full aspect model of *Context* defines in addition behavior to create and leave contexts, and associate *ContextParticipants* with *Contexts*. These models have been omitted for space reasons.

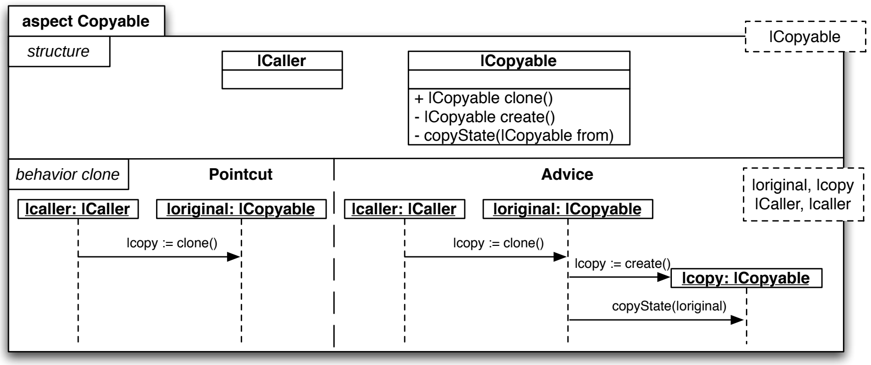


Fig. 3. Model of the *Copyable* Aspect

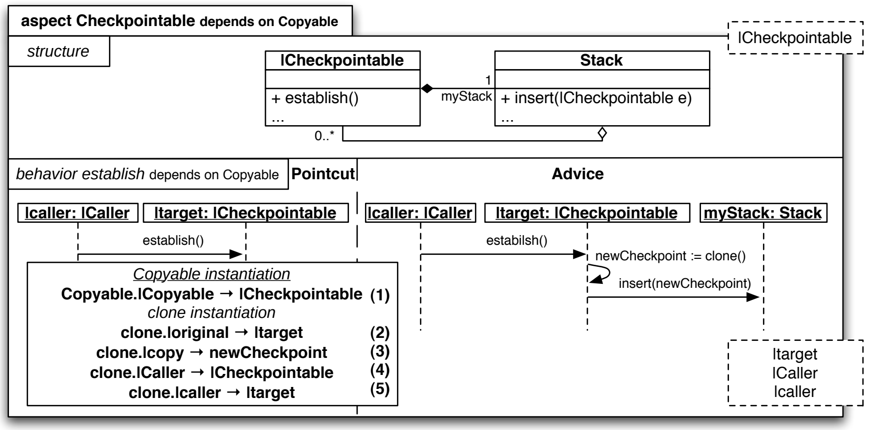


Fig. 4. *Checkpointable* Depends on *Copyable*

3.3 Expressing Aspect Dependencies

Some aspects might depend on the structure or behavior provided by other aspects. AspectOptima, for example, defines a *Checkpointable* aspect that provides objects with the functionality to *establish*, *restore* and *discard checkpoints* of their state. When *establish* is invoked, *Checkpointable* relies on the behavior provided by *Copyable* to create an identical copy of the object. Such a dependency can be expressed at the model level by *instantiating* the required aspect within the dependent aspect as shown in Fig. 4 [4].

The *structure model* of *Checkpointable* shows that a checkpointable object has an associated stack of checkpoints. The *establish behavior model* describes that when the `establish()` operation of a checkpointable object is invoked, the object first clones itself and then inserts the new copy into its stack of checkpoints. By invoking `clone()` on itself, *Checkpointable* depends on the checkpointable object to be copyable as well. To make sure that this is the case, *Copyable* is in-

[4] For space reasons, only the *establish* behavior of *Checkpointable* is shown in Fig. 4

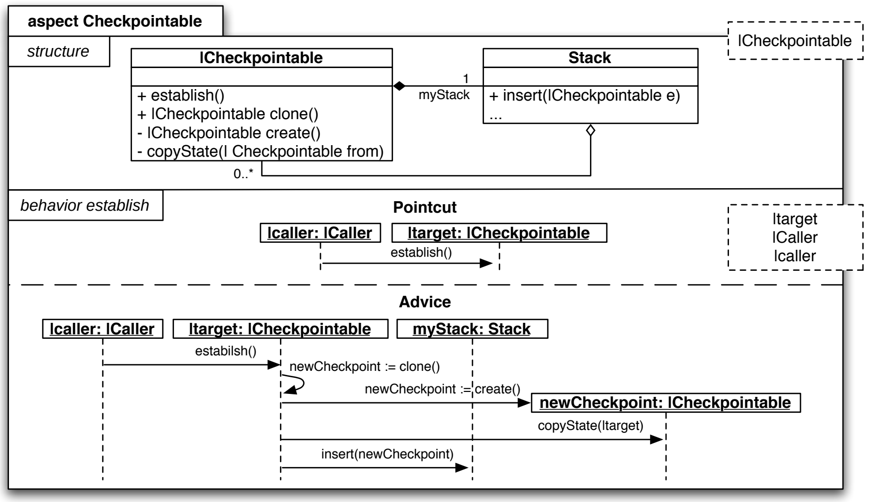


Fig. 5. The Independent *Checkpointable* Aspect (after weaving of *Copyable*)

stantiated within *Checkpointable* by binding all template parameters of *Copyable* to *Checkpointable*-specific elements as shown in Fig. 4: (1) *Copyable* applies to all *Checkpointable* (meaning that all the structure and behavior specified by *Copyable* is also applied to *Checkpointable*); (2) The *original* instance to be cloned is the instance *target*; (3) The *Caller class* of the instance invoking `clone()` is a *Checkpointable*; (4) The *caller* instance invoking `clone()` is *target*; (5) The returned copyable instance of `clone()` named *copy* is the checkpointable instance *newCheckpoint*. As a result, *Checkpointable* can call the `clone()` method present in *Copyable* on the *Checkpointable* instance *target* without explicitly specifying the `clone()` method in the structure of *Checkpointable*.

3.4 Weaving Inter-dependent Aspects

Before an aspect A that depends on an aspect B can successfully be woven with an application model, B has to be correctly instantiated and *woven into aspect* A itself to create an *independent model*. This weaving is not different from any other weaving. After all template parameters of B have been bound to concrete elements of A according to the instantiation directives, the pointcut of B is matched against the advice of A, and any occurrences of the pointcut within the advice of A are composed with the advice of B.

Fig. 5 shows the result of weaving *Copyable* into *Checkpointable*. The resulting aspect model is not dependent on *Copyable* anymore, since all the structure and behavior defined in *Copyable* is already included in the model. This model, however, is only shown for illustration purpose. It defeats the idea of reuse. If, in the future, changes are made to the *Copyable* aspect, this independent *Checkpointable* model does not take advantage of these changes. To fully exploit reuse, aspect dependencies should be kept unresolved until the aspects are woven with the final application model.

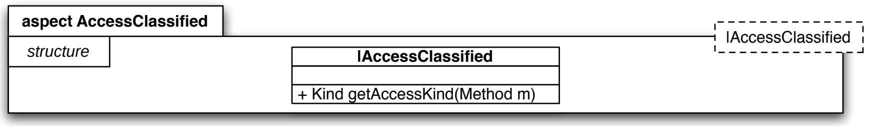


Fig. 6. The *AccessClassified* Aspect

4 Modeling of AspectOPTIMA

In this section we illustrate the reusability of our aspect models by presenting more of AspectOptima. For space reasons it is impossible to show all 28 aspects models. We are going to concentrate in this section on the modeling of the *Checkpointing* aspect, which is provided in AspectOptima as one of the strategies to implement atomicity for transactions. *Checkpointing* makes sure that when a thread working within a checkpointing context tries to modify a checkpointable object for the first time, a checkpoint of the object is made automatically and stored with the context.

Checkpointing depends directly on the aspects *Tracing*, *Context* and *Checkpointable*, and indirectly on *Traceable*, *Copyable* and *AccessClassified*. The models for *Context*, *Copyable* and *Checkpointing* have already been shown in the previous sections. The following subsections are therefore going to present *AccessClassified*, *Traceable* and *Tracing*.

4.1 AccessClassified

AccessClassified provides the functionality to classify the operations of an object into *read* operations, i.e. operations that only read the state of the object, and *write* operations, i.e. operations that modify the state of the object. As shown in Fig. 6, this functionality is encapsulated in the method `kind getAccessKind(Method m)`. Since this functionality does not involve any message sending between objects, the model of *AccessClassified* only defines structure.

4.2 Traceable

The *Traceable* aspect encapsulates the structure and behavior necessary to create a trace of an operation invocation. A trace instance contains all important information about an operation invocation: for checkpointing we are mainly interested in storing the access kind of the operation. The access kind is determined using the functionality offered by *AccessClassified*: the instantiation directives within *getTrace* declare that all *Traceable* objects have to be *AccessClassified* as well.

4.3 Tracing

The *Tracing* aspect monitors object accesses made by context participants that are associated with the context. It stores a trace of all operation invocations made on traceable objects. The model of the *Tracing* aspect depends on *Context* and *Traceable* as shown in Fig. 8. The *traceMethod* behavior specifies the interactions that create a new trace *before* any operation *m* is executed on a *Traced* object. *Tracing* also defines the behavior for a method `boolean wasModified(Object o)` that consults the current trace to determine if, within this context, a context participant has executed a modifying operation on the object *o*.

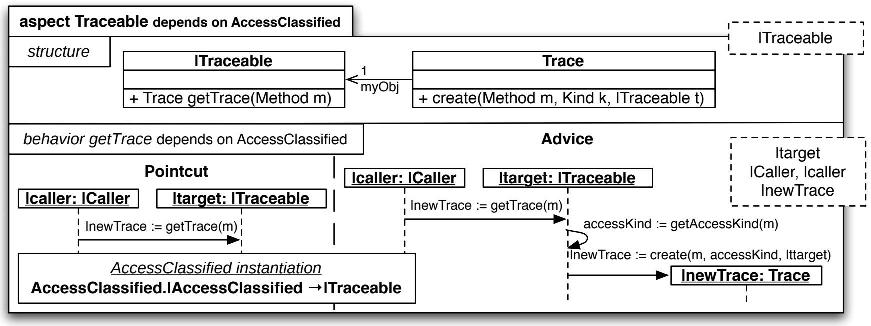

aspect Traceable depends on AccessClassified
structure
ITraceable
+ Trace getTrace(Method m)
1
myObj
Trace
+ create(Method m, Kind k, ITraceable t)
ITraceable
behavior getTrace depends on AccessClassified
Advice
Pointcut
lcaller: ICaller
ltarget: ITraceable
lnewTrace := getTrace(m)
AccessClassified instantiation
AccessClassified.IAccessClassified → ITraceable
lcaller: ICaller
ltarget: ITraceable
lnewTrace := getTrace(m)
accessKind := getAccessKind(m)
lnewTrace := create(m, accessKind, ltarget)
lnewTrace: Trace
ltarget
ICaller, lcaller
lnewTrace

Fig. 7. The Traceable Aspect

4.4 Checkpointing

Checkpointing makes sure that the first time a *ContextParticipant* invokes a modifying operation on a checkpointable object within the context, a checkpoint of the object is automatically established. The intersting fact about *Checkpointing* is its dependency on *Tracing*. *Checkpointing* does not remember the objects that it checkpoints. It relies on the trace collected by *Tracing*. It is therefore essential for all methods that are monitored by *Checkpointing* to also be traced by *Tracing*. This is achieved by instantiating *Tracing* within *Checkpointing*, and by specifying in the instantiation directives that the method that *Tracing* is supposed to trace is the same method that *Checkpointing* is monitoring (see (1) in Fig. 9).

4.5 Applying Checkpointing

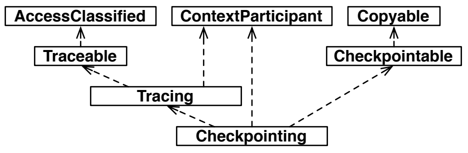


Fig. 10. Aspect Model Dependencies

Before a reusable aspect A can be instantiated and woven with a target model, all aspects that it depends on must be instantiated and woven into the advice of A. If these aspects also have dependencies, then these have to be resolved first. The instantiation and weaving order for *Checkpointing* can be deduced from the dependency hierarchy depicted in Fig. 10. Fig. 11 shows the structure and behavior that is obtained when applying *Checkpointing* to the bank application model of Fig. 1.

5 Discussion on Reusability

The AspectOPTIMA case study illustrates reusability very well. Any of the aspects can be reused in a different application whenever the functionality it provides is needed.

In the full design of AspectOPTIMA, many of the above aspect models are reused within others. For example, *Shared*, an aspect that guarantees single-writer/multiple-reader access to objects also depends on *AccessClassified*. *Versioned*, an aspect that can create multiple context-local copies of objects, also depends on *Copyable*. *2-Phase Locking*, *Deferring* and *Recovering*, three aspects that help providing isolation and atomicity to transactions, depend on *Tracing*.

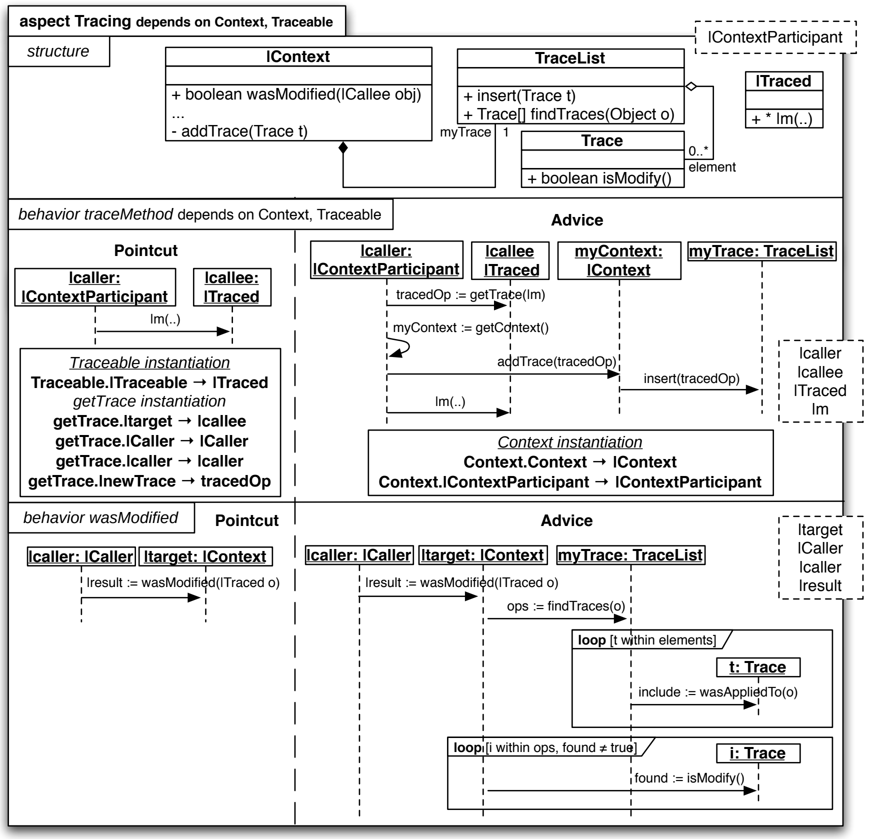


Fig. 8. The *Tracing* Aspect

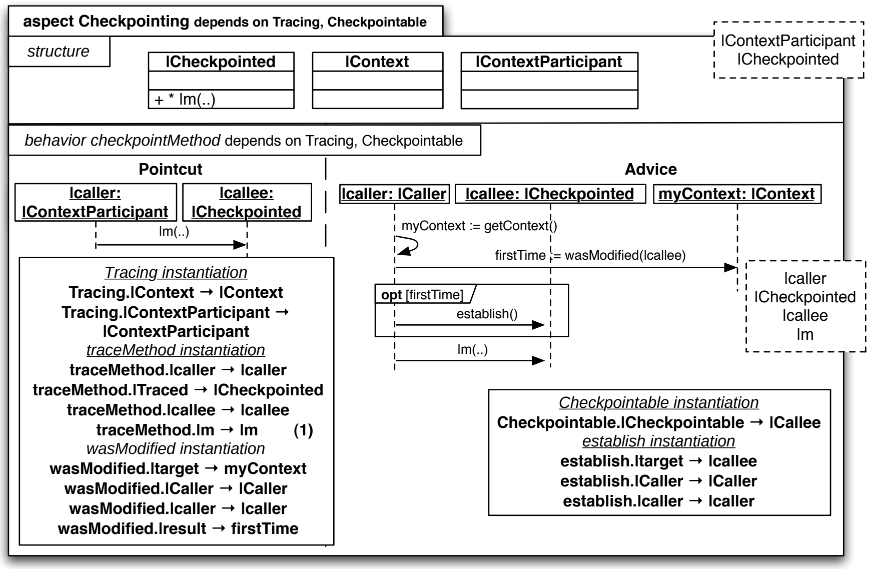
Finally, of course, all aspects are reused in different configurations when they are combined to create different transaction models. Unfortunately, space limitations forbid us to describe these models and their dependencies in detail. The interested reader is referred to [10].

Reuse within aspects only creates unidirectional dependencies: even if an aspect B reuses A, A does not depend on B and can be reused in isolation.

Finally, we want to note that in a real-world application, our aspect-oriented modeling approach would be used in combination with pattern-matching techniques when specifying aspect bindings. For instance, the following instantiation specifies that *all* method invocations on *Account* objects must be checkpointed: Checkpointing.|ContextParticipant → Thread, Checkpointing.|Checkpointed → Account, Checkpointing.|Caller→*, Checkpointing.|caller→*, Checkpointing.|m→*.

5.1 Modeling Features Required for Reusability

Based on our experiments with AspectOptima and on findings reported by many other researchers, we have identified several modeling language features that we believe are mandatory to support reusable aspect modeling:



aspect Checkpointing depends on Tracing, Checkpointable
structure
ICheckpointed
+ * Im(..)
IContext
IContextParticipant
IContextParticipant
ICheckpointed
behavior checkpointMethod depends on Tracing, Checkpointable
Pointcut
Icaller: IContextParticipant
Icallee: ICheckpointed
Im(..)
Tracing instantiation
Tracing.IContext → IContext
Tracing.IContextParticipant → IContextParticipant
traceMethod instantiation
traceMethod.Icaller → Icaller
traceMethod.ITraced → ICheckpointed
traceMethod.Icallee → Icallee
traceMethod.Im → Im (1)
wasModified instantiation
wasModified.Itarget → myContext
wasModified.ICaller → ICaller
wasModified.Icaller → Icaller
wasModified.Iresult → firstTime
Advice
Icaller: ICaller
Icallee: ICheckpointed
myContext: IContext
myContext := getContext()
firstTime := wasModified(Icallee)
opt [firstTime]
establish()
Im(..)
Icaller
ICheckpointed
Icallee
Im
Checkpointable instantiation
Checkpointable.ICheckpointable → ICallee
establish instantiation
establish.Itarget → Icallee
establish.ICaller → ICaller
establish.Icaller → Icaller

Fig. 9. The *Checkpointing* Aspect

- **Template Parameters**: Template parameters can be used in a reusable aspect model as generic placeholders for structure or behavior expected by the target model.
- **Binding / Instantiation**: In order to use a reusable aspect model within a target model, the template parameters have to be bound to elements from the target model. This instantiation of the reusable aspect model creates a context-specific aspect model.
- **Separate Binding**: The instantiation directives that specify the binding do not belong into the aspect model itself, since they are different for each target model. Bindings using wildcards or more powerful pattern matching techniques should be provided to support one-to-many bindings.
- **Inter-Aspect Dependency Declaration**: The modeling formalism must support to express inter-aspect dependencies. If aspect A depends on reusable aspect B, A must provide instantiation directives for B.
- **Aspect-Aspect Weaving**: It should be possible to compose an aspect model with an application model, but also with another aspect model. This is mandatory to allow aspects to be reused within other aspects.
- **Template Parameter-Preserving Weaving**: If an aspect B is instantiated and woven into a reusable aspect model A that defines template parameters, the template parameters of A should still be present in the resulting aspect model. This is mandatory to preserve genericity and hence reusability.
- **Dependency-Consistent Weaving**: When a reusable aspect model A is instantiated and woven with a target model, the weaver first recur-

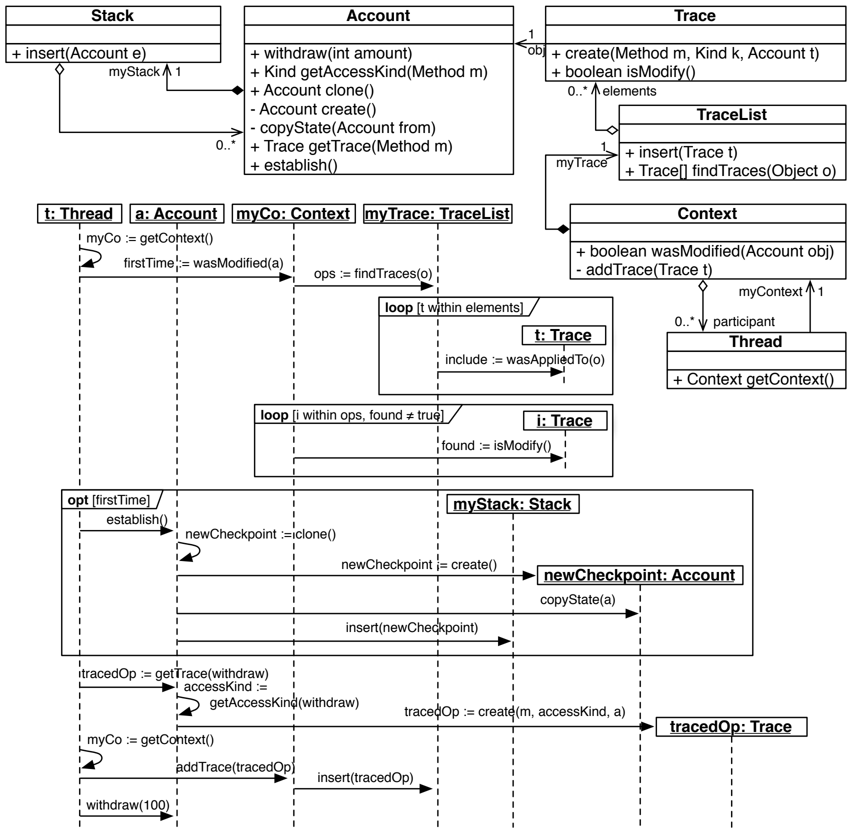


Fig. 11. Application Model after applying *Checkpointing*

sively instantiates and weaves all aspects that A depends on with A (to create an independent model of A).

- **Late Dependency Resolution**: Instantiation and weaving due to dependencies is performed only during the final target model creation. Independent intermediate aspect models are never exposed.

6 Related Work

6.1 Aspect-Oriented Modeling Approaches

Our reusable aspect models are based on the class diagram weaving approach [5] and the sequence diagram weaving approach [8] presented in section 2. Other related aspect-oriented modeling approaches are briefly described in this section.

Clarke and Baniassad [11] define an approach called *Theme/UML*. It introduces a theme module that can be used to represent a concern at the modeling level. Themes are declaratively complete units of modularization, in which any of the diagrams available in the UML can be used to model one view of the structure and behavior the concern requires to execute. In Theme/UML class diagrams and sequence diagrams are typically used to describe the structure and behavior of the concern being modeled. Just like in our approach, the binding to a base model is done by template parameter instantiation. In contrast to our approach, Theme/UML does not support model weaving.

Similarly to our approach, Whittle and Araujo [12] represent behavioral aspects with scenarios. Aspectual scenarios are modeled as interaction pattern specifications and are composed with specification scenarios. The weaving process is performed in two steps. First state machines are generated from the aspects and from the specification. The weaving is then performed by composing the obtained state machines.

The Motorola WEAVR approach [13] and tool have been developed in an industrial setting. Behavior is modeled using the Specification and Description Language (SDL), a formalism related to state diagrams. In order to be able to reuse aspects, *mappings* have to be defined (equivalent to our instantiations) that link a reusable aspect to the application-specific context in which it is to be deployed.

6.2 Reusability

In [14] the authors propose *framed aspects*, an approach that uses AOP to modularize crosscutting and tangled concerns and frame technology [15] to allow aspect parameterization, configuration, and customization. In framed aspects, the identification of features (here called feature aspects), and detection of dependencies and interferences is performed following the high-level feature interaction approach promoted by FODA [16]. Once this is done, the features are modularized within *framed aspects*, together with their dependencies.

Framed aspects are made up of three distinct modules: the framed aspect code (normal and parameterized aspect code), composition rules (aspect dependencies, acceptable and incompatible aspect configurations), and specifications (user-specific customization). These modules are composed to generate customized aspect code using a frame processor. Framed aspects achieve separate aspect bindings and aspect dependencies through parameterization and composition rules respectively. Composition rules can also be used for specifying acceptable and incompatible aspect configurations. The above mentioned constructs enable framed aspects to be reused in contexts other than that for which they were implemented.

7 Conclusion and Future Work

We have presented in this paper an approach for specifying reusable aspect models that define structure (using class diagrams) and behavior (using sequence diagrams). We have demonstrated the high degree of reusability of our aspect models by modeling 8 inter-dependent aspects of the AspectOptima case study. Based on this experience, we identified several modeling language features that we deem essential to support reusable aspect modeling.

Since our approach currently only allows the use of sequence diagrams to express behavior, we are limited in our behavioral modeling to describe message exchanges / method invocations only. We are planning to investigate how to integrate state-based modeling approaches such as [13]. We are also interested in extending our approach to address aspect interference, i.e. when two aspect models that specify contradicting behavior are applied to the same target model elements. Again, AspectOptima can be used as a testbed, since [10] highlights 11 interference problems. Finally, we believe that in order to obtain target models with well-defined public interfaces, instantiation directives should offer control over method visibility. More research is required to determine exactly how this could be accomplished.

References

1. Soares, S., Laureano, E., Borba, P.: Implementing distribution and persistence aspects with AspectJ. In: Proceedings of OOPSLA, ACM Press (2002) 174–190
2. Cunha, C.A., Sobral, J.L., Monteiro, M.P.: Reusable aspect-oriented implementations of concurrency control patterns and mechanisms. In: AOSD 2006, ACM Press (2006) 134 – 145
3. Rashid, A.: Aspect-Oriented Database Systems. Springer-Verlag (2004)
4. Kienzle, J., Guerraoui, R.: AOP - Does It Make Sense? The Case of Concurrency and Failures. In Magnusson, B., ed.: 16th European Conference on Object–Oriented Programming – ECOOP 2002. Number 2374 in Lecture Notes in Computer Science, Malaga, Spain, Springer Verlag (2002) 37 – 61
5. France, R., Ray, I., Georg, G., Ghosh, S.: Aspect-oriented approach to early design modelling. IEE Proceedings Software (August 2004) 173–185
6. Reddy, R., Ghosh, S., France, R.B., Straw, G., Bieman, J.M., Song, E., Georg, G.: Directives for composing aspect-oriented design class models. Transactions on Aspect-Oriented Software Development **3880** (2006) 75–105
7. Klein, J., Hélouet, L., Jézéquel, J.M.: Semantic-based weaving of scenarios. In: AOSD 2006, ACM Press (2006) 27–38
8. Klein, J., Fleurey, F., Jézéquel, J.M.: Weaving multiple aspects in sequence diagrams. Transactions on Aspect Oriented Software Development (2007) To appear
9. Kienzle, J., Gélineau, S.: AO Challenge: Implementing the ACID Properties for Transactional Objects. In: Aspect-Oriented Software Development – AOSD 2006, ACM Press (2006) 202 – 213
10. Bölükbaşi, G.: Aspectual Decomposition of Transactions. Master's thesis, School of Computer Science, McGill University, Montreal, Canada (2007)
11. Clarke, S., Baniassad, E.: Aspect-Oriented Analysis and Design: The Theme Approach. Addison Wesley (2005)
12. Whittle, J., Araujo, J.: Scenario modelling with aspects. IEE Proceedings Software **151** (2004) 157–171
13. Cottenier, T., v.d. Berg, A., Elrad, T.: Stateful aspects: the case for aspect-oriented modeling. In: 10th Aspect-Oriented Modeling Workshop, ACM Press (2007)
14. Loughran, N., Rashid, A.: Framed aspects : Supporting variability and configurability for aop. In: In International Conference on Software Reuse (ICSR-8), Springer Berlin/Heidelberg (2004) 127–140
15. Bassett, P.: Framing Software Reuse: Lessons from the Real World. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1997)
16. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, S.: Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University (1990)