Modeling the Evolution of Aspect Configurations using Model Transformations

Uwe Zdun, Mark Strembeck
Institute of Information Systems, New Media Lab
Vienna University of Economics, Austria
{uwe.zdun|mark.strembeck}@wu-wien.ac.at

ABSTRACT

In this paper we introduce an approach to address the evolution of aspect configurations with model transformations. We use model transformation diagrams (MTDs) to define valid behavioral model states of a system as well as valid transitions between those states. MTD transformations can be used to define evolutionary changes in the weaving process of an aspect-oriented system. To allow for a straightforward incorporation of aspects in UML models, we extend UML2 activity diagrams with joinpoint start and end nodes. In this paper, each model state in an MTD refers to an extended UML2 activity diagram.

1. INTRODUCTION

In recent years a number of approaches for UML-based modeling of aspects have been proposed. Some approaches are extending the UML using a UML profile (see e.g. [6, 2]), others perform a meta-model extension, i.e. they extend the UML familiy of languages with new language elements (see e.g. [10, 4]). So far these approaches focus on mapping the elements of aspect-oriented environments (mainly the concepts are based on AspectJ [7]) to UML modeling elements. That is, the focus is on representing aspects in UML models.

The effects of applying aspects – i.e. how a model evolves if an aspect is woven – have only been marginally in focus of aspect-oriented modeling approaches so far. This concern, however, is important to be considered for a number of situations:

- In the early stages of system design we need to translate requirements into classes and aspects. In particular, we require some approach to show the evolution from a non-aspect-orineted model to an aspect-oriented model, as well as the interactions between the aspect-oriented and non-aspect-oriented parts of the system.

- Often a number of different aspect configurations can be woven for one and the same system. That is, the aspects woven into the system can be changed either at compile-time, load-time, or runtime – depending on the used aspect weaving mechanism. For example, consider a logging aspect, which is woven into the debugging environment only, but not into the productive system. Here, the evolution options resulting from the weaving time for the aspect configurations and their corresponding effects should be modeled as well.

- Often aspects have interdependencies or interactions among each other, a concern which of course should be modeled. For instance, consider a persistence aspect is allowed to be woven, but only if a storage device aspect is woven as well.

To address these problems, this paper proposes an approach to model the behavioral evolution of aspect configurations in software systems using model transformation diagrams. In other words, we use a model transformation to represent the aspect weaving step. The model transformation diagrams are an extension to UML 2.0. In particular, they model the aspect weaving dependencies via model transformations between different UML Activity Diagrams. Here, the Activity Diagrams show the behavior in the system with different aspect configurations. To enable the modeling of aspect-related behavior in Activity Diagrams we introduce a simple extension to Activity Diagrams for representing the start and end of the joinpoints of an aspect in the control flow.

2. THE APPROACH

In this section, we explain our model transformation diagrams, and our extension to Activity Diagrams for representing the start and end of the joinpoints of aspects.

2.1 Model Transformation Diagrams

We have defined the Model Transformation Diagrams (MTD) as a meta-model extension to the UML 2.0 standard (see Figure 1[1]). To define MTDs formally, we specify the new package *ModelTransformations*. The graphical notation of our model transformation diagrams is similar to UML2 interaction overview diagrams, however, the MTD semantics differ significantly. The UML2 interaction overview diagrams are a variant of activity diagrams and describe the flow of control between different nodes (see [9]). In contrast, our MTDs are a variant of state machines. Model transformation diagrams describe changes of specification of a software system. These changes are modeled through transitions between different diagrams. In this paper, we use only UML2 activity diagrams in the MTDs, to model transformations of the *behavioral model state*. (Please note that in our full meta-model definition, there are also structural model states, but these are not used in this paper.)

The main transition type used in MTDs are *transform transitions*. Transform transitions express that the source model state of the transition is transformed to the target model state of the transition. A transition from one behavioral model state to another means that the behavior of a certain system aspect is transformed, so that after the transition, the system behavior conforms to the state specified by the transition's target. For instance, the example transitions in Figure 2 show two model transformations between two activity diagrams: one adds a condition between the two activities, and the reverse transformations removes the condition. Figure 2 also contains

[1]Due to the page limit we do not include the full formal definition including OCL constraints of the meta-model extension here, but provide only the corresponding meta-model as an overview.

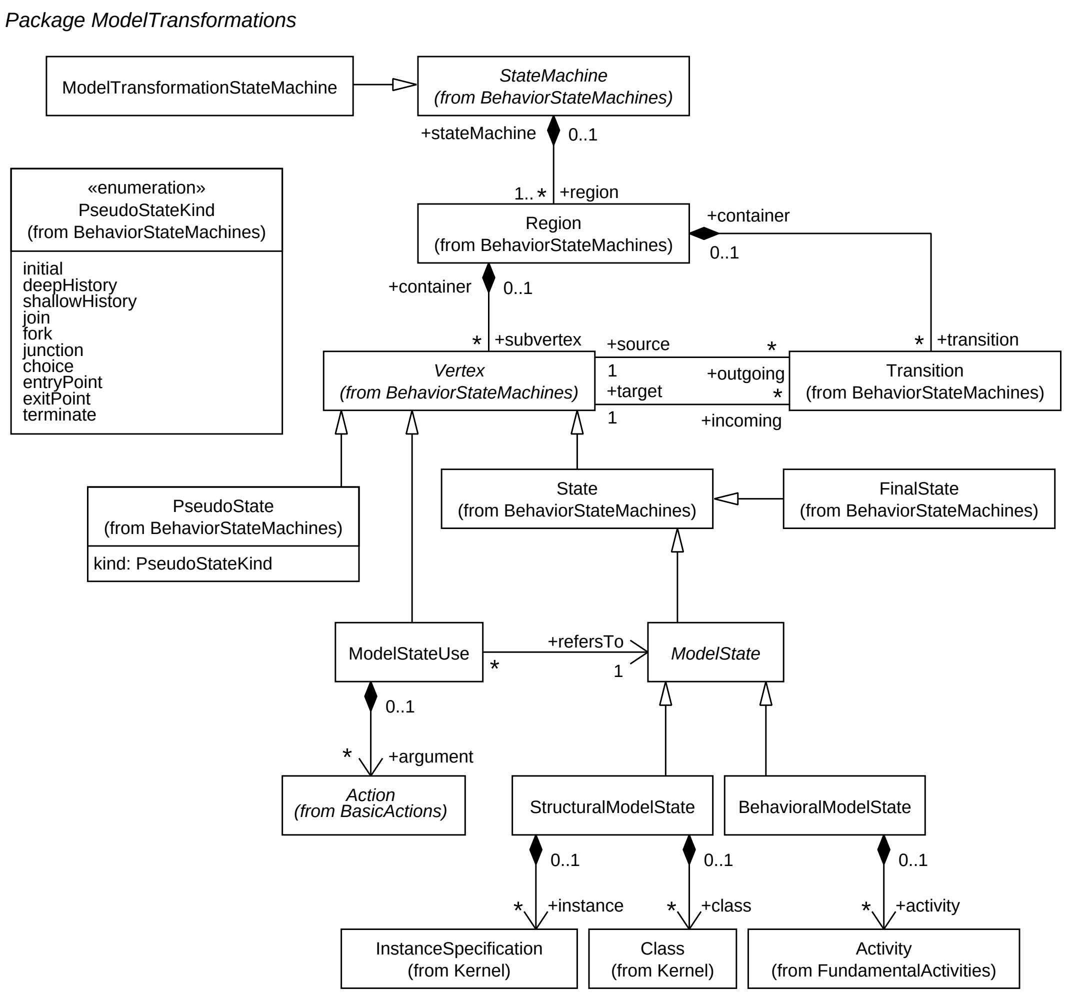

Package ModelTransformations
ModelTransformationStateMachine
StateMachine (from BehaviorStateMachines)
+stateMachine
0..1
1..*
+region
«enumeration» PseudoStateKind (from BehaviorStateMachines)
initial
deepHistory
shallowHistory
join
fork
junction
choice
entryPoint
exitPoint
terminate
Region (from BehaviorStateMachines)
+container
0..1
+container
0..1
*
+subvertex
+source
*
*
+transition
Vertex (from BehaviorStateMachines)
1
+outgoing
+target
*
1
+incoming
Transition (from BehaviorStateMachines)
PseudoState (from BehaviorStateMachines)
kind: PseudoStateKind
State (from BehaviorStateMachines)
FinalState (from BehaviorStateMachines)
ModelStateUse
+refersTo
*
1
ModelState
0..1
*
+argument
Action (from BasicActions)
StructuralModelState
BehavioralModelState
0..1
0..1
0..1
*
+instance
*
+class
*
+activity
InstanceSpecification (from Kernel)
Class (from Kernel)
Activity (from FundamentalActivities)

Figure 1: Meta-model for Message Transformation Diagrams (MTD)

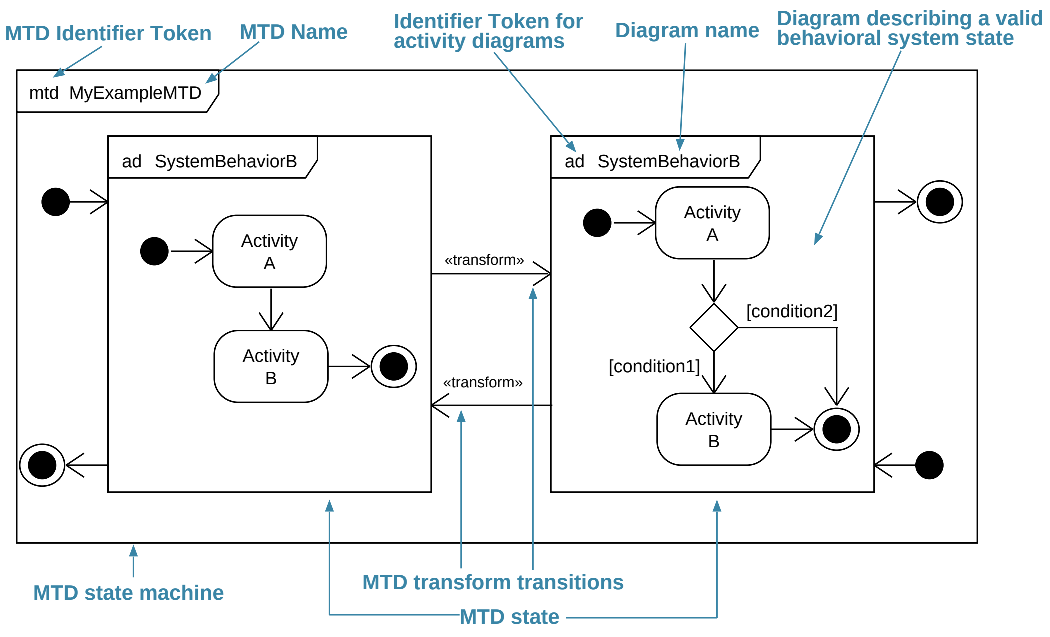


Figure 2: Informal overview for the elements of MTDs

informal explanations for our notations. A formal UML meta-model extension for MTDs can be found in [11].

In the first place, MTDs are a means to depict possible model transformations. The idea, presented in this paper, is to apply the transform transitions in the MTDs to model aspect weaving relationships. This way different behavioral model states show models of the behavior of the system in different aspect configurations. The transform transitions then show the possible ("legal") weaving steps between these model states.

2.2 Extending Activity Diagrams with Joinpoint Start and End Activities

In our approach, we model the behavior of aspects as part of the activity diagrams describing the system's behavior. That is, we show scenarios of the aspect in action. However, it is necessary to distinguish the aspect-oriented and non-aspect-oriented parts of the activity diagram. Moreover, in case more then one aspect is used, we need to distinguish different the aspects modeled in the same activity diagram.. Otherwise we would not be able to properly model aspect interactions.

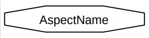
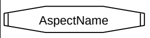
NODE TYPE	NOTATION	Explanation & Reference
JoinpointStart	 AspectName	JoinpointStart is an Activity that can be used in an Activity Diagram to indicate that the aspect "AspectName" has intercepted the control flow at this point. All subsequent steps in the Activity Diagram until a JoinpointEnd Activity with "AspectName" is reached are handled by the aspect "AspectName". Optionally, a Joinpoint Start node can have a tagged value "pointcut" that indicates the name of a pointcut designating this joinpoint. See Activity from FundamentalActivities.
JoinpointEnd	 AspectName	JoinpointEnd is an Activity that can be used in an Activity Diagram to indicate that the interception of the control flow by the aspect "AspectName" has ended. See Activity from FundamentalActivities.

Figure 3: Definition of two Activities for start and end of joinpoints in Activity Diagrams

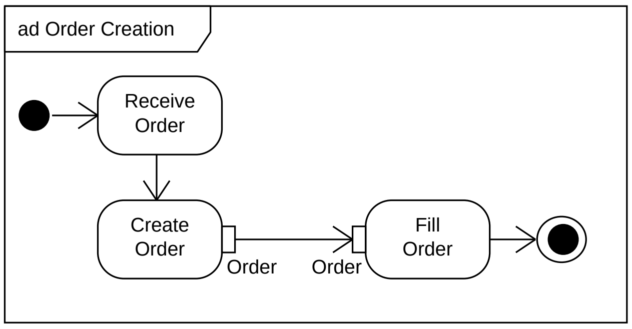


Figure 4: Activity Diagram for order creation

To address this problem, we introduce two new Activities as subclasses of the UML2 Activity meta-class (from FundamentalActivities, see [9]). JoinpointStart is an Activity that can be used in an Activity Diagram to indicate that the aspect referred to via "AspectName" has intercepted the control flow at this point. All steps in an Activity Diagram between a JoinpointStart and the corresponding JoinpointEnd Activity (referred to via the same "AspectName") are handled by the respective "AspectName" aspect. In addition it is possible for another aspect to intercept the control flow in between. In other words: JoinpointEnd is an Activity that can be used in an Activity Diagram to indicate that the interception of the control flow by the aspect "AspectName" has ended. Optionally, JoinpointStart Activities can have a tagged value "pointcut" that indicates the name of a pointcut designating this joinpoint. Figure 3 summarizes the definitions.

3. EXAMPLE: ORDER HANDLING

In this section, we consider an example from the early stages of designing an order handling system. In a first step, we design a simple activity for order creation according to the following short scenario description: when an order is received, an order object needs to be created and then the order object is filled with values. This simple control flow is shown in the activity diagram "Order Creation" in Figure 4.

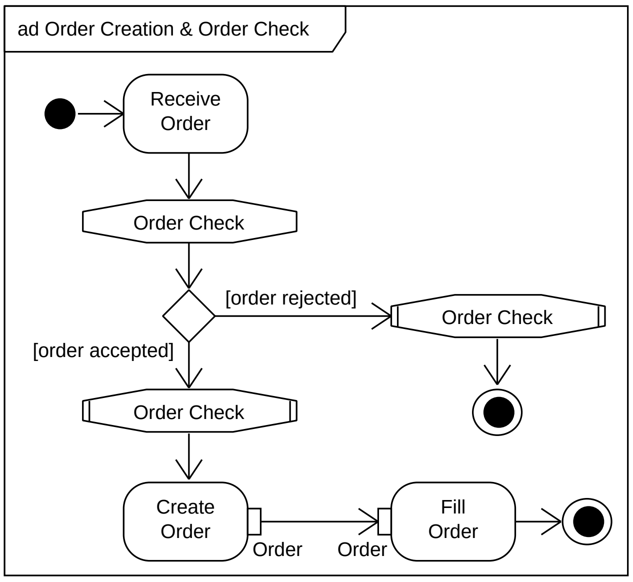


Figure 5: Activity Diagram for combining order creation with order checking

Next, we design other fundamental activities of order handling. During the ongoing design work, we realize that in some customer systems which should be used with the order handling system, a check is required, whether the order can be accepted or not. This check is not only relevant for order creation, but it must also be performed before an order is changed or re-submitted. Thus "Order Check" is a cross-cutting concern in our system and should be modeled as an aspect. To do so, we need to intercept the control flow between the Receive Order and Create Order activities. Similarly, we need to extend other activity diagrams that have joinpoints belonging to this aspect. The pointcuts for the corresponding aspect can be derived in later design stages by looking at all occurrences of the aspect's joinpoints and by defining proper (cross-cutting) designations for these points in the control flow. The woven aspect is shown in the Activity Diagram "Order Creation & Order Check" in Figure 5.

A second aspect that cross-cuts many order handling activities is "Order Persistence". This aspect needs to intercept the control flow after the order is filled in, and must call the Make Persistent Activity. The woven aspect is shown in the Activity Diagram "Order Creation & Order Persistence" in Figure 6.

For this aspect we need to consider one special case, though. If the aspect "Order Check" is configured, all rejected orders should be

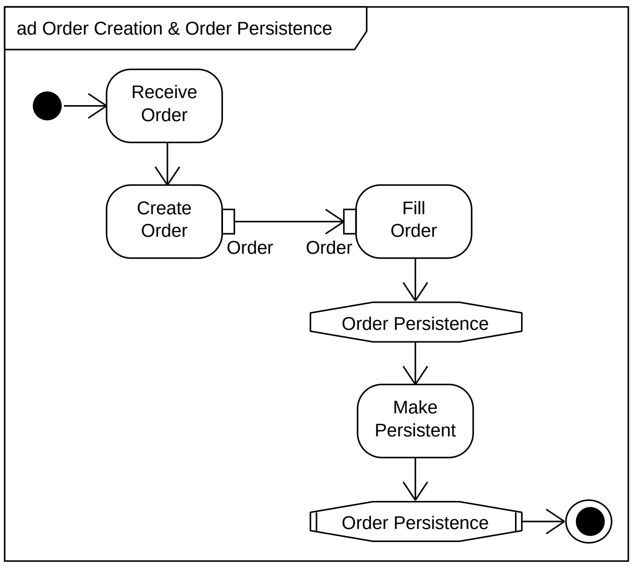


Figure 6: Activity Diagram for combining order creation with persistence

logged in the persistence store. That is, the two aspects have an interdependency among each other. Because both aspects are optional extensions, we need to model this interaction in a separate Activity Diagram "Order Creation & Order Check & Order Persistence" in Figure 7. Here, we can see that the "Order Persistence" aspect is cross-cutting the activities in this diagram. If the aspect is used, a rejected order log entry object is created, and the Make Persistent Activity is called.

Finally, we need to model the possible weaving-time aspect evolutions for this system. We use an MTD to show the possible weaving configurations for the two optional aspects described above. The diagram in Figure 8 shows that in any case the basic "Order Creation" diagram is the starting point for weaving. The aspect weaver can either weave order persistence, order checking, or no aspect. If one of the two aspects is chosen, the other aspect can optionally be woven as well. In this case, the behavioral state of the system is transformed to the Activity Diagram "Order Creation & Order Check & Order Persistence", so that the aspect interaction is modeled as well.

Please note that in this example we have shown the aspect weaving process independently of the concrete weaving time. Our approach is capable to model aspect weaving at compile-time, load-time, or runtime. Though, the MTD needs to be changed slightly if runtime weaving is supported. Runtime weaving would mean that we could turn off the aspects again. That is, we would introduce backward transformations between the model state nodes (the "mrefs" in the figure) to model runtime weaving properly.

4. RELATED WORK

Aldawud et al. [1] present a number of steps they apply to model aspect-oriented systems. In particular, they model the static system structure via class diagrams. System behavior, including aspects and crosscutting, is modeled with UML statecharts. Their approach, however, is not able to depict evolutionary changes resulting from (static or dynamic) weaving of aspects which is one of the main benefits of MTDs.

Gray et al. [3] describe an elaborated approach to support aspect-oriented domain modeling which has partially similar objectives

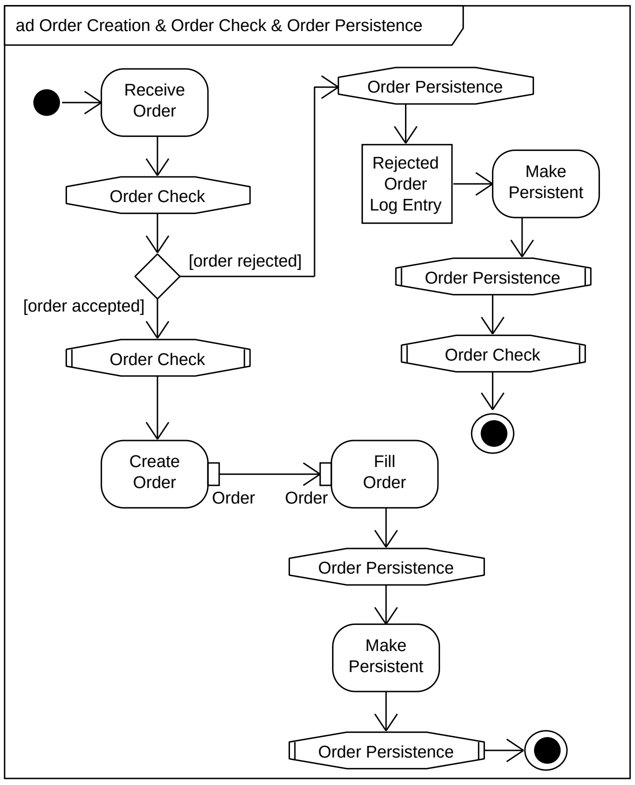


Figure 7: Activity Diagram for combining order creation with persistence and order checking

to our approach. For each modeling domain they define domain-specific weavers which operate on the abstraction layer of models (not source code). To specify these weavers they defined the embedded constraint language (ECL) as an extension to the OMG object constraint language (OCL). The ECL is used to specify transformations between models and to specify strategies that define how a concern is applied in a certain model context. ECL operates on XML files which are used to store the corresponding models and Gray et al. implemented a tool to generate C++ source code from ECL specifications.

Barros and Gomes [2] use UML2 activity diagrams to model crosscutting in aspect-oriented development. They define a new composition operation they call "activity addition" via an UML profile. Activity additions are used for weaving a crosscutting concern in a model. In particular, they define two stereotypes to mark certain nodes in activity diagrams that define the so called interface nodes which are then used to merge two or more activity diagrams, and the so called subtraction nodes which define what nodes need to be removed from a given activity diagram.

Jezequel et al. [5] represent crosscutting behavior using contract and aspect models in UML. They model contracts using UML stereotypes, and represents aspects using parameterized collaborations equipped with transformation rules expressed with OCL constraints. OCL is used in the transformations for navigating instances of the UML meta-model.

Han et al. [4] present an approach to support modeling of AspectJ language features to narrow the gap between implementations based on AspectJ and the corresponding models. Mahoney and Elrad [8] describe a way to use statecharts and virtual finite state machines to model platform specific behavior as crosscutting concerns. They

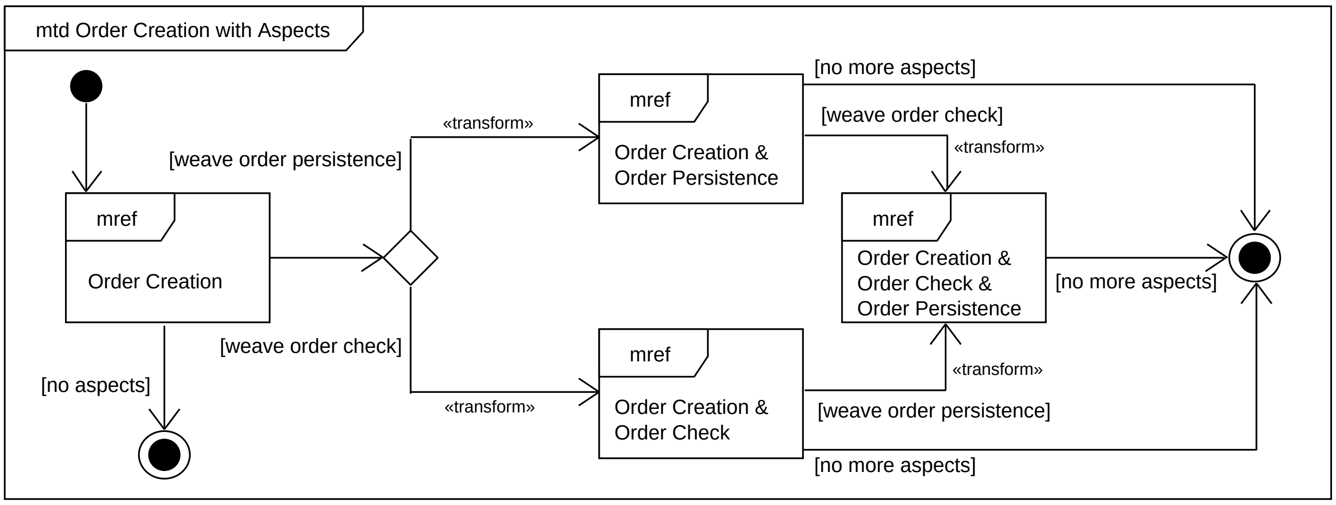

mtd Order Creation with Aspects
mref
Order Creation
[no aspects]
[weave order persistence]
[weave order check]
«transform»
«transform»
mref
Order Creation & Order Persistence
mref
Order Creation & Order Check
[no more aspects]
[weave order check]
«transform»
mref
Order Creation & Order Check & Order Persistence
[no more aspects]
«transform»
[weave order persistence]
[no more aspects]

Figure 8: MTD for order creation with its aspects

especially plan to evaluate the effectiveness of their approach in a model driven development context. Tkatchenko and Kiczales [10] present an approach to model crosscutting concerns. They extend the UML with a joint point model, advice and inter-type declarations, and role bindings. Moreover, they provide a weaver to process the corresponding extensions.

5. CONCLUSION

In this paper, we briefly presented an approach to model the evolution of aspect configurations via model transformations. In particular, we defined model transformation diagrams (MTDs) as an UML2 extension. In essence, MTDs are state machines which are applied to model the evolution of software systems. Each state in an MTD refers to a model that defines a valid structural or behavioral specification of the corresponding system. Transitions between those states describe valid transformations between those models. In this paper, however, we focused on the specification of behavioral system facets to model the evolution of aspect configurations. Therefore, we additionally introduced Joinpoint start and end activities that allow for a clear separation of the aspect-oriented and non-aspect-oriented parts of a system specification, as well as the modeling of crosscutting aspects. In our future work, we will provide tool support for MTDs both on the modeling level and source code level. In addition to behavioral states, we also use structural model states in MTDs to model the evolution of structural aspect models.

6. REFERENCES

[1] O. Aldawud, A. Bader, and T. Elrad. Weaving with Statecharts. In *Proc. of the Workshop on Aspect Oriented Modeling with UML*, April 2002.

[2] J. Barros and L. Gomes. Towards the Support for Crosscutting Concerns in Activity Diagrams: a Graphical Approach. In *Proc. of the AOSD Modeling with UML Workshop*, October 2003.

[3] J. Gray, T. Bapty, S. Neema, D. Schmidt, A. Gokhale, and B. Natarajan. An Approach for Supporting Aspect-Oriented Domain Modeling. In *Proc. of the 2nd International Conference on Generative Programming and Component Engineering (GPCE),*, September 2003.

[4] Y. Han, G. Kniesel, and A. Cremers. Towards Visual AspectJ by a Meta Model and Modeling Notation. In *Proc. of the International Workshop on Aspect-Oriented Modeling*, March 2005.

[5] J. Jezequel, N. Plouzeau, T. Weis, and K. Geihs. From contracts to aspects in uml designs. In O. Aldawud, G. Booch, S. Clarke, T. Elrad, W. Harrison, M. Kande, and A. Strohmeier, editors, *Aspect-Oriented Modeling with UML*, Enschede, The Netherlands, April 2002. http://lglwww.epfl.ch/workshops/aosd-uml/index.html.

[6] M. M. Kande, J. Kienzle, and A. Strohmeier. From AOP to UML – A Bottom-Up Approach. In *Proc. of the Workshop on Aspect Oriented Modeling with UML*, April 2002.

[7] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, Oct 2001.

[8] M. Mahoney and T. Elrad. Modeling Platform Specific Attributes of a System as Crosscutting Concerns using Aspect-Oriented Statecharts and Virtual Finite State Machines . In *Proc. of the International Workshop on Aspect-Oriented Modeling*, March 2005.

[9] The Object Management Group. Unified Modeling Language: Superstructure. http://www.omg.org/technology/documents/formal/uml.htm, August 2005. Version 2.0, formal/05-07-04, Object Management Group.

[10] M. Tkatchenko and G. Kiczales. Uniform Support for Modeling Crosscutting Structure. In *Proc. of the International Workshop on Aspect-Oriented Modeling*, March 2005.

[11] U. Zdun and M. Strembeck. Modeling Composition in Dynamic Programming Environments with Model Transformations. In *Proc. of the 5th International Symposium on Software Composition*, Vienna, Austria, March 2006. LNCS, Springer-Verlag.