

Towards a Dynamic Join Point Model for UML Action Semantics

Rafael Chaves and Luiz Carlos Zancanella

Departamento de Informática e Estatística
Universidade Federal de Santa Catarina
88040-900 – Florianópolis, SC, Brazil
{chaves, zancanella}@inf.ufsc.br

Abstract. This work acknowledges the potential benefits of mixing the use of the Model Driven Software Development and Aspect-Oriented Software Development, two different strategies for separating concerns. The model driven approach proposes a clear separation between concerns belonging to the problem domain and those related to implementation details, providing a fast track for mapping executable models into running applications. The aspect-oriented paradigm offers an elegant mechanism for expressing how different concerns relate to each other, allowing for a higher degree of modularity. We believe that together, models and aspects provide the necessary framework for building software that is much easier to maintain. In this context, we propose an approach for combining the two technologies based on a new dynamic join point model for UML action semantics.

1 Introduction

Model Driven Software Development (MDSD) and Aspect-Oriented Software Development (AOSD) approaches make a perfect match. Both approaches tackle the issue of separation of concerns with different, complementary strategies. MDSD proposes a clear separation between concerns belonging to the problem domain and those related to implementation details by handling them at different levels of abstraction. AOSD offers powerful, non-invasive mechanisms for expressing how different concerns relate to each other, allowing for a higher degree of modularity. Together, AOSD and aspects provide a solid framework for building software that is easier to understand, modify and extend.

The Model Driven Architecture (MDA) is an OMG initiative that aims to provide a uniform conceptual model and technology set for model driven development. For modeling, the obviously recommended notation is UML, the modeling language adopted as a standard and maintained by OMG since 1997. UML received a significant addition to incorporate semantics for executable actions and procedures, or *action semantics*, resulting in the version 1.5 of the specification [11]. Such change was required for UML to provide the necessary foundation for model driven development. With action semantics, UML became capable of representing models

that contain behavior and structure. These complete models are referred to in the context of this paper as *executable models* [3].

The aspect-oriented paradigm is still a recent technology. Most of the developments have occurred in the programming languages front. Today several production-quality programming languages and frameworks that support aspects are available. However, the research community is still looking for ways to bring aspects into the modeling arena, more specifically to UML. Proposals abound [1,2,16,17,18,19,20], but we still don't have a winner. No consensus has been reached so far on how to represent aspects or crosscutting concerns as UML model elements.

Also, most of the efforts in the area of aspect-oriented modeling did not take advantage of UML's ability of representing behavior, mostly because such ability was not available until very recently. The result was a lack of a sound and precise specification for dynamic join points. With OMG's Model Driven Architecture gaining momentum, more and more proposals for using aspects in the context of model driven software development are being made [3,5,21,22,23].

In this context, we propose a new dynamic join point model for executable models based on UML action semantics.

The remaining of this document is structured as follows: section 2 gives some background on existing approaches for aspect oriented modeling, and the basics of UML action semantics; in section 3 we present our proposal; we finish describing other approaches that relate to our proposal and summarizing our proposal.

2 Background

2.1 Approaches for Aspect-Oriented Modeling

There are several proposals for extending UML in order to support aspect-oriented modeling. As anticipated in the introduction, none of them cover appropriately the specification of behavior in models, mostly due to the fact that an action semantics specification for UML models has been introduced just recently [11]. However, these proposals have made many valuable contributions in areas such as representation of aspects in class diagrams and structural composition of models. Following is a summary of some of the existing approaches for aspect-oriented modeling we have analyzed.

Theme/UML (Clarke et al). Theme/UML [1] is a proposal for aspect-oriented modeling with roots in Subject Oriented Programming. For each subject, a partial model (a *design subject* in Theme/UML parlance) is defined containing only those elements from the object model that are related to the corresponding requirement. *Composition patterns* are used for design subjects that represent crosscutting concerns, to be composed with base design models through a *merge* integration strategy.

The main goals of Theme/UML [1]:

- language independence, in order to support different mapping targets and the evolution of current aspect-oriented programming (AOP) languages;
- weaving at modeling level, to allow validation of composition before implementation and to support implementation in non-AOP based languages;
- compatibility with existing design techniques and tools.

In general, Theme/UML conceptual framework is sound. However, there are a few weaknesses in this approach: first, dynamic join point representation is limited to sequence diagrams; second, the notation requires the extension of UML in unsupported ways. And last, the use of UML bindings to represent that two models should be composed together might work for simple models, but may become cumbersome with bigger models [27].

Aspect-Oriented Design Model (Stein et al). Aspect-Oriented Design Model (AODM) [2,27] is intended as a technique for modeling computer systems in UML targeting an implementation in AspectJ. In order to do that, AODM brings many of the features of the AspectJ language, such as its primitive pointcut designators, to UML.

An important characteristic of AODM is that it makes use of standard UML extension mechanisms to support the creation of aspect-oriented models.

In spite of its ties with aspect-oriented programming in AspectJ, [27] shows that the join point model of AODM can support other implementation approaches, such as composition filters, adaptive programming and Hyper/J.

As Theme/UML, AODM also falls short of providing a comprehensive mechanism for dynamic join point representation. Only sequence diagrams are considered.

Other Approaches. Several other approaches [17,16,18,19,20] to aspect-oriented modeling have been proposed. Although they provide valuable contributions such as asynchronous advices [19] and pointcuts as 1st class citizens [20], most of their interesting characteristics can be found in either Theme/UML or AODM. And, once more, none of them go beyond the use of sequence diagrams to represent dynamic crosscutting.

2.2 Action Semantics

Since version 1.5 of its specification, UML includes support for executable models in the form of action semantics. This support allows the designer to assign complete meaning to active entities in a model. To that end, the designer has to use an action language, which provides capabilities similar to those offered by common programming languages but at a higher level of abstraction. UML defines the semantics such action languages have to conform to but does not specify any syntax. Instead, vendors are supposed to provide their own languages. This way, many different languages can be used as action languages, including existing 3GL, 4GL, diagram-based and domain-specific languages.

The UML action semantics are defined around a small set of concepts:

- **actions** are elementary units of behavior that take a set of input values, perform some computation, and produce a set of output values as result. Actions implement common operations such as reading/writing data, sending

messages to objects, creating/destroying objects, evaluating conditions, etc. Actions are often grouped, so they can perform more interesting operations together. Loops and conditional statements are examples of groups of actions where one action controls the execution of sub-actions;

- **procedures** are pieces of behavior defined by a set of actions that can be attached to behavioral elements in UML models, such as operations, state transitions and constraints. Procedures can take arguments provided by requesters and produce results once they are finished executing;
- **pins** are channels for data transmission between actions. The data produced by an action is made available through its **output pins**, which are connected to other actions' **input pins**. The connection of an input pin of an action to an output pin of another action establishes a flow of data between the two actions. **Data flows** implicitly impose ordering between actions so that the action producing the data has to complete before the action consuming the data can be started;
- finally, **control flows** allow for explicit sequencing of actions which otherwise would be inherently concurrent.

For illustration purposes, let's see a fragment of an executable model expressed using UML action semantics. Imagine that one wishes to specify the behavior for the operation `Account.withdraw()`. The pseudocode for such operation could look like:

```
class Account
  attribute balance
  ...
  operation withdraw(amount)
      balance <- balance - amount
  end
end
```

Using action semantics (ad-hoc notation), the same operation could be as below:

```
WriteAttribute<attribute=balance>(
  ReadCurrentObject,
  ApplyFunction<function=sum>(
    ReadAttribute<attribute=balance>(
      ReadCurrentObject
    ),
    ReadVariable<variable=amount>
  )
)
```

The uniform and fine-grained characteristics of actions make them a good option as an intermediate representation for executable models, favoring interoperability between tools.

3 Dynamic Join Points for Executable Models

UML action semantics allows specifying complex behavior for elements such as operations or constraint expressions in a great deal of detail. The action semantics model is highly abstract, well-formed and uniform. And most importantly, the conceptual weight is minimal.

Based on that, we decided investigating the *execution of an action* as the fundamental join point¹ for executable UML models. Given the granularity and diversity of primitive actions, the approach seems promising – every single elementary atom of behavior in the base program could be exposed to crosscutting behavior.

Of course, with more power, complexity increases as well. It is possible that users might appreciate a more coarse-grained join point model closer to the syntax provided by the action language. In the majority of cases, pointcut designators specified using a front-end language will have a straightforward mapping to an action semantic based designator. Exceptions would be pointcuts related to features specific to the action language, not having similar constructs in the UML model.

3.1 Conceptual Framework

We decided to adopt in our proposal the principles and terminology around AspectJ's join point model. The reason is twofold: to leverage some of the benefits that are unique to AspectJ's model, and to allow readers familiar with that language to more easily transfer their experience with aspect-oriented programming to our model.

Join Points. The execution of any action constitutes a join point. Since every interesting behavior has a corresponding action, it is trivial to support join points commonly exposed by popular aspect-oriented programming languages such as operation invocation, attribute access/modification and object instantiation. Besides, it makes possible to expose less usual but equally interesting join points such as manipulation of associations and the iteration through elements of collections.

Pointcuts. A *pointcut* is an arbitrary subset of the join points available in the base program. A *pointcut designator* is a unary predicate that determines what join points belong to a pointcut.

Pointcut designators themselves are specified using action semantics. A pointcut designator procedure takes a join point context object as input and produces a boolean value as output, indicating whether the join point should be picked.

For example, a pointcut that contains all accesses to the *Account.balance* attribute could have the following designator (pseudocode):

¹In the context of this section, the terms *crosscutting*, *join point* and *pointcut* refer to *dynamic crosscutting*, *dynamic join point* and *dynamic pointcut* respectively.

```

action = ReadAttribute
  and action.attribute.name = "balance"
  and action.attribute.owner = #Account

```

Observe that pointcuts make use of meta-level information about the base model to figure out what join points are interesting.

Join point context. The join point context exposes meta-level information about the action being currently executed so the pointcut can evaluate whether the join point should be picked or not. This context object is also available to any advices when they are activated. The context provides access to statically determined properties, such as the action type, the model element the action targets, or even actions related to the current action (via a data flow, for example). But dynamically computed information may also be obtained and used. For instance, values passed into the action as input pins, or produced by the action through its output pins.

3.2 Comparison with AspectJ Join Point Model

Static pointcuts can be evaluated during transformation time. The type of the action being executed is the most obvious static information a pointcut designator can check. Other common static properties to check are the action-specific target element (i.e., the classifier in a *CreateObject* action, the operation in a *CallOperation* action, the attribute in a *ReadAttribute* action, etc.). The table below does a comparison between AspectJ's static pointcut designators and the corresponding action target element to be selected according to our model.

Table 1: AspectJ PCDs and the corresponding actions and elements in UML.

AspectJ PCD	UML action	target element
<i>call(<operation>)</i>	<i>CallOperation</i>	<i>operation</i>
<i>call(<type>.new)</i>	<i>CreateObject</i>	<i>classifier</i>
<i>execution(<operation>)</i>	<i>CallProcedure</i>	<i>procedure</i>
<i>get(<attribute>)</i>	<i>ReadAttribute</i>	<i>attribute</i>
<i>set(<attribute>)</i>	<i>WriteAttribute</i>	<i>attribute</i>
<i>handler(<type>)</i>	<i>Handler</i>	<i>classifier</i>

Property-based pointcuts require information that is only available during runtime. For the rest of this section, we will discuss how to map some of AspectJ's property-based pointcuts to our model. AspectJ examples were taken and adapted from [8].

***args(<type>)*.** Used in conjunction with *call* or *execution* PCDs, matches the runtime type of one or more arguments passed by a caller to a callee.

```
pointcut testEquality(Point p): target(Point)
  && args(p)
  && call(* *(*));
```

Basically this pointcut picks up any operation call where the target is a *Point* and the single argument to the operation is an instance of class *Point* as well.

```
action = CallOperation
  and action.inputs.argument.size = 1
  and action.inputs.argument[1].type isKindOf #Point
```

target(<type>). Used in conjunction with *call* or *execution* PCDs, matches the runtime type of the target.

```
pointcut setter(): target(Point) &&
  (call(void setX(*)) ||
  call(void setY(*)));
```

This pointcut picks any operation call where the operation is named *setX* or *setY*, and the target is instance of the *Point* class.

```
action = CallOperation
  and (action.operation.name = "setX"
  or action.operation.name = "setY")
  and action.inputs.target[1].type isKindOf Point
```

this(<type>). Used in conjunction with any other PCDs, matches the runtime type of the current executing object.

```
!this(Point) && target(Point) && call(* *(..))
```

The pointcut above picks all calls to any operations in instances of *Point* (or a subtype) not invoked from *Point* itself.

```
action = CallOperation
  and not(currentFrame().self isKindOf Point)
  and action.inputs.target[1].type isKindOf Point
```

3.3 Less Conventional Join Points

Loops. In [10] a proposition is made for exposing loops as join points. One of the challenges in that work is to recognize loops in compiled code. With UML action semantics, the only way to perform loops is through *Loop* action, regardless the programming style or compiler used, so this is a non-issue here.

The *Loop* action [11, §2.20.1.4] is a composite action that controls other two actions: the condition action (test) and the conditionally executed action (body). The former is

any action capable of producing a boolean result, the latter can be virtually any action (or group of actions).

Capturing the execution of the loop as whole is trivial: in the pointcut designator, we just need to check if the action is the *Loop* action.

Picking the join point related to the nested test or body is more complicated. For any action, we need to check the parent composite action currently running. If it is a *Loop* action, and it is in the *executingTest* state [11, §2.20.2.3], then the current action is a test action. If the loop action is in the *executingBody* state, then it is body action.

From this description it is not hard to imagine that capturing join points for loops can be prohibitively expensive if not done in association with a type based (static) check.

Exception Throwing. AspectJ has a primitive PCD for handlers of exceptions. With UML action semantics, it is also possible to capture the moment the exception is thrown as shown below. The corresponding action is the *Jump* action, and the exception object is available through its input pins [11, §2.25.5.3].

```
action = Jump
  and action.inputs.jumpOccurrence[1].type
    isKindOf SecurityBreachException
```

Discussion. These examples of more eccentric join points are presented here for illustrative purposes. The only goal is to show how powerful a join point model based on UML action semantics can be. It is not clear at this time whether there are sound use cases for these scenarios. The loop action join point had a valid use case in the context it was proposed [10] – parallelization of iterations in AspectJ – but does not have one in this context, since actions are inherently concurrent.

4 Related Work

4.1 Executable UML² and Aspects

Mellor [3] proposes a framework for aspect-oriented modeling in the context of MDA. The ideas in the paper are a twist of the “Executable UML” approach [12] with an emphasis on aspects. From the standpoint of separation of concerns, the main concepts presented in the paper are:

- *domains* are autonomous and group related abstractions;
- *bridges* establish dependencies between domains;
- a *join point* is an element (of any sort) in a domain model that can be tied together with a another join point provided by a different model, forming a *join*;
- *joins* can connect join points of different types, regardless they being static or dynamic, synchronous or asynchronous. There can be joins between classes,

² “Executable UML” in the context of this section denotes specifically Stephen Mellor's MDA based modeling method

between objects, between an operation and a signal, between a state transition and a signal, etc;

- a *pointcut* is a set of join points and the behavior to be performed on those join points (a mix of the traditional definitions of pointcut and advice).

This paper (and [12, §17]) is an overview of how the author's methodology could make use of aspects. It has some interesting ideas, but it fails to provide a more concrete proposal on combining executable models and aspects. The author prefers to wait and see what the next version of UML (2.0) has to offer in the area of crosscutting concerns, if anything. The more valuable contributions come from the authors' experience in the area of model driven software development and executable (or “translatable”) models.

4.2 UMLAUT

“UMLAUT is a framework for building tools dedicated to the manipulation of models described using the Unified Modeling Language” [13].

[15] does an exploratory work on manipulating UML models using action semantics, due to its ability of manipulating the UML meta-model. The paper shows various examples where model transformations written using UML action semantics are used for refactoring, application of design patterns, and aspect weaving at the model level.

In [14], the UMLAUT framework is presented in the context of implementing weavers for aspect-oriented models. The authors recognize that the Object Constraint Language (OCL) [11, §6] is insufficient for implementing model transformations because it is just a constraint language, not being capable to perform computations that produce side effects. UML action semantics is chosen as the “language” for meta-model manipulation.

There are many points in common between UMLAUT effort and our present work, the main ones being the use of action semantics for manipulating the UML meta-model and the belief in the importance of action semantics for tool interoperability.

5 Summary

The combined use of Model Driven Software Development and Aspect-Oriented Software Development is a trend that is attracting the interest of the software development community. However, so far, no consensus on how this combination should occur exists. Several approaches for representing aspects in UML class diagrams exist, but most ignore the need for models to specify basic and crosscutting behavior. This was the focus of this work. Our approach adopts action semantics for specification of behavior in the base and aspect models. Moreover, we propose a dynamic join point model for UML based on action semantics. Action semantics provide a rich, detailed model for behavior specification. By taking the execution of actions as the basic join point type, we intend to provide a join point model that is at the same time powerful and easy to understand.

References

1. CLARKE, Siobhán, WALKER, Robert. Towards a Standard Design Language for AOSD. In proceedings of the 1st International Conference on Aspect-Oriented Software Development, Enschede, The Netherlands, April, 2002.
2. STEIN, Dominik, HANENBERG, Stefan, UNLAND, Rainer. Designing Aspect-Oriented Crosscutting in UML. In Workshop on Aspect-Oriented Modeling with the UML, AOSD'02. Netherlands, April, 2002.
- STEIN, Dominik, HANENBERG, Stefan, UNLAND, Rainer. On Representing Join Points in the UML. In 2nd Workshop on Aspect-Oriented Modeling with UML, UML 2002. Dresden, Germany, September 30, 2002.
3. MELLOR, Stephen. A Framework for Aspect-Oriented Modeling. In 4th Workshop on Aspect-Oriented Modeling with UML, UML 2003. San Francisco, October, 2003.
5. VÖLTER, Markus. Patterns for Handling Cross-Cutting Concerns in the context of MDSd.
6. MDA Guide Version 1.0.1. MILLER, Joaquin, MUKERJI, Jishnu (ed.). Object Management Group. <http://www.omg.org/mda>, 2003.
8. The AspectJ Programming Guide. The AspectJ Team. 2003. <http://eclipse.org/aspectj>.
10. HARBULOT, Bruno, GURD, John. **A join point for loops in AspectJ.**
11. UML specification v1.5. Object Management Group, March, 2003. <http://www.uml.org>.
12. MELLOR, Stephen, BALCER, Marc. Executable UML.
13. UMLAUT Web site. <http://www.irisa.fr/UMLAUT/>
14. HO, Wai-Ming, JEZEQUEL, Jean-Marc, PENNANEACH, François, et al. A Toolkit for Weaving Aspect Oriented UML Designs. In 1st International Conference on Aspect-Oriented Software Development. Enschede, Netherlands, April, 2002.
15. SUNYÉ, Gerson, PENNANEACH, François, HO, Wai-Ming, et al. Using UML Action Semantics for executable modeling and beyond. In 13th Conference on Advanced Information Systems Engineering. Interlaken, Switzerland, 2001.
16. ZAKARIA, Aida, HOSNY, Hoda, ZEID, Amir. A UML Extension for Modeling Aspect-Oriented Systems. In 2nd Workshop on Aspect-Oriented Modeling with UML, UML 2002. Dresden, Germany, September 30, 2002.
17. BASCH, Mark, SANCHEZ, Arturo. Incorporating Aspects into the UML. In 3rd Workshop on Aspect-Oriented Modeling with UML, AOSD'03. Boston, Massachusetts, March, 2003.
18. KANDÉ, Mohamed, KIENZLE, Jörg, STROHMEIER, Alfred. From AOP to UML – A Bottom-Up Approach. In Workshop on Aspect-Oriented Modeling with the UML, AOSD'02. Netherlands, April, 2002.
19. ALDAWUD, Omar, ELRAD, Tzilla, BADER, Atef. UML Profile for Aspect-Oriented Software Development. In 3rd Workshop on Aspect-Oriented Modeling with UML, AOSD'03. Boston, Massachusetts, March, 2003.
20. GROHER, Iris, SCHULZE, Stefan. Generating Aspect Code from UML Models. In 3rd Workshop on Aspect-Oriented Modeling with UML, AOSD'03. Boston, Massachusetts, March, 2003.
22. BARBOSA, Pablo A., CONTRERAS, Carlos F., RODRIGUEZ, Juan M. MDA and Separation of Aspects: An approach based on multiple views and Subject Oriented Design.