An Abstract Metamodel for Aspect Languages

Wilke Havinga, Tom Staijen, Arend Rensink, Lodewijk Bergmans and Klaas van den Berg
{havingaw,staijen,rensink,bergmans,vdberg}@cs.utwente.nl
Universiteit Twente

ABSTRACT

Aspect-oriented programming (AOP) is a paradigm that can be used to improve separation of concerns. The essence of this paradigm lies in the composition of an existing program — the base program — with a type of module called an aspect, in such a way that the execution of the base program is modified in a controlled way, without modification of the base program itself (on the code level). AOP is widely accepted as an excellent structuring mechanism, but it has also often been pointed out that an encompassing theory of aspect composition is as yet missing. We define an abstract metamodel in which we capture the elements that we believe are the typical aspect oriented programming concepts. Also, these elements are crucial to the understanding of the semantics of AOP language constructs. We intend the metamodel to be a first step towards modeling and comparing AOP languages, as well as a foundation to define the semantics of AOP languages.

1. INTRODUCTION

Aspect-oriented programming (AOP) is a paradigm that can be used to improve separation of concerns. The essence of this paradigm lies in the composition of an existing program — the base program — with a type of module called an aspect, in such a way that the execution of the base program is modified in a controlled way, without modification of the base program itself (on the code level). AOP is widely accepted as an excellent structuring mechanism, but it has also often been pointed out that an encompassing theory of aspect composition is as yet missing. There are several reasons for this, of which we name two that have motivated the current work:

- There are many languages that are understood to support the aspect-oriented paradigm in one way or another; yet the mechanisms by which they do so are often quite diverse;
- The mechanism of composing an aspect with a base program (so-called weaving) is usually defined on an implementation level, and hence its effect on the semantics is not well-understood. This is aggravated by the absence of a clear semantics of the base language and the aspects, i.e., the operands of the weaving mechanism.

The first of these issues may be called the language concern, and the second the semantics concern. In this paper we describe a proposal to address both concerns: namely, we define a metamodel in which we capture, on an abstract level, the elements that we believe are, on the one hand, underlying the typical aspect oriented programming concepts, and, on the other, crucial to the understanding of the semantics. The abstractness in the metamodel lies in the fact that, according to our approach, any given (concrete) programming language should appear as a specialisation of the metamodel.

All in all, we intend the abstract metamodel to be a first step towards the following goals:

- A classification and comparison of the mechanisms available in different aspect-oriented languages.
- A common (albeit abstract) model of AOP language semantics, which can facilitate interoperability of existing tools as well as for a basis for simulation or verification tools applicable to a wide spectrum of languages.

The structure of the paper is as follows: in the next section, we introduce the metamodel and explain the overall structure and underlying concepts. In Section 3 we explain the elements in the metamodel in more detail. Section 4 shows how constructs from different AO programming languages can be explained in terms of the metamodel, showing by example how we believe to have fulfilled the requirements above. We present related work and conclusions in Sections 5 and 6, respectively.

2. APPROACH

The metamodel is positioned on a level that is, on the one hand, abstract enough so that essentially all constructs that can be found in concrete AO languages can be explained by specialising it, and on the other, concrete enough so that the essence of the operational semantics can be captured. As we will see, this level of abstraction is higher than what is required for a language metamodel: many essential language characteristics are not distinguished in our approach.

Figure 1 depicts the elements of our approach. The contribution of this paper lies in the topmost elements: the conceptual metamodel and the metamodel interpreter. In terms of the OMG metamodelling hierarchy [10], these lie on the M2-layer, meaning that actual AO programs (at M1) are regarded as instances of our metamodel. In fact, as the figure shows, in practice we see our metamodel as a core of common concepts that are specialized in the various concrete AO languages, which in turn are instantiated. Again, from the language perspective there is room for a layer in between our conceptual level and the concrete language level, in which more language-oriented features are identified and distinguished. A specialisation of the metamodel would provide a representation of, for example, a language that allows defining aspects, e.g. AspectJ [7, 8] or Composition Filters [1].

Where the metamodel proper captures the static structure of programs (albeit on a very abstract level), which is relatively well-understood, the “interpreter” in the figure effec-
tively constitutes the semantics, which is a concept on which there is much less common understanding. We approach the definition of the semantics by giving a flow diagram (in Section 3) that puts the basic steps in the execution of an AO program in relation to one another, and also to the static metamodel. The core concept of the semantics is the diversion of control flow from the base program to the aspect. We believe that all AOP mechanisms can be ultimately explained in terms of such diversions; the main difficulty lies in explaining when the diversion occurs, and what happens between the diversion and the moment control returns to the original program.

3. THE CONCEPTUAL META-MODEL

3.1 Metamodel of Base Programs

A program without aspects is modeled as a unit, as shown in Figure 2.

In this diagram, a unit consists of a sequence of atomic base actions. These actions perform transformations on the program state. This state should be able to represent an exact moment in the execution of a program. Thus, the state includes structure, runtime values, control flow and the history of the program.

3.2 Aspectual Meta-Model

This section describes the concepts used to describe aspect programs, and explains in what way the execution model of aspect programs is different from normal (e.g. object oriented) programs.

Figure 3: Concepts used to describe the meta-model.

Figure 4: Execution diagram of an advised unit.

Figure 4 shows the extension of the normal program execution. Base program actions are alternated with a test if a Diversion Point has been reached. If this is the case,
a scheduler uses Diversion Bindings that couple DPSs to an Advice and schedules one of more actions to be executed. The scheduler uses optional Scheduling Constraints to choose the next action. When the divertion is complete, the control is returned to the base level on either the same or another point. At this point, the Diversion Point Selectors are evaluated again, because there might be other Advices that need to be executed.

4. EXAMPLES

Mapping programs with aspects to the meta-model is not straightforward, as our model does not define an explicit concept to represent, for example, around-advice. The concept of around-advice highlights a big difference between the AspectJ joinpoint model and the Diversion Points in our meta-model: in AspectJ the “execution of a method” can be used as a joinpoint, even though the method may consist of many base actions and even an aspect feature, proceed(). In contrast, diversion points represent atomic moments or points during the execution of the program - they point to a place between two actions, not to an action itself. For this reason it does not make sense to speak of before and after advice in the context of our metamodel.

4.1 Before Advice

We start with a simple example of a before advice. Suppose we have a class A that calls a method foo() on class B. We want to log all calls to B.foo() using a before-advice. The logging is handled by the method log() in class C. Classes A, B and C are all part of the base system. In the meta-model, this scenario can be modeled as a DPS that matches when the next action is a call to method B.foo(). A binding couples this DPS to the execution of a method that holds the advice. The advice is a call to method C.log(), a base action.

Figure 5 shows the control-flow of the system. At a certain moment during the execution of a program, control is given to (or rather taken by) the Diverter, the part of the aspect-system that matches the current state with the specified DPSs. If DP1 (the call to method B.foo()) is reached, control is taken by a Scheduler, which schedules the actions that are bound to the matching DPSs. Then the first action is executed, the execution of C.log(). When this action is completed, and no more actions are scheduled, control is returned to the base-system. Since no changes have been made to the stack of the base-system, method foo() will then be executed.

4.2 Around advice

In the next example, we create a mapping of an AspectJ around-advice to the concepts of our meta-model. We want to time the duration of the execution of B.foo(). A method C.timing(). This method will store the current time before the execution of B.foo() and calculate the duration afterwards. Again, the Diversion Point Selector will select the moments when the next action will be a call to B.foo(). This point is shown as DP1 in Figure 6. Now, there are two things to consider. Firstly, somewhere during the execution of the advice, we need to insert the execution of B.foo(). However, we do not want an aspectual action like proceed() in the base-level code. Therefore, we will create a Diversion Point Selector to select the moment when proceed() would have been called, otherwise. However, at this selected Diversion Point, the next action to be scheduled should be an action bound to DP1. In the figure this is shown. At DP1, the first action scheduled is a call to BindingManipulator.prepare(DP2). This aspect action will modify any Diversion Bindings that are bound to Diversion Point Selectors that select the proceed-moments in the next around advice. The Bindings are modified suchs that, at DP2, the scheduler will know that is has to select the next action for DP1. If, at this moment, no more around advices are scheduled, the scheduler will schedule B.foo(), the original base action. Secondly, since an around advice will either skip the selected base action or execute it itself, we have to modify the program state such that it will not execute B.foo(). There, the last aspect action bound to a Diversion Point with one or more around advices will manipulate the base-level stack to have it skip the current action.
4.3 Inter-type declarations

Inter-type declarations could be considered an Advice Action at load-time growing the static structure of the program. When existing methods are overridden we could either (1) replace them in the static structure, or (2) add the new methods to the static structure and add another diversion point selector and an associated advice action to replace execution of the original method with the newly added method.

4.4 Aspect data instantiation

In many aspect oriented languages, aspects are represented as an extension of classes: like normal classes, they can have instance variables and methods. As our model focuses on representing the behavior of aspects, we try to define what it means when an aspect defines methods and/or fields. A rough distinction can be made: there exist symmetrical AOP languages, which see aspects as an extension of normal classes, i.e. as classes with the additional ability to superimpose behavior elsewhere. In such an approach, aspects are simply a part of the program, and can be instantiated like normal classes. In an asymmetrical approach however, the base system is completely oblivious to the aspect level, which means it is impossible to directly instantiate aspects from the base code. We assume that in a symmetrical language the issues related to instantiating aspects are handled as part of the base code, i.e. they fall outside the scope of this metamodel. This section instead deals with the sharing of (aspect) data between advices, and the different instantiation mechanisms that are applied in current languages.

When data should be shared between advices, the aspect program has to specify how and when instances of aspect data (that is, instances of the variables encapsulated by the aspect definition) should be created. In many languages the instantiation mechanism is implicit, but several languages give various levels of control over the creation of aspect data - possibly even at the level of individual variables. In many languages, the default is to have a single instance of each aspect for the entire application. However, a lot of other aspect instantiation schemes exist, for example: per thread, per target object, per joinpoint, or even more elaborate schemes such as association aspects [11].

Because many different models exist, we do not try to directly capture all of those mechanisms in our metamodel. Instead, we create a metamodel that leaves room to implement arbitrary concrete aspect instantiation mechanisms.

To explain how aspect data instantiation can be represented within our metamodel, we take two simple examples that use different instantiation mechanisms. As a first example, imagine an aspect that dynamically switches the scheduling strategy for a queuing mechanism. Such an aspect could use an advice to count the number of calls to the methods put and get, which are part of a class Queue. The number of calls has to be stored, and should also be available to the advice that decides about the scheduling mechanism. In this case, a single instance of the aspect variables containing counters for the number of calls to methods get and set is sufficient.

Figure 7 shows a sequence diagram describing the interaction between base- and aspect-level for this example. Elements that are specific to this case (such as specific advices) have been marked in italics - but the scenario is a generic example of advices sharing an aspect instance. In this example, the aspect level monitors the base level, and diverts control to the aspect level when a diversion point (such as a call to the method put within class Queue) occurs. The aspect level diversion mechanism evaluates all diversion point selectors, resulting in a set of the matching selectors. Based on this information, the advice scheduler determines which advice should be executed. The aspect state manager determines the right context for the execution of the selected advice. To make this decision it uses an aspect instantiation specification (which is part of the aspect program) to determine whether it should create a new aspect instance, or retrieve an existing one (from a table that is part of the aspect state). In this case, a new instance of the aspect instance representation class SchedulingData (containing the counter) is created and stored as part of the aspect state. The advice countAdvice is executed with this object as its context. At a later moment, another diversion point (DP2) occurs, and the advice strategyAdvice is selected for execution. The aspect state manager retrieves the existing instance of class SchedulingData, and executes the advice using that object as part of the advice context.

Figure 8 shows the control flow for an aspect that implements a caching mechanism for method results. Presuming the method return values depend on instance variables, such a cache has to be kept per object to which the advice cachingAdvice is applied. The main difference as compared to figure 7 is that here, the aspect state manager reflects on the current diversion point (thisJoinPoint in AspectJ), and stores/retrieves an instance of class PerObjectCacheData in the aspect state for each object (in this example, o1 and o2) that occurs as a target.
In this example, we handle the concern of regulating access to a simple document filing system. The filing system consists of folders (class `Folder`) in which documents can be filed. The concern `AccessRegulation` consists of one filtermodule, `AccessEnforcement` (line 2-11), which enforces access constraints to the filing system. This module defines two inputfilters (line 9-10). Each filter definition has an identifier (allow and deny) followed by the filter type (Dispatch and Error). Each type of filter has its own semantics, defining what should happen (1) in case a messages matches (the filter accepts the message), and (2) in case a message does not match this filter instance (the filter rejects the message).

Without explaining the syntax of the matching part of a filter instance in great detail\(^1\), the dispatch filter on line 9 accepts calls to the method `getContents` as well as calls to the method `addDocument` under the condition `create`, which is true if the current user is allowed to add new documents (the condition is defined on line 6). When a dispatch filters accepts a message, it dispatches the call, in this case to the inner object (i.e. the original target object). When it rejects a message, the dispatch filter does not take any action, which means the message will continue through the filter set. In this example, the filter definitions are separated by the ';' (sequential) operator, specifying that the filter on line 9 takes precedence over the one on line 10.

### 4.5.1 Mapping to the metamodel

Figure 9 shows a mapping of composition filter concepts to the concepts of the metamodel as described in figure 3.

---

**Figure 8: Per-object aspect instantiation**

**Figure 9: Mapping composition filter concepts to the metamodel**

---

\(^1\)For a detailed explanation of the basic concepts of composition filters and superimposition, see [1]
a diversion point selector) determines what ‘advice’ should be applied at this point. Because each filter type can specify specific actions for the accept and reject cases, each filter definition can lead to two diversion point selectors: one when the message is accepted, another when the message is rejected. To describe the diversion point selector of the reject case, the conditions and matching part are simply negated. In practice most filters only influence the program (apply advice, in terms of the metamodel) in either the accept or reject case. For example, the dispatch filter influences the program (by explicitly dispatching a message) only when it accepts a message, while the error filter only influences the program (by throwing an exception) when it rejects a message.

Therefore, a mapping of the example in listing 1 to a concrete representation in terms of the metamodel could be created as below. In this example, DPS1 and DPS2 are instances of the Diversion Point Selector metamodel-class in 9:

DPS1: A message is sent to an object of type Folder; the selector of the called method is getContents or the condition ‘create’ is true and the selector of the called method is addDocument (line 9 in listing 1).

DPS2: A message is sent to an object of type Folder; the selector of the called method can be anything (‘*’) - note that this is the negated version of the matching part in line 10 (listing 1), because we are interested in the case where the error-filter rejects a message.

Note that no diversion point selectors are defined for the cases where the dispatch filter reject or the error filter accepts a message. In those cases, no advice is applied, hence no diversion of the control flow is necessary.

The diversion point selectors DPS1 and DPS2 are bound to advice that should be executed when the dispatch filter accepts, or the error filter rejects, respectively. In figure 9 this relation between accepting/rejecting messages (in terms of filter modules) and the execution of advice (in terms of the metamodel) is shown. A dispatch advice would contain metalevel actions that (potentially) modify the target and selector of the message, instruct the aspect level to continue execution of the base level (i.e. to go ahead and actually send the message), as well as indicate that no other advice can be applied at this same diversion point. Once a message is dispatched to the inner target object, it will not go through other filters at this same diversion point. This notion is important, because in this example, when DPS1 matches, DPS2 will always match as well. Therefore, a scheduling constraint between DPS1 and DPS2 is specified as part of the model, to represent the sequential operator (‘;’) specified in the filtermodule definition. After the advice that is bound to DPS1 is executed, the scheduler has been instructed by the meta level action of the dispatch filter that it should not continue the evaluation of other advice at this diversion point. A graphical representation of the control flow in this example is given in the form of a sequence diagram in figure 10. However, in general filter types do not constrain the execution of other advice at the same diversion point. This way, it is possible to compose several filter definitions that modify the message or execute additional actions.

Figure 10: Example control flow for the dispatch filter, represented in terms of the metamodel

4.6 Other

One can image that regular pointcut/advice constructs can be mapped quite straightforward to the proposed execution model. In this section we will explain how less straightforward language concepts of language models could be mapped to the model. We emphasize could since there are more possible solutions to these mappings.

4.6.1 Stateful aspects

In [3] a model is proposed where events in the base program cause transitions of an aspect’s LTS. Any of these transitions might also cause an advice to be executed. Since, in our model, the program state contains history information we are able to create a Diversion Point Selector for every state in the aspect’s LTS. The advice bound to the transitions in the LTS can be mapped to actions bound to the Selectors.

4.6.2 Aspects on aspects

Aspects on aspects means that advice can also be bound to Diversion Points in the execution of other aspects. Since these aspects can cause base actions, our model is suitable for aspects on aspects. If we would also want to be able to bind advice to an aspect action, we would have to change the Execute Base Action label in figure 4 to Execute Action. Diversion Point Selectors should then also be able to select aspect action, or more generically, to contain predicates of the the aspect-level state.

5. RELATED WORK

Masuhara and Kiczales present a closely related work in [9]; in this paper they present a sandbox for modeling aspect-oriented languages. The paper illustrates how to use this sandbox for four significantly distinct Aspect-Oriented Programming Languages (AOPLs). A key property of their proposal is that they model two input languages (e.g. a ’base language’ and an ’aspect language’) and a target language (i.e. the domain where the join points occur). The semantics of an AOPL can be expressed in this model by describing how the two input languages together either lead to an execution, or to a new, woven, program. Hence their model aims at providing a common framework for expressing the semantics of an AOPL through a transformation approach. This is a different focus then the model we present in this paper, where we try to model the essential common
To model and compare AOP languages. Hence, the elements in our model more closely resemble the concepts in AOP languages. Nonetheless, both proposals may be suitable as a foundation for expressing semantics of AOPLs or for constructing an aspect language sandbox.

The paper by Filman and Friedman [5] introduces two main concepts as the underlying principles of AOP: quantification and obliviousness. Quantification is expressed in our model by the fact that diversion point selectors may apply to many (distinct) diversion points in the execution of the program. Obliviousness relates to the direction of the dependencies between the base program and the aspect specifications: namely the latter refer (implicitly) to the previous, but there are no explicit dependencies from the base program to the aspect specifications.

In [4], a formal definition is given for ‘crosscuts’ (c.f. pointcut designators, or in the terminology of this paper, diversion point selectors). This relates to one of the concepts in our meta-model and should be described as an alternative way of specifying diversion point selectors.

Chavez and Lucena propose a metamodel on AOP in [2], as a UML metamodel. The focus is on modeling aspects w.r.t. base programs, and to describe aspects as having Crosscutting relationships, where the latter are modeled as associations. In our opinion, this work is describing the concepts at an abstract level, where it is impossible to reason about the detailed meaning of language concepts.

6. CONCLUSION

In this paper, we have presented a meta-model for aspect-oriented programming languages. The meta-model is described as a set of related concepts, along with sequence diagrams to illustrate the dynamic behavior of most important dynamic elements of the meta-model. The meta-model has been designed to be very generic, allowing many different AOP languages to be fit in the framework. To demonstrate that this genericity is indeed achieved, we have shown in section 4 how a wide range of common AOP techniques, such as before and around advice, inter-type declarations, aspect instantiation, and stateful aspects can be modeled within this framework. We have also shown how a more elaborate set of related AOP techniques in the composition filters model fits in our framework.

The process that eventually results in the dynamic execution of an aspect has also been designed to be very flexible as to allow for multiple policies (such as different advice scheduling/ordering mechanisms) and control flows. The (top-level) control flow is described in detail through the use of sequence diagrams. These provide an initial basis for a more detailed and precise specification of the semantics. Several options are available to do so; in the context of the AOSD-Europe Network of Excellence, a similar framework is being developed, where the semantics are specified through the definition of an interpreter. This interpreter can execute a specific instantiation of the model, i.e. a program. We also intend to explore the possibilities of defining the semantics of the metamodel elements by means of graph transformations, along the lines of the semantics presented in [6]. This is currently work in progress.

We foresee the following applications of the meta-model presented in this paper:

- To model and compare AOP languages (by investigating the commonalities and differences in instantiating the metamodel elements).
- As a foundation to define the semantics for AOP languages.
- As the foundation for exchanging AOP programs in a language-independent manner (cf. UML meta-models).

7. ACKNOWLEDGMENTS

The research presented in this paper was funded by the AOSD-Europe Network of Excellence. This paper has benefited significantly from the input we received by several partners in the project, most notably Johan Briand, Jacques Noyé, and Mira Mezini.

References


