Prototyping and Composing Aspect Languages

using an Aspect Interpreter Framework

Wilke Havinga Lodewijk Bergmans Mehmet Aksit

Software Engineering group – University of Twente
P.O. Box 217, 7500 AE Enschede, The Netherlands
{w.havinga,l.m.j.bergmans,m.aksit}@ewi.utwente.nl

Abstract

Domain specific aspect languages (DSALs) are becoming more popular because they can be designed to represent recurring concerns in a way that is optimized for a specific domain. However, the design and implementation of even a limited domain-specific aspect language can be a tedious job. To address this, we propose a framework that offers a fast way to prototype implementations of domain specific aspect languages. A particular goal of the framework is to be general enough to support a wide range of aspect language concepts, such that existing language concepts can be easily used, and new language concepts can be quickly created.

We show mappings of several domain specific aspect languages to demonstrate the framework. Since in our approach the DSALs are mapped to a common model, the framework provides an integrating platform allowing us to compose programs that use aspects written in multiple DSALs. The framework also provides explicit mechanisms to specify composition of advices written in multiple DSALs.

1. Introduction

The benefits of using domain specific aspect languages (DSALs) are widely recognized [5, 11, 18]. In fact, the idea of expressing each crosscutting concern using a dedicated domain-specific language was at the very heart of the first proposals called “AOP” [8].

However, designing and implementing DSALs can be a tedious job. For example, each aspect language has to define under which circumstances an aspect should influence the program, and implement mechanisms to facilitate this (e.g. using bytecode weaving).

In addition, most applications will need to express concerns from different problem domains, making it desirable to write programs using multiple DSALs. That way, each DSAL could be used to effectively address the concerns within its specific domain.

It is not trivial to compose aspects expressed in several DSALs however, as each language typically constructs its own model of the program; unless a lot of care is taken, the effects of one aspect may not be reflected in the models constructed by other DSALs. In addition, aspects written in several DSALs may interact with each other, possibly in undesirable ways (depending on the situation).

Our paper contributes the following to address these problems:

(1) We propose an aspect interpreter framework that can be used to prototype domain specific aspect languages. As our framework supports a wide range of aspect language concepts, it can be used to prototype diverse DSALs in a reasonable amount of time, as we will show in section 3.

(2) Using our approach, aspects written in several (domain-specific) languages are mapped to a common model. As a result, we can compose applications that are written using multiple DSALs, as we will show in section 4.1.

(3) The framework provides explicit mechanisms to specify composition of advices, even if advices are written in several DSALs. This is discussed in section 4.3.

In this paper, we show implementations of only two DSALs. However, our work is based on a thorough study of aspect oriented languages [14], as well as the modeling of their possible implementation mechanisms using an interpreter, as presented in [4].

In the next section, we briefly introduce the framework itself, and discuss some of our design and implementation considerations. Section 3 presents more details about the framework by showing the implementations of several DSALs using our framework. Section 4 discusses the composition of aspects written in multiple DSALs, including specifications to resolve the interactions between aspects. We discuss related work in section 5, and conclude the paper in section 6.

2. JAMI - an aspect interpreter framework

One of the defining features of AOP languages is the support for ”implicit invocation” of application behavior. That is, behavior can be invoked without an explicit reference (such as a *method call* statement) being visible in the (source) code at the point of invocation. Implicit invocation is a key feature of the interface between the base program and the aspect program. The framework to model aspect language mechanisms we present in this paper is strongly based on implicit invocation as the connection between the base program and the aspects (advices).

In this section, we briefly discuss the concepts used in the aspect language domain, based on a reference model proposed in [14]. We then propose a framework that provides common implementations of these concepts, while supporting variations on these concepts found in different aspect languages. The general design and architecture of the framework was first proposed in [4] and [7]. Finally, we briefly outline the workflow used to prototype DSALs using this framework.

2.1 Common aspect language concepts

Aspect languages must first of all support the concept of *pointcuts*. Pointcuts define the circumstances under which an aspect influences a program – for example, at certain locations (such as entering a particular method) or under particular runtime conditions (e.g., only when a variable x equals 5). Pointcuts can be seen as predicates or conditions over the *execution state* of a program. The execution state may, in a broad sense, include information about the call stack, objects, or even the execution trace and structure of the program. As pointcuts may match at several places or moments during the execution of a program, the concept of *joinpoints* is used to model references to the relevant execution state (e.g., which method is being intercepted) whenever a pointcut matches. Pointcuts can be bound to *advices*, which may add to or replace parts of the original program behavior and/or its runtime state. Advices can use the joinpoint information to adapt their behavior based on the current runtime situation. Finally, *bindings* specify how pointcuts and advices are connected and grouped into modules (usually called aspects). In addition, bindings are also used to bind "aspect state" – data stored by aspects, such that it can be shared between advices (i.e., similar to sharing state between methods by using instance variables).

2.2 Framework implementation

Aspect languages adopt varying implementations of the concepts listed above. We provide a framework that implements the behavior of these high-level concepts, and allows for their refinement to facilitate specific language implementations.

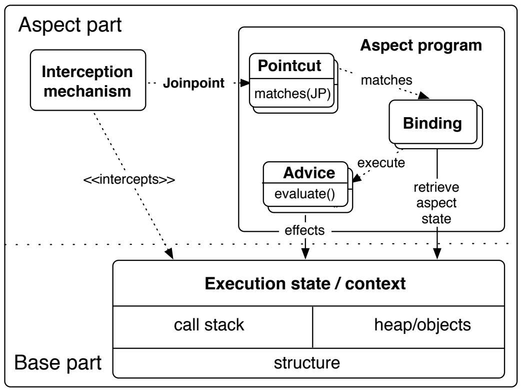


Figure 1. The Java Aspect Metamodel Interpreter - an overview

Figure 1 shows a global overview of our framework, called the Java Aspect Metamodel Interpreter (JAMI) [1]. Basically, this framework enforces the high-level structure and control flow of aspects, while providing implementations of common concepts at an abstraction level that is appropriate when prototyping DSALs – as we intend to demonstrate in section 3. By enforcing a fixed high-level control flow, our framework provides a common platform that enables composition of aspects written in multiple DSALs, as we will show in section 4. To provide the flexibility required to model features of particular languages, each concept can be either instantiated in a dedicated configuration of framework elements, or refined (extended) when necessary. JAMI provides many of the common implementations found in different aspect languages.

2.2.1 Control flow

We briefly discuss the high-level control flow within JAMI. In principle, the base program (a normal Java application) runs as it would without the interpreter. However, the *interception mechanism* (see figure 1) intercepts the control flow at any point that is of potential interest to the aspect interpreter. Our current implementation uses a regular AspectJ aspect to intercept *all* method calls and field assignments[1]. Apart from intercepting method calls, the mechanism keeps track of *context* information that may be of interest to the framework. Currently, it keeps track of the call stack, senders, targets, and method signatures of all calls on the stack, as well as field assignments. Upon interception of the control flow, the mechanism creates a *joinpoint* object representing the current joinpoint. A refinement class exists for each different joinpoint type, such as *MethodCallJoinpoint*, *MethodReturnJoinPoint* or *AssignmentJoinpoint*. Each of these joinpoint objects keeps a reference to the relevant context information - e.g., the method that was executing upon interception, etc.

Subsequently, each *pointcut* registered with the aspect interpreter is evaluated against the current joinpoint (see figure 1). As indicated before, pointcuts are basically conditions that either match or do not match a particular joinpoint. Thus, the main *pointcut* class consists of only an *evaluate* method, which returns true or false based on whether it matches the current joinpoint. Refinement classes are provided for many common pointcut conditions; new ones can be created if necessary to implement DSAL-specific pointcut types. For example, we provide pointcuts that match based on the type of joinpoint, method signature, or target object type. In addition, there are pointcut classes that can combine other pointcuts using regular logic expressions (*and*, *or*, *not*). Many concrete examples of implemented pointcut conditions will be shown in section 3.

One or more pointcuts can be associated to one or more advices using *bindings* (see figure 1). For each matching pointcut, the interpreter looks up the corresponding *advice* through these bindings. Advice can be expressed in terms of elementary advice "building blocks" provided by JAMI, which allows the expression of many common types of advice without creating custom implementations for each advice. In addition, advice can be expressed using normal base code, when necessary. Advices may also want to share state among each other, or among different executions of the same advice. Therefore, we also provide *bindings* to aspect state; this will be discussed in more detail in section 3.

When several pointcuts match at the same joinpoint, the order of advice execution has to be resolved. This issue is discussed in detail in section 4.

3. Features of JAMI, demonstrated by example

In this section, we show 2 aspect languages optimized for a specific task, implemented using the Java Aspect Metamodel Interpreter. We first introduce a running example that we will use to demonstrate each language.

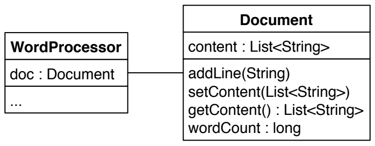


Figure 2. Example application, used throughout the paper

Figure 2 shows the UML class-diagram of a simple word processor application. Within this application, class *Document* defines some methods to modify a document (*addLine()* and *setContent()*), a method to obtain the document content (*getContent*), as well as a method that counts the current number of words in the document (*wordCount*).

In the following subsections, we extend this example using aspects written in several domain-specific aspect languages.

3.1 Using the D/COOL domain-specific aspect language for synchronization

To show that JAMI can be used to conveniently accommodate complex, existing domain-specific languages, we implement a relevant subset of the coordination aspect language "COOL", which is part of the D language framework. The language is documented extensively in the dissertation describing this framework [11].

Suppose we want to add a spellchecker to our word processor, which runs concurrently with the user interface by using a separate thread. To ensure correct behavior when multiple threads may

[1] The use of an AspectJ aspect as an interception mechanism limits the *joinpoint granularity* to what is supported by AspectJ. However, the interception mechanism consists of only 100 lines of code, and could easily be replaced if support for finer joinpoint granularity is desired.

access a document concurrently, we use a synchronization specification written in COOL, as shown in listing 1. By using COOL, we do not have to put any synchronization-related code in the Java source code itself.

```
1  coordinator Document {
2    selfex addLine, setContent;
3    mutex {addLine, setContent};
4
5    mutex {addLine, getContent};
6    mutex {addLine, wordCount};
7    mutex {setContent, getContent};
8    mutex {setContent, wordCount};
9  }
```

Listing 1. Using COOL to synchronize reader/writer access

Listing 1 specifies that we want to coordinate instances of class *Document*. Line 2 specifies that the methods *addLine* and *setContent* are self-exclusive; i.e. only 1 thread at a time may be running those methods. Line 3 specifies that these methods are mutually exclusive in addition; i.e. only one thread may be active in either *addLine* or *setContent* at a given time.

Lines 5-8 also specify pairs of methods not allowed to run at the same time - *addLine* and *setContent* are writer methods, and should not run at the same time as reader methods *getContent* or *wordCount*.

By default, COOL synchronizes method access *per object*, i.e. in the above example, several threads can still run method *addLine* at the same time, as long as they do so within different object contexts. In addition, COOL allows to specify a *per class* modifier, which makes the synchronization "global" for the specified class.

3.1.1 Mapping to JAMI

We now describe a mapping of the COOL specification above to JAMI. First, for each method involved in a synchronization (i.e. selfex/mutex) specification, we calculate the set of methods that may not be entered while another thread is active within that method. For method *addLine*, this "exclusion set" contains *addLine* itself (because of the selfex specification on line 2), as well as methods *setContent*, *getContent* and *wordCount* (because of the mutex specifications on line 3, 5 and 6). How these exclusion sets are determined exactly is documented in [11]; we do not repeat the details here.

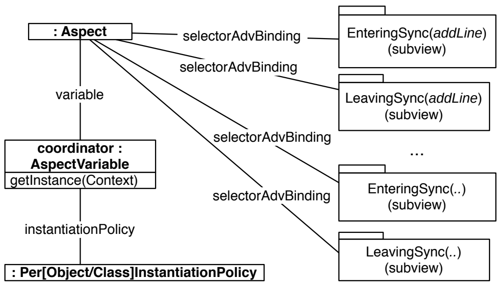

: Aspect
selectorAdvBinding
selectorAdvBinding
selectorAdvBinding
selectorAdvBinding
EnteringSync(addLine) (subview)
LeavingSync(addLine) (subview)
...
EnteringSync(..) (subview)
LeavingSync(..) (subview)
variable
coordinator : AspectVariable
getInstance(Context)
instantiationPolicy
: Per[Object/Class]InstantiationPolicy

Figure 3. Expressing COOL coordinators using JAMI concepts

The coordinator specification is modeled (see figure 3) as an aspect that defines one *AspectVariable* named *coordinator*. In JAMI, aspects consist of bindings between selectors and advices, in addition to definitions of aspect state (aspect variables), which may be shared between advices. In JAMI, each aspect variable has its own "instantiation policy". For example, a "singleton" policy means that there is one instance of the variable for the entire program, a "per object" policy means there is one instance of the aspect variable for each target object (where the current target object depends on the join point context), etc. Instantiation is usually implicit (although it is also possible to specify explicit instantiation): new instances are automatically created when needed (using the default constructor of the specified variable type), i.e. on first use in a particular context.

In this example, the aspect variable *coordinator* has a "per object" or "per class" instantiation policy, depending on the specified granularity of the coordinator. Thus, the variable is shared between advices belonging to this coordinator, and can be used to regulate the synchronization. In addition, two selector-advice-bindings are defined for each method involved in a synchronization specification; one will be executed upon entering the method, one upon leaving.

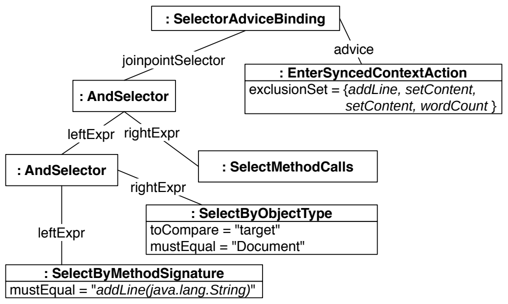


Figure 4. Entering a synchronization context: pointcut and advice

Figure 4 shows the object diagram for the selector-advice-binding executed upon entering method *addLine*. It matches only join points of type *MethodCall*, of which the target object is of type *Document*, and of which the signature of the called method is *addLine*. Before the call is executed, we execute the advice *EnterSyncedContextAction*, an advice class specific to this language.

We show the source code of this advice in listing 2. First, the advice retrieves (line 2-4) the *coordinator* aspect variable instance belonging to this specific context (i.e. object or class, depending on the instantiation policy). This *coordinator* object ensures that the synchronization "bookkeeping" itself is properly synchronized. While the advice holds a lock on this object (line 6-21), it can safely inspect the *MethodState* objects for this coordinator. For each method (involved in synchronization), such a *MethodState* object tracks which threads are currently running that method. While other threads are active in any method in the exclusion set of the currently invoked method (line 11,12), the advice waits (releasing the lock on the *coordinator* while waiting) until this is no longer the case (line 14-16). When the loop is left, it means the method is free to run - after the advice registers the current thread with the corresponding *MethodState* object (line 20) and releases the lock on the coordinator object.

```
1  public boolean evaluate(InterpreterContext metaContext) {
2    CoordinatorImplementation coord =
3      (CoordinatorImplementation) metaContext.getAspect().
4      getDataFieldValue(metaContext, "coordinator");
5
6    synchronized(coord) {
7      boolean shouldWait;
8      do {
9        shouldWait = false;
10       // Wait while any other thread is active in any method in our exclusionset
11       for (String excludedMethod : exclusionSet)
12         shouldWait |= coord.getMethodState(excludedMethod).isActiveInOtherThread()
             ;
13
14       if (shouldWait) {
15         try { coord.wait(); }
16         catch(InterruptedException e) { }
17       }
18     } while (shouldWait);
19     // This method is now allowed to run, register it
20     coord.getMethodState(myMethodName).enteringMethod();
21   }
22   return true;
23 }
```

Listing 2. Advice executed when entering a synchronized method

To conclude the implementation, another pointcut is created to intercept join points that occur upon *leaving* any of the methods involved in the synchronization specification. The object diagram is analogous to figure 4, except the pointcut now matches only join points of type *MethodReturn*, and executes an advice of type *LeaveSyncedContextAction*. We show the source of this advice in listing 3. The advice waits until it obtains a lock on the coordinator object within the given context (object or class, as in the previous advice), allowing it to update the synchronization "bookkeeping". Once the lock is obtained, it deregisters the current thread from the *MethodState* object for this method (line 6). It then notifies all waiting threads (if any), such that they can re-evaluate their waiting conditions (line 8).

```
1   public boolean evaluate(InterpreterContext metaContext) {
2     CoordinatorImplementation coord = ...; // as in previous listing
3
4     synchronized(coord) {
5       // deregister this thread from running this method
6       coord.getMethodState(myMethodName).leavingMethod();
7       // Notify all threads (not just one), as potentially several may be allowed
            to continue
8       coord.notifyAll();
9     }
10    return true;
11  }
```

Listing 3. Advice executed when leaving a synchronized method

3.2 An experimental DSAL to implement caching

As another example, we implement an experimental language that introduces a modular way to specify caching of method return values (also called *memoization*).

Methods (or functions) to which memoization is applied, traditionally have to conform to the following conditions: (1) the method depends on its (input) parameters only; (2) given the same input parameter values, it should return the same result every time; (3) the method should have no side effects. Our implementation maintains the last two requirements. However, the first requirement is often violated in object-oriented programming, as results of a method call are often influenced by instance variables (within the same object) or specific method calls (on the same object). Therefore, our implementation extends the notion of memoization as defined above, by allowing cached results to be invalidated when the value of particular fields change, or when particular methods are called.

In our example application from figure 2, the method *wordCount* is a good candidate for memoization. The method has no side effects, but depends on the value of instance variable *content*. This variable is written by method *setContent*, as it contains the statement "this.content = newContent;". The method *addLine*, containing the statement "content.add(line);" does not overwrite the instance variable itself; it does however modify its contained object structure. Therefore, calls to method *addLine* should also invalidate the return value of *wordCount*.

We specify the above using a domain specific aspect language as shown in listing 4.

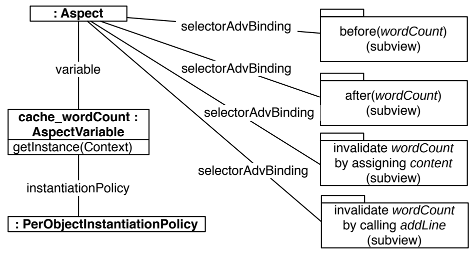
```
1   cache Document object {
2     memoize wordCount,
3     invalidated by assigning content
4             or calling addLine(java.lang.String);
5   }
```

Listing 4. Example specification of a memoization aspect

This specification means the following: apply a caching aspect on each *Document* object (line 1). This caching aspect will memoize the return value of method *wordCount* (line 2). The cache will be invalidated when a new value is assigned to instance variable *content* within the corresponding *Document* object (line 3), or when the method *addline(..)* is called on the *Document* object (line 4).

3.2.1 Mapping to JAMI

We now show how to map the specification shown in listing 4 to JAMI. As figure 5 shows, we create an aspect variable of type *Cache* for each *memoize* declaration. Its instantiation policy can again be specified as per object or per class - in the example above, we want to cache the return value of method *wordCount* for each object of type *Document*. The class *Cache* models a simple wrapper object that can store and retrieve an object, as well as clear its currently stored value.



: Aspect
selectorAdvBinding
selectorAdvBinding
selectorAdvBinding
selectorAdvBinding
variable
before(wordCount) (subview)
after(wordCount) (subview)
cache_wordCount : AspectVariable
getInstance(Context)
invalidate wordCount by assigning content (subview)
instantiationPolicy
invalidate wordCount by calling addLine (subview)
: PerObjectInstantiationPolicy

Figure 5. Mapping a caching aspect to JAMI concepts

For each memoized method, we need a pointcut that intercepts calls to that method, coupled to an advice that returns the cached value (if one is stored). Another pointcut intercepts *returns* from the memoized method, coupled to an advice that stores the return value in the cache. Finally, a pointcut is needed for each cache validation specification, coupled with an advice that invalidates the cache. In this example there are two such pointcuts, corresponding to the invalidation specifications in line 3 and 4 of listing 4).

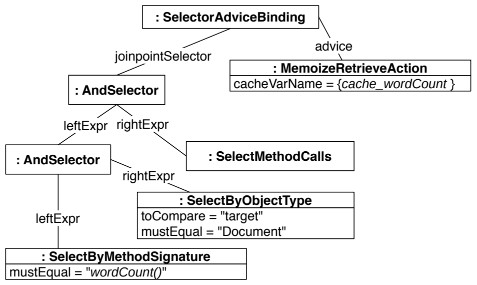


Figure 6. Selector-advice binding for retrieving cached values

As shown in figure 6, we intercept calls to the method of which the results should be cached. The advice that is executed is shown in listing 5. First, the advice retrieves the aspect variable corresponding to this *memoize* declaration (line 2+3). If the cache currently contains a value (which means it must have been set after a previous call), we instruct the interpreter not to execute the original call after it finishes executing this advice (line 7), and instead to set the return value to the value found in the cache (line 8).

```
1   public boolean evaluate(InterpreterContext metaContext) {
2     Cache cache = (Cache)metaContext.getAspect()
3       .getDataFieldValue(metaContext, cacheVarName);
4
5     if (cache.hasValue())
6     { // Use cached value!
7       metaContext.setExecuteOriginalCall(false);
8       metaContext.setReturnValue(cache.getValue());
9     }
10    return true;
11  }
```

Listing 5. Advice: retrieving a cached value

After the method returns, the advice in listing 6 is called, which stores the return value of the method. First, the advice retrieves the cache variable (line 3+4). Next, it stores the return value of the called method, which can be obtained through the interpreter context (line 6). Note that we do not take method parameters into account in this implementation (fortunately, the method *wordCount()* does not have any). This is done to avoid cluttering the example; adding this behavior would be straightforward.

```
1   public boolean evaluate(InterpreterContext metaContext)
2   {
  Cache cache = (Cache)metaContext.getAspect()
    .getDataFieldValue(metaContext, cacheVarName);

  cache.setValue(metaContext.getReturnValue());
  return true;
}
```

Listing 6. Advice: storing a cached value

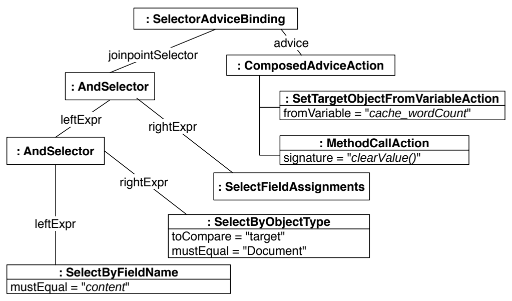


Figure 7. Selector-advice binding for invalidating cached values

To finalize our example, we show one of the pointcut-advice-bindings used to invalidate the cache. Figure 7 shows a pointcut that will match field assignments, but only to the field named *content*, and when the assignment takes place within an object of type *Document*. The advice is to call the method *clearValue* on the aspect variable *cache_wordCount*.

4. Composition of multiple DSALs

As each DSAL is designed to address concerns within a particular problem domain, we would often want to combine the use of several such languages within a single application[2]. Implementing this is not straightforward however, as partial programs expressed in several languages have to be composed into a single combined, working application. Even if this is technically feasible (which is not necessarily the case), running the combined application may reveal unexpected and/or undesired results.

In this section, we discuss how several aspects written in different DSALs (all implemented using JAMI) can be composed and used within the same application. We discuss several difficulties that may occur in this case, and explain how JAMI can help to address these issues.

4.1 DSAL composition in JAMI

In general, the composition of multiple aspect languages is far from trivial. As an example, consider the common implementation of aspect languages as transformation of the source code or byte code representation of the base program (where each of these aspect language implementations may, or may not, share a common infrastructure). This would require the sequential execution of aspect language implementations over the incrementally transformed base code. Typically, such a byte code transformation is not commutative, meaning that the behavior of the resulting program could vary, according to the –normally undefined– execution order of the aspect language implementations. A similar story holds for the sequential execution of multiple aspect interpreters at each join point.

In section 3, we have shown how aspects written in several DSALs are mapped to JAMI elements. Such aspects, expressed in terms of JAMI elements, or refinements of JAMI elements, can be deployed within a single application–even though they originate from different aspect languages. This is enabled by the common runtime platform provided by JAMI.

This platform defines common abstractions and a common data structure for the representation of aspects (e.g. in terms of pointcut expressions, advice-selector bindings, ordering constraints, etc.). Further, the framework imposes a unified high-level control flow for the execution of aspects, as shown schematically in figure 1. At the same time, while adopting these predefined abstractions and high-level control flow, for each language there is a large freedom to define in varying ways how e.g. pointcuts can be defined and matched.

Thus, using JAMI, it is possible to execute aspects written in different DSALs within a single application. This does not require any tailoring or design decisions that are specific to the other DSALs that are combined. However, this does not guarantee that the resulting application will show the "correct" or "desired" behavior. As is the case with aspects written in a single language, interactions or interference may also occur between aspects written in different DSALs.

This phenomenon has also been observed before: e.g. in [12], two categories of aspect interactions are distinguished:

- *co-advising*: the composition of advice of multiple aspect languages at a shared join point.
- *foreign advising*: this corresponds to the notion of "aspects on aspects", where advice from one aspect language may apply to a join point associated with the execution of advice in another aspect language.

In the remainder of section 4, we first discuss the issue of co-advising, followed by an explanation of the advice composition mechanism of JAMI in section 4.3. Although JAMI also addresses foreign advising, we do not discuss this due to lack of space.

4.2 Co-advising

When multiple pointcuts match at the same join point, the order in which advices bound to these pointcuts are executed may lead to different behavior [15, 6], if there are dependencies between the aspects. Reversely, in the absence of any ordering specification at shared join points, the application behavior may be non-predictable and undesirable.

The above is also true if the shared join points originate from programs written in different aspect languages. For individual languages, many mechanisms exist to deal with this.

However, when pointcuts originate from different languages, there are two additional issues:

- We need improved or additional mechanisms to compose advices from different aspect languages. The reason is that we (want to) assume DSALs to be developed independently, so that aspects written in a particular DSAL are likely (and preferably) unaware of those written in another DSAL. JAMI supports a uniform constraint model (first proposed in [15]) that facilitates ordering constraints *within* as well as *between* languages. We demonstrate this below.
- There is a distinction between *language-level* and *program-level* composition [12]. In particular for DSALs, composition constraints may be specific to a combination of DSALs, and should apply to all aspects written in those DSALs (i.e. language-level constraints). However, it may –in addition– be possible that some constraints are program-specific (i.e. program level).

[2] Note that the entire discussion about the composition of DSALs technically also holds for the composition of general purpose aspect languages, or a mixture of these. However, we believe composition of DSALs is much more realistic to expect, hence we focus on this.

4.2.1 Example: composing the synchronization and caching aspects

When we deploy the aspects for synchronization (shown in listing 1) and caching (listing 4) within our original application (see figure 2), we observe that several shared join points occur, as most calls to methods within class *Document* are advised by both aspects. Therefore, we need to determine in what order these advices should be executed.

As an example, we consider the join point that occurs when returning from method *wordCount*. At this join point, a caching advice

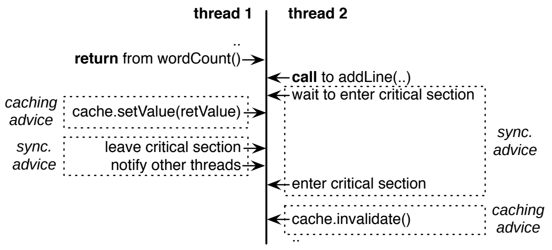


Figure 8. Using the correct advice ordering

will store the value that was returned by the method. The synchronization advice leaves the critical section that was entered before the method was executed, as shown in listing 3. In this case, the caching advice –at the end of a method– should be executed before the synchronization advice. This is illustrated in figure 8, whereas figure 9 illustrates a specific scenario of two threads where –in both cases– the synchronization advice precedes the caching advice. In the latter case, a different thread executing a *writer* method may invalidate the cache as soon as the critical section is left, while subsequently the caching aspect stores an (already invalidated!) value in the cache. In that case, the next call to *wordCount* would return a cached value that is incorrect. To generalize the example,

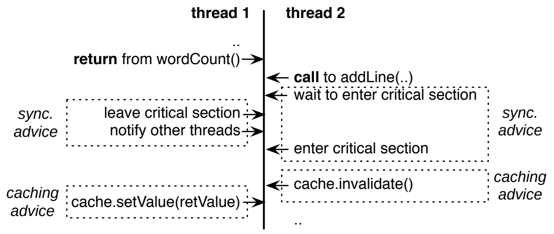


Figure 9. Concurrent execution with incorrect advice ordering

we observe that any caching advice should occur within the critical sections as imposed by the synchronization advice. Specifically, for advices executed at a shared *MethodCalljoinpoint*, the synchronization advice should have precedence, while at a shared *MethodReturnJoinpoint*, the caching advice should have precedence. This is an example of a language-level composition constraint.

4.3 The advice composition mechanism of JAMI

JAMI offers two complementary advice composition mechanisms. First, it implements a generic ordering constraint mechanism as proposed in [14]. At shared join points, constraints may limit which advices are currently applicable. Such constraints may be conditional, and may for example depend on which advices where already executed (at the same join point). Even so, the application of constraints may still leave several advices eligible for execution. Second, JAMI therefore supports a "scheduling" interface to determine the further selection of advice execution. Different strategies can be implemented to disambiguate the selection of advice. Our default implementation picks an arbitrary element from the set of applicable bindings, and in addition prints a warning that the program is potentially ambiguous. In addition, the scheduler can decide to cancel further advice executions at a given join point, if requested to do so by particular advice actions[3].

Constraints are decoupled from the "aspect modules", and are instead kept as a separate set of entities within the aspect evaluation framework. This enables the specification of constraints between selector-advice-bindings that are part of several aspects, or that even originate from several aspect languages.

As discussed above, for our example we want to specify language-level composition based on the originating language of each selector-advice-binding. We do not need any program-level constraints, in this case. Therefore, we simply create constraints between all selector-advice-bindings, such that caching advices occur within the critical section created by synchronization aspects (if the advices apply at the same join point), and decorator advices get even higher priority.

A functional implementation (in JAMI) that composes aspects written in the two DSALs discussed in this paper – including the constraints as discussed in this section, is downloadable as part of the example discussed throughout this paper [1].

5. Related work

In [13], Masuhara and Kiczales propose the Aspect Sand Box, an interpreter framework to model aspect mechanisms. Using this framework, the effects of aspects are defined in terms of weaving semantics. The weaving process is modeled by extending or modifying the interpreter of a base language that models a single-inheritance OO language (which can be seen as a core subset of Java). In comparison, JAMI defines a common runtime environment for aspects, which allows us to express explicit ordering constraints between advices, and enables the deployment of multiple aspect languages within a single application. As discussed in section 4, it would be harder to define a single weaver that models the composition of multiple languages.

The AspectBench Compiler (*abc*) [2] is a workbench that – like JAMI – facilitates experimentation with new (aspect) language features. Unlike JAMI however, it focuses mainly on extensions to AspectJ, and strives to provide an industrial-strength compiler architecture that facilitates efficient implementations of extensions to the AspectJ language. In contrast, while designing JAMI we specifically tried to avoid design decisions that would limit the flexibility of our framework. In addition, *abc* is not designed to handle composition between multiple languages.

The work from Kojarski and Lorenz [9, 10] is strongly related to ours; in particular, they also investigate the issue around the composition of multiple aspect languages. In [10], seven interaction patterns among features of composed aspect languages are described. Some of these, such as *emergent advice ordering*, are also discussed in this paper. However, because (1) JAMI introduces its own set of *abstract features*, such as selector-advice bindings, and (2) in our interpreter-based approach, individual aspect languages are not translated into base language terminology, hence, there is never accidental interaction, not all interaction patterns are applicable. However, the proposed analysis approach could also be applied in the context of our work.

In [9], the AWESOME framework is described; instead of an interpreter-based approach, this adopts a weaver-based approach,

[3] This corresponds to the run-time detection and resolution of aspect interactions in [17].

that also addresses foreign advising, and language-level, but currently –according to [9]– not program level co-advising (which we presented in section 4.3).

The *Reflex* AOP kernel [18, 17] is also closely related work; it is a reflection-based kernel for AOP languages, with a specific focus on the composition of aspect programs. To this extent, it provides an (extensible) set of composition operators, which can be used when translating an aspect specification to a representation in terms of the kernel-level abstractions. Although there are many similarities with JAMI, a key difference of the current implementation is that it is weaving-based, rather than interpreter-based. Mostly due to this, the support for foreign advising is limited (as e.g. exemplified in [10]).

The XAspects project [16] implements a system to map DSALs to AspectJ source code. The approach addresses the need to compose aspects written in multiple DSAL, but does not provide explicit mechanisms to deal with interactions between aspects, other than suggesting the use of the AspectJ *declare precedence* construct. Compared to this, JAMI offers more elaborate ways to specify the composition of aspects.

In [3], Brichau et.al. propose the definition and composition of DSALs ("Aspect-Specific Languages") using Logic Metaprogramming. Although their approach is not based on a typical OO framework, it does allow the reuse and refinement of aspect languages. It is based (in [3]) on static source code weaving (by method-level wrapping). The composition of aspect languages (program level composition is not supported) is achieved by explicit composition of languages into new, combined languages. In our opinion, this is less flexible, as it requires explicit composition for each configuration of aspect DSALs that occur in an application, and the late addition of a new aspect language in a system may not be possible without restructuring the composition hierarchy.

6. Conclusion

In this paper, we have shown implementations of two domain-specific aspect languages, using our aspect interpreter framework. Using this framework, it took only 3-4 days (per language) to create functional prototypes of these diverse DSALs. Aspects written in these DSALs can be composed with regular Java programs at runtime, in an interpreted style.

We have used JAMI in a programming language course to teach the common aspect language concepts and various implementations thereof. As part of this course, students successfully developed small DSALs within limited allotted time. This supports our claim that JAMI can be used to prototype DSALs while requiring relatively little effort, even including the learning curve of the framework itself.

We contribute the effectiveness of JAMI as a framework for prototyping DSALs in large part to its flexibility and expressiveness. For example, as aspects are completely dynamically evaluated, it is easy to experiment with pointcuts that express complex selection criteria over the runtime state. In addition, our support for "aspect state" using variables that each may have different instantiation policies provides a flexible way to implement aspect language features, while requiring relatively little effort.

We have shown that our framework supports applications composed of aspects written in several DSALs. In addition, we have discussed interactions that may occur when combining multiple DSALs, and demonstrated mechanisms implemented as part of JAMI to specify aspect composition – also of aspects written in different languages. The framework as well as the examples shown in this paper can be downloaded from the JAMI website [1].

7. Acknowledgments

This work is supported by European Commission grant IST-2-004349: European Network of Excellence on Aspect-Oriented Software Development (AOSD-Europe).

References

[1] Java Aspect Metamodel Interpreter - http://jami.sf.net/, 2007.

[2] P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. *abc*: An extensible aspectj compiler. *Transactions on Aspect-Oriented Software Development I*, 3880/2006:293 – 334, February 2006.

[3] J. Brichau, K. Mens, and K. De Volder. Building composable aspect-specific languages with logic metaprogramming. In *1st Conf. Generative Programming and Component Engineering*, volume 2487 of *lncs*, pages 110–127, Berlin, 2002. Springer-Verlag.

[4] J. Brichau, M. Mezini, J. Noyé, W. Havinga, L. Bergmans, V. Gasiunas, C. Bockisch, J. Fabry, and T. D'Hondt. An Initial Metamodel for Aspect-Oriented Programming Languages. Technical Report AOSD-Europe Deliverable D39, Vrije Universiteit Brussel, 27 February 2006 2006.

[5] M. D'Hondt and T. D'Hondt. Is domain knowledge an aspect? In C. V. Lopes, A. Black, L. Kendall, and L. Bergmans, editors, *Int'l Workshop on Aspect-Oriented Programming (ECOOP 1999)*, June 1999.

[6] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In K. Lieberherr, editor, *Proc. 3rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2004)*, pages 141–150. ACM Press, Mar. 2004.

[7] W. K. Havinga, T. Staijen, A. Rensink, L. M. J. Bergmans, and K. G. van den Berg. An abstract metamodel for aspect languages. Technical Report TR-CTIT-06-22, Enschede, May 2006.

[8] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *11th Europeen Conf. Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[9] S. Kojarski and D. H. Lorenz. Awesome: an aspect co-weaving system for composing multiple aspect-oriented extensions. In R. P. Gabriel, D. F. Bacon, C. V. Lopes, and G. L. S. Jr., editors, *OOPSLA*, pages 515–534. ACM, 2007.

[10] S. Kojarski and D. H. Lorenz. Identifying feature interactions in multi-language aspect-oriented frameworks. In *Proceedings of the 29th International Conference on Software Engineering*, pages 147–157, Minneapolis, MN, May 20-26 2007. ICSE 2007, IEEE Computer Society.

[11] C. V. Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, College of Computer Science, Northeastern University, 1997.

[12] D. H. Lorenz and S. Kojarski. Understanding aspect interactions, co-advising and foreign advising. In *ECOOP 2007 Second International Workshop on Aspects, Dependencies and Interactions*, 2007.

[13] H. Masuhara and G. Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In L. Cardelli, editor, *ECOOP 2003—Object-Oriented Programming, 17th European Conference*, volume 2743 of *lncs*, pages 2–28, Berlin, July 2003. Springer-Verlag.

[14] I. Nagy. *On the Design of Aspect-Oriented Composition Models for Software Evolution*. PhD thesis, University of Twente, June 2006.

[15] I. Nagy, L. Bergmans, and M. Aksit. Composing aspects at shared join points. In A. P. Robert Hirschfeld, Ryszard Kowalczyk and M. Weske, editors, *Proceedings of International Conference NetObjectDays, NODe2005*, volume P-69 of *Lecture Notes in Informatics*, Erfurt, Germany, Sep 2005. Gesellschaft für Informatik (GI).

[16] M. Shonle, K. Lieberherr, and A. Shah. XAspects: an extensible system for domain-specific aspect languages. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 28–37, New York, NY, USA, 2003. ACM Press.

[17] É. Tanter. Aspects of composition in the Reflex AOP kernel. In *Proceedings of the 5th International Symposium on Software Composition (SC 2006)*, volume 4089 of *lncs*, pages 98–113, Vienna, Austria, Mar. 2006. Springer-Verlag.

[18] É. Tanter and J. Noyé. A versatile kernel for multi-language AOP. In R. Glück and M. R. Lowry, editors, *Generative Programming and Component Engineering, 4th International Conference (GPCE)*, volume 3676 of *Lecture Notes in Computer Science*, pages 173–188. Springer, Sept. 2005.