AWESOME: An Aspect Co-Weaving System for Composing Multiple Aspect-Oriented Extensions *

Sergei Kojarski
College of Computer and Information Science
Northeastern University
360 Huntington Ave.,
Boston, Massachusetts 02115, USA
kojarski@ccs.neu.edu

David H. Lorenz
Dept. of Mathematics and Computer Science
The Open University of Israel
108 Ravutski St., P.O. Box 808,
Raanana 43107, Israel
lorenz@openu.ac.il

Abstract

Domain specific aspect-oriented language extensions offer unique capabilities to deal with a variety of crosscutting concerns. Ideally, one should be able to use several of these extensions together in a single program. Unfortunately, each extension generally implements its own specialized weaver and the different weavers are incompatible. Even if the weavers were compatible, combining them is a difficult problem to solve in general, because each extension defines its own language with new semantics. In this paper we present a practical composition framework, named AWESOME, for constructing a multi-extension weaver by plugging together independently developed aspect mechanisms. The framework has a component-based and aspect-oriented architecture that facilitates the development and integration of aspect weavers. To be scalable, the framework provides a default resolution of feature interactions in the composition. To be general, the framework provides means for customizing the composition behavior. Furthermore, to be practically useful, there is no framework-associated overhead on the runtime performance of compiled aspect programs. To illustrate the AWESOME framework concretely, we demonstrate the construction of a weaver for a multi-extension AOP language that combines COOL and AspectJ. However, the composition method is not exclusive to COOL and AspectJ—it can be applied to combine any comparable reactive aspect mechanisms.

* This work was supported in part by NSF's *Science of Design* program under grants numbered CCF-0438971 and CCF-0609612.

OOPSLA'07, October 21–25, 2007, Montréal, Québec, Canada.

Categories and Subject Descriptors D.1.5 [*Programming Techniques*]: Aspect-oriented Programming; D.2.12 [*Software Engineering*]: Interoperability; D.3.4 [*Programming Languages*]: Processors

General Terms Design, Languages

Keywords AOP, aspect extension, aspect mechanism, aspect weaver, composition, DSL, framework, pluggability.

1. Introduction

Aspect-oriented programming (AOP) [21] is celebrating a decade of research and development and industry adoption. New language features are continually being proposed. These features need not only be implemented and evaluated, but also tested to work with existing AOP languages. Facilitating the construction of new aspect extensions that incorporate new features and that can be composed with other extensions is thus important for making research advances accessible for experimentation and use in realistic settings.

Supporting the composition and use of newly developed aspect extensions together with established main stream extensions can leverage and broaden their respective impact. The ability to program in a multi-extension AOP language can also help compare features. It can eliminate tradeoffs associated with choosing an extension with the most appropriate features. Most importantly, it can help realize the vision of domain specific aspect languages (aspect DSLs).

Unfortunately, despite the availability of extensible aspect weavers (e.g., [4, 5]) and extension composition frameworks (e.g., [37, 36, 22]), implementing industry-quality weavers that are composable remains a complex and costly task. For example, `abc` [4, 5] is more extensible than `ajc` [17], but does not support composition with foreign extensions. Reflex [37] and XAspects [36] support composition, but ignore foreign advising [28]: they lack the customizability necessary for preventing aspects from "misadvising" foreign aspects. Pluggable AOP [22] resolves the flawed foreign advising behavior found in Reflex and XAspects. However,

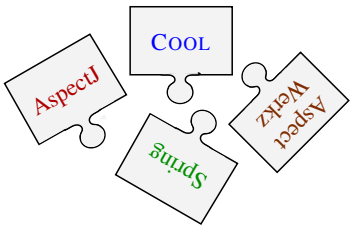

Figure 1. Pluggable weavers

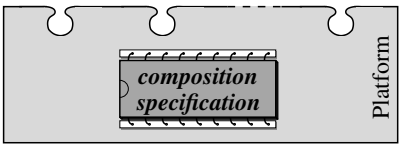

Figure 2. Composition framework

Pluggable AOP is impractical for combining "real world" AOP languages. It supports the customization of individual extensions, but provides only limited customization of the composition semantics. Also, Pluggable AOP composes extensions by constructing an interpreter, which is inefficient, and thus deemed inappropriate for industrial use.

Today, potentially useful aspect extensions are not readily available because either it is too difficult to implement them or they are implemented but cannot be used together with AspectJ [20]. The unattended need for combining COOL [27] and AspectJ is a representative example. Software engineering studies [32, 33, 40, 41] that compared COOL and AspectJ have concluded that COOL code is easier to understand and debug than Java or AspectJ code. Yet, COOL is not widely used; AspectJ programmers cannot embed COOL code in their AspectJ programs.

1.1 The Composition Problem

There are two main impediments to overcome. The first is the ***composition specification*** problem [24]: *given a set of n extensions, identify and resolve feature interactions in their composition.* For example, COOL extends Java with a method synchronization mechanism; AspectJ extends Java with an advice binding mechanism. In a composition of COOL and AspectJ, coordinators and aspects may interact in unexpected ways.[1] These interactions need to be identified and resolved.

The specification problem is inherently complex and difficult, and its resolution is outside the scope of this paper. It is complex because for a choice of n extensions there are $O(n^2)$ pairwise interactions to specify.[2] It is difficult because several reasonable resolutions exist for each interaction [24].

The second impediment, which is the main focus of this paper, is the ***composition implementation*** problem: *design a composition framework such that, given n aspect weavers (Figure 1) and a composition specification (Figure 2), plugging them into the framework implements the composition under the specification.*

The implementation challenge is to design a framework that has the following characteristics:

Minimum performance overhead. To be practically useful, the framework should construct weavers without inflicting performance degradation in the runtime of compiled aspect programs.

Maximum code reuse. The framework should provide libraries and abstractions that support rapid development of individual weavers. Components that are common to all AspectJ-compatible weavers should be reused whenever possible.[3] New weavers should implement only the necessary extension-specific operations. Avoiding unnecessary code repetitions in constructing different weavers also improves their reliability.

Auto-configuration. The framework should support automatic composition of multiple weavers into a multi-weaver that exhibits a reasonable default behavior, thus avoiding whenever possible the tedious task of resolving all the interactions explicitly.

Manual override. Although the default multi-weaver behavior is likely to be appropriate for most cases, a composition specification might require to resolve some of the feature interactions differently. The framework must allow the language designer to override parts of the default configuration in order to comply with the specification.

1.2 Contribution

The main contribution of this paper is an Aspect co-WEaving System for cOmposing Multiple Extensions (AWESOME). The AWESOME framework is:

- **Composable:** enables third-party composition of aspect weavers into a multi-weaver with a reasonable default behavior.
- **Customizable:** provides means for customizing the behavior of the constructed multi-weaver to cater for the composition specification; and
- **Efficient:** employs a compile-time weaving scheme.

In comparison to other frameworks (Table 1), AWESOME is the only one to provide a flexible customization mechanism. The compile-time weaving scheme used by AWE-

[1] A *coordinator* in COOL is the equivalent of an *aspect* in AspectJ.

[2] In the worst-case, there might be $O(2^n)$ combinations to consider.

[3] By *AspectJ-compatible* we refer to a reactive join point and advice aspect extension [23] that can be reduced to AspectJ, e.g., COOL, AspectWerkz [6], CaesarJ [3].

Property / Framework	**Reflex**	**XAspects**	**Pluggable AOP**	**AWESOME**
Composition approach	Reflection	Preprocessing	Interpretation	Compilation
Composability	-	-	+	+
Customizability	-[4]	-	-	+
Efficiency	+[5]	+	-	+
Real-world languages	-	+	-	+
Specification	-	-	+	+
Evaluation	-	-	+	+

Table 1. Comparison of aspect extension composition frameworks.

SOME is more practical and efficient than the reflection-based and dynamic weaving schema used by Reflex and Pluggable AOP, respectively. The quality of the code woven in AWESOME is comparable to that produced by standard aspect compilers.

AWESOME is also the only composition framework to be demonstrated and tested on real-world languages. We demonstrate the construction of an AWESOME weaver for a multi-extension AOP language, named COOLAJ, that combines COOL and AspectJ. Although we are not the first to pursue a combination of COOL and AspectJ, we are the first to do so systematically by: (a) giving a specification of COOLAJ; (b) constructing a weaver for COOLAJ; and (c) evaluating the weaver by testing its behavior against the COOLAJ specification. We also compare the woven code to code produced by `ajc` [17] and other weaving algorithms.

Another contribution is the analysis of the extension composition problem; a set of requirements for an aspect extension composition framework; and a reasonable default resolution of the feature interaction problem in a composition of aspect extensions. To concretely illustrate the composition problem and its solution, we include specific implementation details for the composition of COOL and AspectJ into COOLAJ. While a specification and a weaver for COOLAJ is a novel and useful contribution in and of itself, the composition approach is not specific to COOL and AspectJ. It generalizes to a large category of (reactive) aspect mechanisms [23].

Outline By way of background, we first explain how individual weavers work. In Section 2 we review the working of the `ajc` weaver for AspectJ, and in Section 3 we describe the working of a weaver for COOL. The objective of this overview is to provide the reader with a familiarity with the internal components of a weaver and to introduce the necessary terminology. In Section 4 we analyze the foreign- and co-advising interactions, and we formulate the requirements for the co-weaving system. A novel aspect-oriented architecture for the framework is presented in Section 5. In Section 6 we refactor the implementation of the AspectJ and the COOL weavers to reflect this new AOP design. In Section 7, as a case study, we informally specify and describe the implementation of a multi-weaver for COOLAJ. In Section 8 we evaluate our AWESOME system and demonstrate its pluggability, correctness, and efficiency.

2. An Aspect Compiler

An *aspect compiler* compiles aspect programs into an executable. We begin by describing the high-level architecture of an aspect compiler. For concreteness, we describe the `ajc` compiler for AspectJ.

In general, an aspect compiler has a front-end and a back-end. The *front-end* translates aspects to (annotated) classes in the base language. For example, the front-end of `ajc` translates aspects written in AspectJ (`.java` and `.aj` files) to annotated classes in Java.[6] The translation process in `ajc` is mostly straightforward: an aspect is translated to a Java class with the same name; an advice declaration is transformed into a method declaration with the same body. The compiled advice method is also annotated with attributes that store its aspect-specific data (e.g., pointcut declarations). The annotations distinguish aspect classes from other Java classes, and provide pointcut designators for advice methods.

The *back-end* implements the semantics of the aspect extension. The semantics define the meaning of advice weaving in terms of computations. A *computation* in this context is a block of program execution, e.g., a method execution. It encapsulates a sequence of operations that define a behavior and a dynamic context that includes all arguments and other values accessible by the computation. An advice is a *computation transformer* [17, 22, 30, 31, 42]. It takes a computation and produces a transformed computation that runs the advice body before, after, or instead of the original computation.

While the extension's semantics define weaving in terms of dynamic runtime abstractions, the weaver implements the semantics statically by transforming the base and aspect classes. The weaver transforms a computation by transforming its *shadow*, a body of code that defines a computation's

[4] Reflex provides support for resolving interactions between aspects, but not between aspect extensions.

[5] The reflection-based weaving scheme of Reflex may degrade the runtime performance of the woven program.

[6] More precisely, the target classes are expressed in the Java Virtual Machine (JVM) bytecode language.

Listing 1. A weaver

```
public void weaveClass(ClassFile cf) {
  Shadow[] shadows = reify(cf);
  for(Shadow shadow:shadows) {
    Advice[] advs = order(shadow, match(shadow));
    mix(shadow, advs);
  }
}
```

Listing 2. The AspectJ weaver

```
public void AJWeaver(ClassFile cf) {
  applyIntroductions(cf);
  weaveClass(cf);
}
```

behavior. For example, the `ajc` weaver transforms Java bytecode. At the bytecode level, an advisable shadow maps to a continuous block of instructions with a well-defined begin and end.

The weaver implements an abstract *weaving process* that comprises four subprocesses, namely **`reify`**, **`match`**, **`order`**, and **`mix`** (Listing 1). These (sub)processes are found in all *reactive aspect mechanisms* [23]—mechanisms that can be semantically modeled as a closed-loop feedback control system—including COOL and AspectJ.

The **`reify`** process takes as input a class file and constructs a weaver-specific representation of the class. For example, the AspectJ weaver represents a class as a set of computation shadows. Its **`reify`** process examines the input Java class cf, and identifies all the shadows that can possibly be advised. Each shadow references a list of instructions embedded in one of cf's methods (the body of the shadow), and provides static and lexical descriptions of these instructions (the static context of the shadow).

The **`match`** process associates elements of the program representation (shadows) with pieces of advice. In AspectJ, the weaver selects the set of advice by matching the description of the shadow (the static context) against the static part of the advice pointcuts.

The **`order`** process sorts and orders all pieces of advice that match the same shadow into a correct application order. The `ajc` weaver orders the pieces of advice according to the rules defined by the AspectJ language semantics.

The **`mix`** process transforms an actual body of a shadow by introducing code of advice (or calls to advice) that match this shadow. The AspectJ weaver transforms the shadow's instruction list by sequentially introducing calls to the advice methods before, after, or instead of the original code. The advice pieces are woven in by sequentially transforming the body of the shadow. An advice then injects new code inside the body of the shadow, immediately before, immediately after, or instead of the original code. The transformation considers instructions that were woven earlier as if they were a part of the original shadow. This way the advice pieces "wrap" around each other in the transformed shadow.

The four processes provide a high-level description of the *advice weaving* semantics. A concrete weaver may also realize other kinds of transformations. For example, the `ajc` weaver implements intertype declarations and advice weaving in two separate steps (Listing 2). First, the weaver extends and transforms the class cf by applying the intertype declarations (the call to applyIntroductions in Listing 2). Once the declarations are applied, the weaver calls weaveClass, which implements the advice weaving behavior. The additional transformations are normally static in nature, and do not interfere with the dynamic advice weaving behavior.

3. A Compiler for COOL

The architecture of a compiler for COOL is similar to that of `ajc`. The front-end translates coordinators in COOL to classes in Java. The back-end instruments the program with calls to methods of coordinator classes. We explain by example the basics of the COOL language and the internal workings of its compiler. Consider the implementation of a bounded stack in Java (Listing 3). `Stack` defines two public methods: `push` and `pop`. An attempt to pop objects off an empty stack or push objects onto a full stack throws an exception.[7]

COOL relieves the implementor of `Stack` from dealing with multi-threading. A separate `Stack` [8] coordinator (Listing 4) imposes the synchronization logic over `push` and `pop` in an aspect-oriented manner. The `Stack` methods are not synchronized. But in the presence of the `Stack` coordinator, the stack object operates correctly even when multiple client threads execute methods simultaneously.

The synchronization policy is expressed in COOL using declarations (**`mutex`**, **`selfex`**, **`condition`**), expressions (**`requires`**), and statements (**`on_exit`**, **`on_entry`**). The **`selfex`** declaration (line 402) specifies that neither `push` nor `pop` may be executed by more than one thread at a time. The **`mutex`** declaration (line 403) prohibits `push` and `pop` from being executed concurrently. The **`requires`** expressions (lines 406 and 412) further guard `push` and `pop` executions. If the guard is false, a thread suspends, even if the **`mutex`** and **`selfex`** conditions are satisfied. The execution resumes when the guard becomes true. `full` and `empty` are **`condition`** boolean variables (line 405).

The **`on_entry`** and **`on_exit`** blocks update the aspect state immediately before and immediately after the execution of an advised method body, respectively. They are used in this example to track the number of elements in the stack

[7] `java.lang.ArrayIndexOutOfBoundsException`

[8] In COOL, the names of the coordinator and the coordinated class must be the same.

Listing 3. A non-synchronized stack

```
public class Stack {
 public Stack(int capacity) {
  buf = new Object[capacity];
 }
 public void push(Object obj){
  buf[ind] = obj;
  ind++;
 }
 public Object pop() {
  Object top = buf[ind-1];
  buf[--ind] = null;
  return top;
 }
 private Object[] buf;
 private int ind = 0;
}
```

Listing 4. A coordinator in COOL

```
coordinator Stack {
 selfex {push, pop};
 mutex {push, pop};
 int len=0;
 condition full=false,empty=true;
 push: requires !full;
  on_exit {
   empty=false;
   len++;
   if(len==buf.length) full=true;
  }
 pop: requires !empty;
  on_entry {len--;}
  on_exit {
   full=false;
   if(len==0) empty=true;
  }
}
```

Listing 5. A translated COOL coordinator class

```
public class StackCoord {
 public synchronized void
 lock_push(Stack target) {
  while (!(!full) ||
   isRunByOthers(pushState) ||
    isRunByOthers(popState))
   try { wait(); }
   catch (InterruptedException e) {}
   pushState.add(Thread.currentThread());
 }
 public synchronized void
 unlock_push(Stack target) {
  pushState.remove(
     Thread.currentThread());
  empty = false;
  len++;
  if (len == target._buf().length)
   full = true;
  notifyAll();
 }
 public synchronized void
 lock_pop(Stack target) {/*omitted*/}
 public synchronized void
 unlock_pop(Stack target) {/*omitted*/}
 private synchronized boolean
 isRunByOthers(List methState) {
  return (methState.size() > 0 &&
   !methState.
    contains(Thread.currentThread()));
 }
 private boolean
  empty = true,
  full= false;
 private List
  pushState = new Vector(),
  popState = new Vector();
 private int len = 0;
}
```

(lines 409 and 413) and to keep the **condition** variables `full` and `empty` current.

Java expressions within COOL statements have read and write access to the coordinator's fields. In addition, expressions may inspect instance variables of the coordinatee, e.g., access the `buf` field of the `Stack` object (line 410). Note that the coordinator's expressions may access not only public but also package-protected and even private fields of the coordinatee object.

3.1 Front-end Translation

The COOL front-end translates a coordinator in COOL to a coordinator class in Java. The name of the class is obtained by appending "`Coord`" to the name of the aspect, e.g., `StackCoord`.

`StackCoord` (Listing 5) implements the synchronization logic via special synchronized methods and instance variables. The class provides a pair of `lock_` and `unlock_` methods and an instance variable for every method that is advised by the coordinator. Specifically, the synchronization for the `Stack.push` method is realized by `lock_push` and `unlock_push`. Similarly, the synchronization logic for `Stack.pop` is realized by `lock_pop` and `unlock_pop`. At any point of the execution, the `pushState` (`popState`) instance variable stores all threads that are currently executing the `push` (`pop`) method on the coordinated object. The coordinator class also includes all fields of its coordinator.

The `lock_` methods implement the semantics for **mutex**, **selfex**, and **requires**, and run **on_entry** blocks. A **while** loop suspends the execution of the current thread if a guard condition is violated. Specifically, the **while** loop

Listing 6. A synchronized bounded stack

```
601 public class Stack {
602   public Stack(int capacity) {
603     buf = new Object[capacity];
604     _coord = new StackCoord();
605   }
606   public void push(Object obj) {
607     _coord.lock_push(this);
608     try{
609       buf[ind] = obj;
610       ind++;
611     } finally {_coord.unlock_push(this);}
612   }
613   public Object pop() {/*omitted*/}
614   public Object[] _buf() {return buf;}
615   private Object[] buf;
616   private int ind = 0;
617   private StackCoord _coord;
618 }
```

in the `lock_push` method suspends the execution of the current thread (by invoking `wait` on the coordinator object) so long as either one of the **`requires`**, **`selfex`**, or **`mutex`** conditions is in violation. The `requires` condition is checked by the `!(!full)` expression (line 504). **`selfex`** fails if `push` is run by another thread (line 505); and **`mutex`** fails if `pop` is run in parallel (line 506). If all the guard conditions are satisfied, the thread executes all the existing **`on_entry`** statements, and locks the coordinated `push` method by adding its `Thread` object to the `pushState` list (line 509).

The `unlock_` methods unlock the coordinated method and run the **`on_exit`** statements. Specifically, `unlock_push` unlocks the coordinated `push` method by removing the current `Thread` object from the `pushState` list (line 513). It then executes the **`on_exit`** statement (lines 515–518) and notifies the other threads waiting on the lock that the coordinated method is free (line 519). Note that accesses to the coordinated object fields (instance variables) are translated into method calls on the coordinated object. Specifically, access to the `buf` field of the coordinated object is translated into a `_buf()` method call (line 517). This is the way in which the coordinator class gains access to protected or private fields of the coordinated class. The method is generated in the coordinated class by the COOL weaver, and simply returns the value of the corresponding field.

3.2 Back-end Weaving

The COOL weaver applies four kinds of transformations to a coordinated class, namely method transformation, constructor transformation, field introduction, and method introduction. When applied to the non-synchronized `Stack` (Listing 3), these transformations yield a synchronized stack (Listing 6). The weaver associates a coordinator with a coordinatee by introducing a `_coord` field in the coordinated

Listing 7. The COOL weaver

```
701 public void COOLWeaver(ClassFile cf) {
702   ClassFile coordAspect = findAspect(cf);
703   if (coordAspect!=null) {
704     addCoordField(cf, coordAspect);
705     transformConstructor(cf, coordAspect);
706     addGetterMethods(cf, coordAspect);
707     weaveClass(cf);
708   }
709 }
710
711 Method[] reify(ClassFile cf) {cf.getMethods();}
712
713 Method[] match(Method shadow) {
714   ClassFile coordAspect = findAspect(
715       shadow.getClass());
716   Method lock = findLock(
717       shadow.getSignature(), coordAspect);
718   Method unlock = findUnlock(
719       shadow.getSignature(), coordAspect);
720   if (lock==null) return new Method[0];
721   return new Method[]{lock, unlock};
722 }
723
724 Method[] order(Method shadow, Method[] advs) {
725   return advs;
726 }
727
728 void mix(Method shadow, Method[] advs) {
729   if (advs.length>0) {
730     addCallBefore(shadow, advs[0].getSignature());
731     addCallAfter(shadow, advs[1].getSignature());
732   }
733 }
```

class (line 617), and adding an initialization statement in the constructor (line 604). The weaver also introduces public getter methods (`_buf()`) for protected and private fields of the coordinated class that need to be accessed by the coordinator.

The weaver transforms the coordinated methods by introducing calls to the coordinator's `lock_` and `unlock_` methods before and after the original body. To ensure invocation of the `unlock_` method, the weaver also introduces a **`try-finally`** block around the original body.

In sum, the COOL weaver realizes the COOLWeaver algorithm (Listing 7). Given a class file cf to be transformed, the weaver searches for its coordinator (findAspect, line 702).[9] If found, the weaver introduces a coordinator field (addCoordField, line 704), transforms the constructors to initialize that field (transformConstructor, line 705), and generates getter methods for protected and private cf fields that are read by the coordinator (addGetterMethods, line 706).

[9] In COOL, each class can be associated with at most one coordinator.

Then, the weaver synchronizes the methods of cf by imposing locking and unlocking advice before and after their bodies, respectively. Advice weaving in COOL follows the same four-process model as in AspectJ (call to weaveClass, line 707). In terms of this four-process model, a shadow in COOL is a method of cf, and the advice are the `lock_` and `unlock_` methods of the coordinator class.

The **`reify`** process of the COOL weaver represents an input class file as a set of methods (the **`reify`** method, line 711). The **`match`** process uses the signature of a yet-to-be-coordinated method to select a pair of `lock_` and `unlock_` advice methods (the **`match`** method, lines 713–722). For every coordinated method, the weaver finds the corresponding `lock_` and `unlock_` methods in the coordinator class (findLock, line 716; findUnlock, line 718). The **`order`** process of the weaver is empty (the **`order`** method, lines 724–726). Lastly, the **`mix`** process (**`mix`**, lines 728–733) introduces a call to the `lock_` method before the method body (addCallBefore, line 730), and a call to `unlock_` after the method body (addCallAfter, line 731).

Listing 8. A logger aspect in AspectJ

```
public aspect Logger {
  pointcut scope(): !cflow(within(Logger));

  before(): scope() {
    System.out.println("before " +
        thisJoinPoint);}

  Object around(): scope() {
    System.out.println("around" +
        thisJoinPoint);
    return proceed();
  }

  after(): scope() {
    System.out.println("after" +
        thisJoinPoint);
  }
}
```

4. Analysis and Design

Now that we have reviewed the working of a weaver, we move on to the main focus of this paper: the problem of composing aspect weavers. Our goal is to build a weaver composition framework for implementing a multi-extension AOP language. Given a set of aspect weavers and a composition specification, the framework should construct an appropriate multi-weaver. In the previous sections we discussed the four-process model of an abstract weaver. In this section we examine the composition specification; and then we derive the design requirements for the composition implementation.

The specification needs to resolve the feature interactions between the composed extensions. There are two main kinds of interactions that the specification should address, namely *foreign advising* and *co-advising* [28]. We reason about the specification by analyzing these interactions, using the composition of COOL and AspectJ as a running example.

4.1 Foreign Advising

A multi-extension AOP language is an AOP language that combines multiple aspect extensions. We call a program in this language a *multi-extension program*. In a multi-extension program, an aspect can generally interact with foreign aspects by advising join points in their execution. The foreign advising interaction determines how aspects in one extension advise foreign aspects in other extensions. Particularly, in a composition of COOL and AspectJ the foreign advising interaction controls the weaving of AspectJ advice into foreign COOL coordinators, and the weaving of COOL advice into foreign AspectJ aspects. For example, consider running a `Logger` aspect in AspectJ (Listing 8) together with our `Stack` coordinator in COOL (Listing 4). `Logger` logs all join points in a program execution, including join points within executions of the `Stack` coordinator. A resolution of the foreign advising interaction must determine what join points `Logger` advises within the `Stack` coordinator, and how. But neither the AspectJ nor the COOL specification define how AspectJ aspects advise COOL coordinators.

Foreign advising is not solvable by merely using a weaver for COOL (AspectJ) to weave the foreign aspects (coordinators), because the one language does not recognize the syntax or semantics of the other. Even though the weavers for COOL and AspectJ may both use Java classes as their intermediate representation, applying the COOL (AspectJ) weaver to the Java representation of foreign aspects (coordinators) will not do the job either. This is because the classes embed synthetic code that is generated during the translation to the intermediate representations, e.g., calls to `wait` and `notifyAll` in the coordinator class `StackCoord` (Listing 5). This synthetic code is specific to a particular implementation of the foreign compiler. The code cannot be traced back to the original source aspect, and exposing it to a foreign weaver may result in the latter advising it, thereby causing unexpected behavior in the program.

In terms of the abstract weaving process, foreign advising is a problem of representing foreign aspects correctly, and is the responsibility of the **`reify`** process. For example, the incorrect behavior observed in translation-based composition frameworks (e.g., XAspects and Reflex) is a result of the **`reify`** process of the framework's weaver failing to provide a correct representation of the foreign aspect classes. Consequently, the weaver erroneously includes shadows also for implementation-specific operations that are introduced by the front-end translator into the intermediate aspect classes

4.2 Co-advising

In a multi-extension program, a join point can generally be advised by several aspects that are written in different extensions. We refer to this behavior as *co-advising*. The co-advising interaction controls the collaborative application of multi-extension advice at the same join point, which is undefined at the level of the individual extensions.

In a composition of COOL and AspectJ, the co-advising interaction coordinates the weaving of COOL and AspectJ advice into the same program element. For example, consider again running the `Logger` aspect in AspectJ (Listing 8) together with the `Stack` coordinator in COOL (Listing 4) and the `Stack` class in Java (Listing 3). The `Logger` and the `Stack` coordinator collaboratively advise executions of the `Stack` methods. A resolution of the co-advising interaction of the composition must determine in what order the pieces of advice of the aspect and the coordinator execute.

In terms of the abstract weaving process, co-advising is a problem of coordinating the **`match`**, **`order`**, and **`mix`** processes of the composed weavers. This problem cannot be resolved just by a sequential application of the individual weavers. If weavers were scheduled to run one after the other sequencely, then (at the same join point) advice that is applied later would always "wraps" around advice that is applied earlier. This would results in a very restrictive behavior that does not support the flexible ordering needed in general for co-advising. Moreover, a weaver may erroneously advise advice binding operations (e.g., calls to advice or coordinator methods) that were introduced into the shadow by previously applied weavers.

4.3 Resolving Feature Interactions

Our analysis clarifies why the feature interaction problem is so complex and difficult. First, in a composition of multiple extensions, the foreign advising interaction generally occurs between every pair of composed extensions. Furthermore, because the interaction involves features that are unique to the interacting parties, the behavior is specific to the composed extensions. For example, a foreign advising interaction between COOL and AspectJ involves terms, expressions, and constructs that are unique to COOL, e.g., **`requires`**, **`on_entry`**, and **`on_exit`**.

Moreover, there is no single correct way to resolve these interactions. For example, a foreign advising interaction between COOL coordinators and AspectJ aspects may allow the aspects to *only* advise access to fields of coordinated objects (that are made from within the coordinators), e.g., access to the field `buf` of the `Stack` class from within the `Stack` coordinator (Listing 4, line 410). Alternatively, the interaction can be resolved by letting the aspects advise *all* field access operations within the **`requires`**, **`on_entry`**, and **`on_exit`** expressions of a coordinator. It can also be resolved to allow the aspects to advise all field access *plus* (*un*)*lock* COOL computations (e.g., as *advice-execution* join points). Each of the three options can be advocated [24], and many more reasonable options exist.

4.4 System Design Requirements

Next, we use the terminology and abstractions from the analysis to formulate three design requirements for the composition implementation. The requirements are: *decoupling*, *composability*, and *customizability*.

4.4.1 Decoupling

The composition framework should decouple abstractions that are common to all AspectJ-compatible weavers from abstractions that are weaver-specific. This reduces the responsibility of the individual weaver to implementing only the extension-specific weaving operations. By reusing the framework's abstractions as much as possible, the development of new weavers is drastically simplified.

4.4.2 Composability

The framework should support the composition of multiple aspect weavers into a default multi-weaver that resolves interactions automatically in a well-defined and reasonable way. This requirement targets the scalability problem that is inherent to the feature interaction problem. It enables an extensible and scalable solution to the problem of composing weavers.

We define the default multi-weaver behavior according to the following principles:

1. **Preserving behavior of individual weavers:** a default multi-weaver preserves the behavior of the individual composed weavers as observed when weaving their respective single-extension programs. For example, a multi-weaver for a composition of COOL and AspectJ would weave pure AspectJ (COOL) programs in exactly the same manner as a stand-alone AspectJ (COOL) weaver would have.
2. **Default foreign advising.** Syntactically, an aspect is a mixture of Java code and extension-specific terms. The default foreign advising behavior allows an aspect to advise a foreign aspect by advising Java statements within its source code, and *only* those statements. For example, a COOL coordinator embeds Java expressions within **`requires`**, **`on_exit`**, and **`on_entry`** constructs. In a default composition of COOL and AspectJ, AspectJ aspects advise COOL coordinators by advising only their Java expressions. Under this behavior, the `Logger` aspect (Listing 8) advises all field access join points within **`requires`**, **`on_exit`**, and **`on_entry`** expressions of the `Stack` coordinator (Listing 4).
3. **Default co-advising.** The co-advising behavior controls matching and ordering of multi-extension advice at a join point. The default matching policy is to unify the individual matching results of the composed extensions. The selected multi-extension advice include all pieces of

advice that match the join point under the semantics of their extensions. Each individual aspect weaver selects advice only from its own aspects, and does not interfere with the matching in foreign weavers.

The default ordering policy relies on the advice types being similar in all AspectJ-compatible extensions. An aspect advises a program by transforming computations at certain join points. We identify three types of transformations: to add advice before a join point computation, to add the advice after the computation, and to introduce the advice instead of (around) the computation. When the multi-weaver selects multi-extension advice at a join point, the default multi-extension ordering behavior is to run multi-extension **before** advice first, then multi-extension **around** advice, and finally the multi-extension **after** advice. For example, a default multi-weaver for a composition of COOL and AspectJ would order `lock` and `unlock` (COOL advice) to execute before and after AspectJ's **`around`** advice, respectively. The multi-weaver also preserves a partial order of same-extension advice within the selected multi-extension advice.

4.4.3 Customizability

Although the default multi-weaver implements a reasonable behavior, a composition specification may define special foreign advising and co-advising behavior. For example, a foreign advising specification for a composition of COOL and AspectJ might choose to allow AspectJ aspects to advise COOL's (*un*)*lock* computations as *advice-execution* join points. Hence, the framework must also allow the language designer to configure the multi-weaver to comply with the composition-specific foreign advising and co-advising specifications.

5. Aspect-Oriented Architecture

This section introduces a practical component-based and aspect-oriented architecture that facilitates the development of aspect weavers, and supports the integration of independently developed aspect weavers into a multi-weaver. We introduce the architecture in three steps. First, we explain the design decisions for decoupling *extension-specific* from *common* components. We refer to the extension-specific components as the *aspect mechanism*, and to the common components as the *platform*. The platform is implemented once; it provides facilities that, if reused, significantly ease the development of new weavers. Second, we present the design principles and decisions that allow the multiple aspect mechanisms and the platform to be automatically composed into a default multi-weaver. Third, we present a solution to the multi-weaver customizability problem. The architecture provides support for configuring the default multi-weaver to comply with a specialized composition specification.

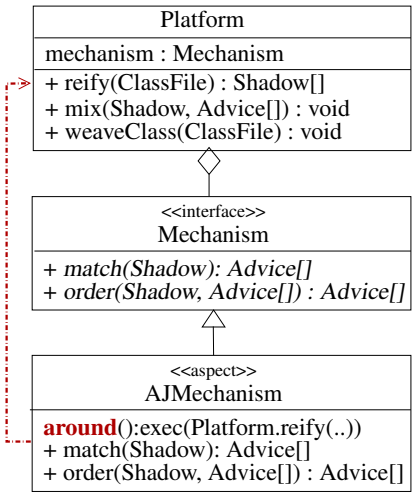

Figure 3. A stand-alone weaver

5.1 Decoupling

We use the four-process weaver model [23] to identify the *extension-specific* and the *common* weaver components.

The **`reify`** process of a weaver constructs shadows for the base language classes and the extension aspects. As a part of its functionality, the process realizes a base representation function, i.e., a function that builds shadows for base program classes. Because the weaver is AspectJ-compatible, it can be realized using the base shadow domain and the base representation function of AspectJ.[10] Thus, the base shadow domain and the base representation function of AspectJ can be shared by all AspectJ-compatible weavers.

If a weaver's extension does not allow an aspect to advise aspects (e.g., a coordinator in COOL cannot advise other coordinators), then the common base representation function realizes the weaver's **`reify`** process in full. However, in the more general case (e.g., in AspectJ, an aspect can advise itself, as well as other aspects), a weaver needs to realize an extension-specific representation function, i.e., a function that builds shadows for aspects. The **`reify`** process of the weaver is thus a composition of the *common* base representation function and the *specific* representation function for aspects.

The **`match`** and **`order`** processes of a weaver are *extension-specific*. An individual weaver matches advice in its own aspects; and orders only extension-specific advice. The **`mix`** process weaves the ordered pieces of advice by transforming the shadow. Since an advice defines a shadow transformer function, **`mix`** can be modeled as a *common* extension-independent process that iteratively applies the advice transformers to the shadow.

Figure 3 depicts the design of an AspectJ weaver as a composition of common and AspectJ-specific components.

[10] The base shadow domain of AspectJ includes all shadows except for *advice-execution*.

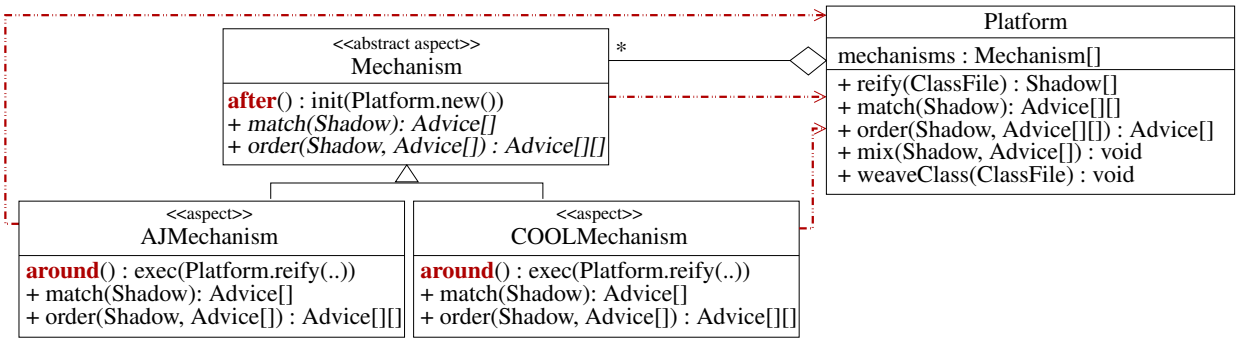

Figure 4. A default multi-weaver

There are two main architectural parts: (a) a *platform* that provides *common* facilities; (b) a *mechanism* that implements *extension-specific* behavior. The platform's behavior is realized by the Platform class. The mechanism is realized by an aspect that implements the Mechanism interface. The dashed arc in the figure denotes advising.

5.1.1 Platform

The platform provides the base shadow domain. Its methods **`reify`**, **`mix`**, and weaveClass implement the common weaver's operations: **`reify`** uses base shadows to represent base classes; **`mix`** weaves advice at each shadow; and weaveClass implements the high-level weaving algorithm (Listing 1).

5.1.2 Mechanism

Each mechanism is realized as an aspect that implements the Mechanism interface by realizing the extension-specific matching and ordering processes via the implementation of the methods **`match`** and **`order`**, respectively. If a mechanism's extension allows an aspect to advise other aspects, then the mechanism realizes the extension-specific representation function as an **around** advice that refines executions of the platform's **`reify`** method. The methods and the advice are implemented with the following conception: the mechanism uses base shadows as a representation domain for base classes; the advice uses base and extension-specific shadows as a representation domain for aspects;[11] and the advice defers to the platform the representation of base classes.

5.2 Composability

The architecture supports the compositions of multiple aspect mechanisms into a multi-weaver. Figure 4 illustrates the extended architecture by showing a default multi-weaver for a composition of COOL and AspectJ. In the extended architecture, the multi-weaver is realized by the platform. The platform mediates between the composed mechanisms, and manages their collaborative application. Furthermore, the Mechanism interface is replaced with the abstract Mechanism aspect, which defines the abstract methods **`match`** and **`order`**. In addition, the aspect advises the Platform's constructor to register the mechanisms with the platform.[12]

At an abstract level, the multi-weaver implements the same high-level weaving process as a stand-alone weaver (Listing 1). The four subprocesses of the multi-weaver are built by integrating and unifying the corresponding processes of the individual weavers. The **`reify`** process of the multi-weaver represents base classes, and aspects that are written in different extensions. The **`match`** and **`order`** processes of the multi-weaver select and order multi-extension advice, respectively. In the extended architecture, Platform provides the methods **`match`** and **`order`** to realize these processes. The **`mix`** process weaves the ordered multi-extension advice by transforming the shadow.

We enable composability of aspect mechanisms by introducing additional design principles:

5.2.1 Mandatory Aspect Representation

To enable a default foreign advising behavior, an aspect mechanism *must* realize an extension-specific aspect representation function, even if the function is *not* normally required for its own stand-alone operation. This policy ensures that a multi-weaver builds shadows for *all* aspects in a multi-extension program, thus letting an aspect observe and advise Java shadows in any foreign aspect. For example, a stand-alone weaver for COOL does not advise coordinators, and thus does not need to represent them. A multi-weaver for a composition of COOL and AspectJ, in contrast, uses the COOL representation function for exposing the coordinators to AspectJ aspects. The aspects can then advise Java shadows within the coordinators.

Intuitively, the aspect representation function provides the most fine-grained representation of an aspect that includes *all* base shadows for its Java fragments, and ded-

[11] Extension-specific shadows represent constructs, declarations, and expressions that are specific to the extension, e.g., the *advice-execution* shadow in AspectJ.

[12] This behavior could have been realized in an object-oriented manner, but AOP enables a more elegant design.

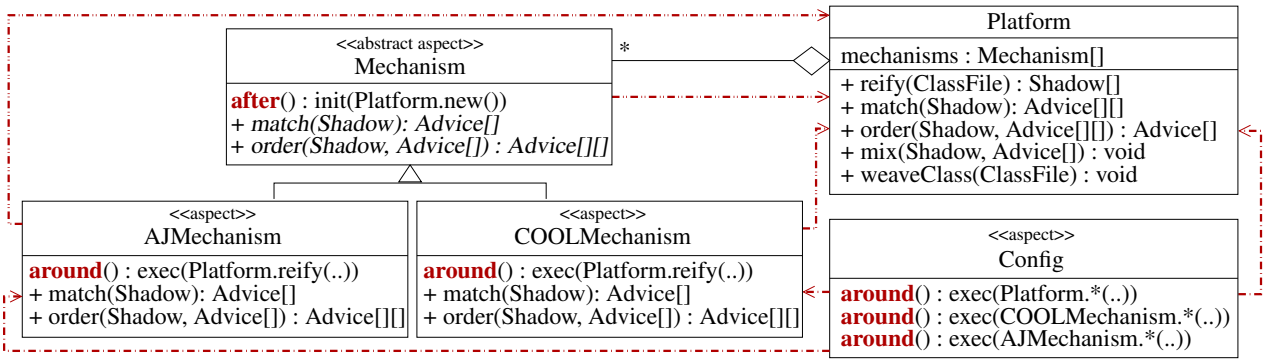

Figure 5. A customized multi-weaver

icated extension-specific shadows for *all* the extension-specific computations. For example, a function for representing coordinators must build shadows for all Java fragments within a coordinator's code (e.g., Java expressions *within* a **requires** statement), and *on_entry*, *on_exit*, *requires*, *lock*, and *unlock* shadows for all the respective computations.

5.2.2 Parallel Matching of Multi-extension Advice

To enable a default multi-extension advice matching behavior, the **match** method of the platform should run the **match** methods of the composed mechanisms in parallel. The multi-extension advice selected at a shadow is then a list of extension-specific advice sets that are produced by the composed aspect mechanisms.

5.2.3 Uniform Advice Types

To enable a default multi-extension advice ordering behavior, an aspect mechanism should partition advice into three ordered sets, namely before, around, and after. In terms of the architecture, the **order** method of a mechanism returns a list of three advice arrays, the first contains **before** advice, the second contains **around** advice, and the third contains **after** advice. The platform's **order** method then runs the **order** methods of the composed mechanisms in parallel, and linearizes their results into a single advice vector in accordance with the default multi-extension ordering policy.

A composition of the aspect mechanisms with the platform produces a multi-weaver with a default behavior. The aspect representation principle enables a default foreign advising behavior, and the other principles enable a default co-advising behavior. The multi-weaver uses as its common shadow domain the union of the common base shadow domain and all the extension-specific shadow domains.[13] It represents the multi-extension aspects and the base classes as appropriate for the composed mechanisms. It has a well-defined multi-extension advice matching and weaving behavior; and it uses the **order** method of the platform for ordering advice.

[13] For simplicity, we assume that the intersection of extension-specific shadow domains is empty [22].

5.3 Customizability

Of course, the default behavior of the multi-weaver may differ from the actually desired one. The specification may dictate foreign advising and co-advising rules that involve several extensions. For example, the specification may require an aspect in AspectJ to advise executions of *lock* and *unlock* in COOL as *advice-execution* join points. Generally, the foreign advising rules alter the semantics of the individual extension for advising foreign aspects. The co-advising rules specify a collaborative behavior for multi-extension aspects that advise the same join point. These rules are composition-specific and thus cannot be defined on the level of an individual extension.

To this end, the architecture provides a Config aspect that customizes the behavior of the default multi-weaver (Figure 5). The configuration aspect implements the composition-specific foreign advising behavior by extending and overriding the **match** and **order** methods of the aspect mechanisms. The aspect specializes the co-advising behavior by advising the **match** and **order** methods of the platform.

The architecture thus supports the construction of a multi-weaver with a customized behavior. The multi-weaver **reify** method recognizes and represents properly aspects of all the composed extensions using a common shadow domain. The adapted **match** and **order** methods of the individual mechanisms select and order extension-specific advice according to the foreign advising specification. The customized multi-weaver **match** and **order** methods select and order the multi-extension advice in accordance with the co-advising specification.

In sum, the architecture (Figure 5) establishes the fundamental principles for designing composable aspect mechanisms. In Section 6 we apply these principles to build a concrete co-weaving system for composing multiple extensions (AWESOME). Using AWESOME, we then implement an aspect mechanism for COOL, another for AspectJ, and then combine the two to produce an AWESOME weaver for the AOP language COOLAJ, which is described in Section 7.

6. Implementation by Refactoring AspectJ

As a proof of concept, we realized the weaving system and the mechanisms by refactoring the `ajc` compiler and the COOL weaver. In the `ajc` code, shared and AspectJ-specific operations are intertwined. Through refactoring we untangled and separated these two kinds of operations, moving the ones in common to the Platform and modularizing the rest in the AspectJ mechanism.[14] The weaver for COOL, on the other hand, uses methods to represent Java classes, and does not have a representation for coordinator classes. In our system, we use AspectJ shadows (with the exception of *advice-execution* shadows) as a base shadow domain. The refactoring here involved the use of *method-execution* shadows for advice matching and weaving; and providing a shadow representation for the coordinator classes.

6.1 Implementing a Platform

The platform is realized by the Platform class. A list of plugged aspect mechanisms is stored in the mechanisms instance variable. weaveClass is a TEMPLATE METHOD [15] that implements the weaving process (Listing 1) using **`reify`**, **`match`**, **`order`**, and **`mix`**.

The **`reify`** method represents a Java class as a set of shadows. We implemented it by factoring out all operations that represent aspects from the representation function **`reify`** of the original AspectJ weaver. The **`match`** method selects advice by calling its **`match`** counterparts in the individual mechanisms:

```
public Advice[][] match(Shadow shadow) {
  Advice[][] result = new Advice[mechanisms.length][0];
  for (int i=0;i<mechanisms.length;i++)
    result[i] = mechanisms[i].match(shadow);
  return result;
}
```

The **`order`** method calls the **`order`** methods of the individual mechanisms, and then linearizes the multi-extension advice:

```
public Advice[] order(Shadow shadow,
                      Advice[][] multiAdvs) {
  Advice[] bfAdv = new Advice[0];
  Advice[] ardAdv = new Advice[0];
  Advice[] afAdv = new Advice[0];
  for (int i=0;i<mechanisms.length;i++) {
    Advice[][] mechAdvs =
      mechanisms[i].order(shadow, multiAdvs[i]);
    bfAdv = addAll(bfAdv, mechAdvs[0]);
    ardAdv = addAll(ardAdv, mechAdvs[1]);
    afAdv = addAll(afAdv, mechAdvs[2]);
  }
  return addAll(ardAdv, addAll(bfAdv, afAdv));
}
```

[14] We also moved the shadow transformation functionality in the Shadow.transform method to the Advice class.

where addAll is an auxiliary method that takes two argument arrays, and concatenates them by appending the second one to the first one. The **`order`** method schedules **around** advice to be woven first, so that **before** and **after** advice "wrap" any **around** advice at the same shadow. Note that weaving order is not the same as execution order.

Finally, the **`mix`** method sequentially applies advice transformers to the shadow:

```
void mix(Shadow shadow, Advice[] advs) {
  for(Advice a:advs) a.transform(shadow);
}
```

The transform method integrates the advice instructions into the shadow.

6.2 An Abstract Aspect Mechanism

We implemented the mechanisms for AspectJ and COOL as aspects that extend the abstract SINGLETON [15] aspect, named Mechanism:

```
public abstract
aspect Mechanism {
 after(Platform mw):
   initialization(Platform.new(..)) && this(mw) {
     mw.mechanisms =
       addAll(mw.mechanisms, new Mechanism[]{this});
 }
 public abstract
 Advice[] match(Shadow shadow);
 public abstract
 Advice[][] order(Shadow shadow, Advice[] advs);
}
```

The **after** advice ensures that aspect mechanism instances are created and plugged into the platform as soon as the platform is instantiated. The concrete mechanisms (AJMechanism and COOLMechanism) provide an implementation for **`match`** and **`order`** and override the Platform.**`reify`** method by advising it with **around** advice.

6.3 Implementing an AspectJ Mechanism

The AJMechanism aspect advises the representation **`reify`** method of the platform. If the argument class is an AspectJ aspect, then the advice provides a shadow representation for it; otherwise, the advice proceeds:

```
Shadow[] around(ClassFile cf): args(cf) &&
  execution(Shadow[] Platform.reify(ClassFile)) {
  return isAJAspect(cf) ? reifyAspect(cf) : proceed(cf);
}
```

where isAJAspect determines whether or not the argument class represents an aspect; and reifyAspect constructs shadow representation of the aspect class. The aspect representation includes AspectJ-specific *advice-execution* shadows.

The **before** advice to the weaveClass method introduces to the multi-weaver an intertype declaration mechanism:

```
before(ClassFile cf): args(cf) &&
  execution(void Platform.weaveClass(ClassFile)) {
  applyIntroductions(cf);
}
```

The **match** and the **order** methods are copied from the original code. We omit them, as well as isAJAspect and reifyAspect, due to space considerations.

6.4 Implementing a COOL Mechanism

The refactoring of the COOL mechanism includes a change to the front-end for translating source COOL coordinators into annotated Java classes. The annotations mark the `lock_` and `unlock_` methods of the coordinator class and identify the **requires**, **on_entry**, and **on_exit** instruction blocks within these methods.

The COOL mechanism introduces shadow types for *lock*, *unlock*, *requires*, *on_enter*, and *on_exit* computations. The *lock* and *unlock* shadows represent executions of the `lock_` and `unlock_` methods. The *requires*, *on_enter*, and *on_exit* shadows represent executions of the corresponding COOL expressions and statements. They map to blocks of instructions within the `lock_` and `unlock_` methods. The bodies of the **requires**, **on_entry** and **on_exit** constructs are Java expressions and statements. The mechanism represents them using the base shadow domain (*field-get* and *field-set* shadows).

The COOLMechanism aspect advises the weaveClass and the **reify** methods of the platform. The **after** advice to the weaveClass method introduces into a coordinated (target) class a coordinator field and getter methods, and transforms the constructor of the class:

```
after(ClassFile cf): args(cf) &&
  execution(void Platform.weaveClass(ClassFile)) {
  ClassFile coordAspect = findAspect(cf);
  if (coordAspect!=null) {
    addCoordField(cf, coordAspect);
    transformConstructor(cf, coordAspect);
    addGetterMethods(cf, coordAspect);
}}
```

The advice around the **reify** method is similar to the corresponding advice in the AJMechanism aspect: if the argument class is a COOL coordinator class, then the advice provides a shadow representation for it; otherwise, the advice proceeds.

The COOL mechanism also provides an implementation for **match** and **order**. **match** selects `lock` and `unlock` pieces of advice by matching the coordinator classes against the *method-execution* shadows. The **order** method schedules the `lock` advice to run before the `unlock` advice.

7. Case Study: An AWESOME Weaver for COOLAJ

Plugging the AJMechanism and the COOLMechanism aspects into the composition Platform produces a multi-weaver with a default behavior. It lets aspects advise join points within **requires**, **on_entry**, and **on_exit** expressions of coordinators. It lets coordinators synchronize methods that are defined within aspects; and it allows coordinators and aspects to co-advise the same method. Although this default behavior is reasonable, a specific multi-extension composition of AspectJ and COOL may require different semantics. In this section we specify such a multi-extension AOP language named COOLAJ. We implement a weaver for COOLAJ by customizing the default multi-weaver using the Config aspect.

7.1 Informal Specification for COOLAJ

The specification for COOLAJ is independent of the AWESOME architecture. It is based only on the syntax and semantics of the AspectJ and COOL languages; not on their implementation. COOLAJ is specified as a conservative composition of AspectJ and COOL, i.e., it follows as much as possible the original semantics of AspectJ and COOL. Specifically, in COOLAJ an aspect is woven into classes and aspects according to the weaving semantics of AspectJ. Similarly, a coordinator is woven into classes according to the weaving semantics of COOL. The specification for COOLAJ differs from the default behavior when it comes to dealing with foreign advising and co-advising:

7.1.1 Foreign Advising

In COOLAJ, aspects advise executions of coordinators through *field-get* and *field-set* join points that are located within **requires**, **on_entry**, and **on_exit** expressions; and through *advice-execution* join points that represent *lock* and *unlock* computations (i.e., executions of `lock_` and `unlock_` methods of the COOL coordinator classes).

The foreign advising specification poses several restrictions on advising join points within a coordinator:

- In the COOLAJ specification, an access (read or write) to a **condition** field can only be advised with **before** or **after** advice. This way aspects cannot override values of these fields, but are still able to observe their access patterns. This restriction is important for protecting the synchronization logic of a coordinator.
- An execution of a *lock* or an *unlock* computation is advisable by aspects as an *advice-execution* join point. However, aspects are limited to advising these join points with **before** and **after** advice only. This restriction ensures that the locking and unlocking operations imposed by coordinators are not overridden by aspects, and always apply in the correct order.

The specification permits coordinators to advise methods that are declared within aspects in the same way as methods within classes. The specification restricts coordinators (aspects) from advising any synthetic code introduced by the foreign mechanism, e.g., coordinators do not advise advice methods in aspect classes, and aspects do not advise getter

Listing 9. LogAdviceOnStack

```
public aspect LogAdviceOnStack {
  pointcut scope():
   !cflow(within(LogAdviceOnStack));
  pointcut tgt(): execution(* Stack.*(..));
  before(): scope() &&
    cflow(tgt()) && !cflowbelow(tgt()) {
    System.out.println(thisJoinPoint);
  }
}
```

methods that are introduced into the coordinated classes by the COOL mechanism.

7.1.2 Co-advising

The COOLAJ co-advising specification coordinates the collaborative application of aspects and coordinators on the same program method:

- The `lock` (`unlock`) advice of COOL is executed before (after) the **`before`**, **`around`**, and **`after`** advice of AspectJ.
- From the perspective of AspectJ aspects, COOL advice executes in the control flow of the method execution join point it advises.

Example For illustration, consider the `LogAdviceOnStack` aspect (Listing 9). Under the semantics of AspectJ, the aspect logs all advice (except for its own) woven at `Stack` *method-execution* shadows. The

```
cflow(tgt)&&!cflowbelow(tgt())
```

pointcut selects not only `tgt()` join points, but also join points within aspects that advise the `tgt()` join points. In particular, `LogAdviceOnStack` would advise join points within the `Logger` aspect (Listing 8), if the two are used together with the `Stack` class (Listing 3). If this aspect is run together with the `Stack` class (Listing 3) and the `Stack` coordinator (Listing 4) under the semantics of COOLAJ, then: (1) it logs executions of the coordinator; and (2) an execution of the `LogAdviceOnStack` advice that prints the *method-execution* join point is synchronized by the coordinator (along with the original method body).

7.2 Customization

We realized a multi-weaver for COOLAJ by providing a Config aspect with three pieces of advice: one implementing the co-advising rules, and the other two realizing the foreign advising rules.

7.2.1 Customizing Co-advising

The AspectJ weaver realizes the semantics of the **`cflow`** pointcut designator by introducing special advice at program shadows. Specifically, every shadow that matches an argument pointcut of a **`cflow`** pointcut is wrapped with a CFlowPush advice and a CFlowPop advice. The CFlowPush advice runs before any other advice at a join point, and pushes the join point on the AspectJ's join point stack. CFlowPop runs after all the other advice and pops the join point off the stack. These advice thus mark the start and the end of a join point's control flow. For example, the pointcut of the `LogAdviceOnStack`'s advice causes the AspectJ weaver to weave the CFlowPush and CFlowPop pieces of advice at the `Stack.push` *method-execution* shadow.

Config implements the co-advising rules of COOLAJ by advising the Platform.**`order`** method. The advice orders COOL advice to be woven between the CFlowPush and CFlowPop advice, but around any other AspectJ advice:

```
Advice[] around(): execution(Advice[] Platform.order(..)) {
    Advice[] advs = proceed();
    advs = mvAdv(advs, LockAdv.class, CFlowPush.class);
    return mvAdv(advs, UnlockAdv.class, CFlowPop.class);
}

private Advice[] mvAdv(Advice[] advs, Class fromAdv,
                                      Class toAdv) {
  int fromPos = elTypePos(advs, fromAdv);
  if (fromPos < 0) return advs;
  int toPos = elTypePos(advs, toAdv);
  if (toPos < 0) toPos = advs.length;
  if (fromPos < toPos) toPos--;
  return move(advs, fromPos, toPos);
}
```

where LockAdv and UnlockAdv classes respectively implement `lock` and `unlock` advice of COOL; elTypePos returns a first position of an object of a given class in the array; and the move method moves an element of an array from one position to another. Specifically, move(advs, fromPos, toPos) moves an element at the position fromPos of the advs array to the position toPos, and shifts elements between fromPos (exclusively) and toPos (inclusively) to the left, if fromPos $<$ toPos, or to the right, if fromPos $>$ toPos. mvAdv is an auxiliary method that co-orders COOL and AspectJ advice.

If the multi-extension advice array contains no COOL advice then Config does not affect it. Otherwise, if the array contains the **`cflow`** advice then Config orders LockAdv (UnlockAdv) to be woven immediately before CFlowPush (CFlowPop), so that at run time the LockAdv (UnlockAdv) advice runs immediately after (before) the CFlowPush (CFlowPop) advice. If the array contains no **`cflow`** advice, then COOL advice is scheduled to be woven the last, thus dominating the AspectJ advice at run-time.

7.2.2 Normalizing Shadow Types

To allow aspects to advise *lock* and *unlock* computations as *advice-execution*s, Config normalizes *lock* and *unlock* shadows of COOL with *advice-execution* shadows of AspectJ

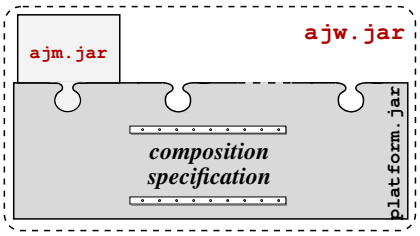

Figure 6. An AspectJ weaver

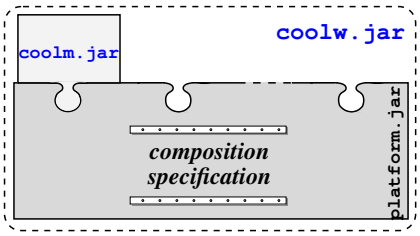

Figure 7. A COOL weaver

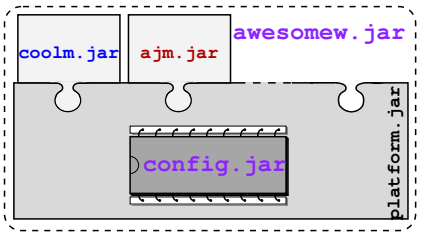

Figure 8. An AWESOME weaver

by advising calls to the **match** and **order** methods of the AJMechanism aspect:

```
Advice[] around(Shadow shadow): args(shadow) &&
  (call(Advice[] AJMechanism.match(..)) ||
   call(Advice[][] AJMechanism.order(..))) {
     return proceed(isLockOrUnlock(shadow) ?
                    maskAsAExec(shadow) : shadow);
}
```

where isLockOrUnlock tests if shadow is a *lock* or *unlock* shadow, and maskAsAExec masks the COOL shadow as an *advice-execution* shadow. As a result, AspectJ advises the *lock* and *unlock* shadows as if they were *advice-execution* shadows.

7.2.3 Restricting Advisability

Config restricts the advising of join points within COOL coordinators by advising the executions of the AJMechanism.**match** method:

```
Advice[] around(Shadow shadow): args(shadow) &&
  execution(Advice[] AJMechanism.match(..)) {
    Advice[] advs = proceed(shadow);
    if (isCondFieldAccess(shadow) ||
        isLockOrUnlock(shadow))
      advs = removeElType(advs, AroundAdvice.class);
    return advs;
}
```

where isCondFieldAccess checks if the shadow represent access to a condition field of a COOL coordinator, and removeElType removes all elements of a given type from the array.

8. Evaluation

To evaluate our approach, we integrated our multi-weaver framework with the `ajc` AspectJ compiler. The main `ajc` class runs the front-end and eventually weaves bytecode classes by invoking the `weave` method on the `org.aspectj.weaver.bcel.BcelClassWeaver` class. We modified this method to call instead the `Platform.weaveClass` method for weaving. This permitted to "plug" a specific multi-weaver into `ajc` by putting a corresponding implementation of the `Platform` class on the class path, and running the AspectJ compiler as usual. We evaluated the pluggability, correctness, and performance.

8.1 Third-party Composition

First, we evaluated the pluggability feature of AWESOME by constructing three different weavers from the same building blocks. The building blocks are jar files containing compiled aspects and classes. **platform.jar** is the stripped down platform containing the `Platform` class and the abstract `Mechanism` aspect. The jars **ajm.jar** and **coolm.jar** contain the concrete independently developed aspects AJMechanism and COOLMechanism for AspectJ and COOL, respectively. **config.jar** contains the `Config` aspect for customizing the composition of COOL and AspectJ.

We verified that it is possible, using the command line, to construct weavers for AspectJ, COOL, and COOLAJ from the four building blocks. We constructed a stand-alone AspectJ weaver, named **ajw**, by plugging just the AspectJ mechanism into the platform. The command line is (Figure 6):

```
ajc -inpath platform.jar;ajm.jar -outjar
ajw.jar
```

where `ajc` is the original (non-refactored) version of the AspectJ compiler. The `inpath` option directs `ajc` to weave classes within jar files. The `outjar` option directs the compiler to save the woven classes into a separate jar file. To construct a stand-alone COOL weaver, named **coolw**, we plugged only the COOL mechanism (Figure 7):

```
ajc -inpath platform.jar;coolm.jar -outjar
coolw.jar
```

and to construct a multi-weaver, named **awesomew**, that combines AspectJ and COOL, we ran (Figure 8):

```
ajc -inpath platform.jar;ajm.jar;coolm.jar;
config.jar -outjar awesomew.jar
```

To compile and run a multi-extension aspect program, a file with unwoven bytecode **unwoven.jar** (an unwoven program including aspect and base classes) was passed to the multi-weaver to produce a woven file:

```
java -cp <weaver>.jar;aspectjtools.jar
org.aspectj.tools.ajc.Main -inpath
unwoven.jar -outjar woven.jar
```

where <weaver> was one of **ajw**, **coolw**, or **awesomew**; and **woven.jar** is the woven bytecode program that can be run on a JVM as a regular Java program.

8.2 Testing

Second, we tested the three weavers to determine with high confidence that indeed **ajw** implements the semantics of AspectJ; **coolw** implements the semantics of COOL; and **awesomew** realizes the specification for COOLAJ. We did this by observing the runtime behavior of test programs; by inspecting their woven bytecode; by analyzing join point traces; and, when possible, by comparing the results to programs compiled with `ajc` or `abc` [5]. Because the framework is based on `ajc`, which is assumed correct, we focused our tests on a coverage of the newly introduced and refactored behavior.

8.2.1 Testing ajw

The **ajw** weaver can be evaluated by comparing **ajw**-woven to `ajc`-woven bytecode. In fact, the main difference between **ajw** and `ajc` is in the design and implementation of the **reify** process. In the implementation of **ajw** we disentangled the monolithic **reify** process of `ajc` into a common platform **reify** method and an AspectJ-specific advice of the AspectJ mechanism. The other processes were either left unchanged (e.g., **match** and **order**), or undergone a coarse-grained (and assumed behavior-preserving) transformations (e.g., **mix**). Thus, we hypothesize that **ajw** is a behavior-preserving refactoring of `ajc`, if the **reify** processes of the two exhibit the same behavior, i.e., given a Java class or an AspectJ aspect they build identical shadow representations.

To test the **reify** process and reason about its shadow representation, we generated an exhaustive join point trace by weaving together three classes and aspects: Stack.java (Listing 3); LogAll.aj, and TouchAll.aj. The `LogAll` aspect advises with **before**, **around**, and **after** advice all the join points in the program, except those within the aspect itself (to prevent an infinite loop),[15] and logs the join points to a file. `TouchAll` also advises everything but itself,[16] but just "touches" the join points with an empty advice.

The woven bytecode is run by a main program that creates a `Stack` object and invokes `push` and `pop` in a single thread. The execution produces an exhaustive trace of the join points within `Stack` and within `TouchAll`. This trace provides a good insight into behavior of the reify process, because it covers almost all types of join points and includes join points within both Java classes and AspectJ aspects.

We executed **ajw**-woven and `ajc`-woven bytecode using the same test program and obtained identical join point traces. We therefore conclude that, at least on this benchmark example, **ajw** behaves the same as `ajc`, and is likely to exhibit an `ajc`-equivalent behavior in general.

[15] `LogAll` advises `!cflow(within(LogAll))` join points

[16] `TouchAll` advises `!cflow(within(TouchAll))`

8.2.2 Testing coolw

We tested whether the runtime behavior of **coolw**-woven bytecode complies with the dynamic semantics of COOL [27] (i.e., if multiple threads are properly synchronized on the woven target methods). We compiled together Stack.java (Listing 3) and Stack.cool (Listing 4), which employs all features of COOL (i.e., **selfex**, **mutex**, **requires**, **on_entry**, **on_exit**, and access to **private** field of a coordinated class from a coordinator aspect). As part of our tests we inspected the StackCoord.java file that was constructed by the COOL front-end from Stack.cool, and we inspected the **coolw**-woven bytecode that was produced by weaving the `StackCoord` coordinator class into the `Stack` Java class.

We ran the woven program and observed its runtime behavior. The test program created a `Stack` instance with a very small capacity (size of 5), and invoked its methods concurrently by five reader and five writer threads. A reader thread attempted to remove 5000 objects from the stack, while a writer thread attempted to add 5000 objects onto the stack. The test program completed successfully (i.e., executed all the threads to completion without throwing an exception), indicating, with a high probability, that the behavior of the woven bytecode is correct. Additional inspection of traces verified that the stack was properly synchronized.

We tested the front-end translator by comparing the generated StackCoord.java against the manual translation presented in Listing 5. StackCoord is said to pass the test, if we can conclude that its `lock_` and `unlock_` methods encode the same behavior as the corresponding `lock_` and `unlock_` methods in Listing 5. We tested **coolw** by comparing the bytecode of the woven `Stack` class against the the manually-woven class presented in Listing 6.

All three tests succeeded. Our COOL implementation exhibited the correct dynamic and compilation semantics on the input program that comprised Stack.java and Stack.cool. We consider the input program to be a representative COOL application since it uses all the features in COOL. We thus conclude with a high degree of confidence that our COOL implementation would generally weave COOL programs correctly.

8.2.3 Testing awesomew

We hypothesize that **awesomew** implements correctly the semantics of COOLAJ. To test this hypothesis we verified that:

1. Given a program with only aspects and classes as input, **awesomew** weaves the program according to the semantics of AspectJ;
2. Given a program with only coordinators and classes as input, **awesomew** weaves the program according to the semantics of COOL;
3. Given a program with coordinators, aspects and classes as input, **awesomew** weaves the coordinators into their

matching classes according to the semantics of COOL; weaves the aspects into classes and other aspects according to the semantics of AspectJ; weaves aspects into coordinators according to the foreign advising specification; and coordinates the weaving of multi-extension advice according to the co-advising specifications of COOLAJ.[17]

The first two cases were validated using the same testing strategy as for **ajw** and **coolw** and by comparing the output of **awesomew** to that of **ajw** and **coolw**. The details are omitted. To validate the foreign advising and co-advising behavior in the third case, we compiled Stack.java, Stack.cool, LogAll.aj, and TouchAll.aj, and we tested the woven bytecode.

We verified the weaving of aspects into coordinators by inspecting the join point trace within the control flow of the `StackCoord` class. **awesomew** correctly weaves aspects into coordinators, if the trace complies with the COOLAJ foreign advising specification, (which defines the shadow representation of COOL coordinators; the normalization between COOL and AspectJ shadow types; and mapping between AspectJ advice types and coordinator-located shadows). In particular, we verified that the trace contains only expected join points, that it reflects executions of `lock_` and `unlock_` methods as *advice-execution* join points, and that **around** advice of `TouchAll` and `LogAll` are properly filtered (e.g., not applied at *advice-execution* join points).

We also tested the ordering of multi-extension advice on a program that contains Stack.java, Stack.cool, LogAdviceOnStack.aj (Listing 9), and TouchAll.aj. The execution trace of a Stack method reflected that: (1) execution of `StackCoord` is advised by `LogAdviceOnStack`; and (2) the first and the last *advice-execution* join points around a `Stack` method execution join point that are not in the control flow of `LogAdviceOnStack` are executions of `StackCoord`, `lock_` and `unlock_` methods, respectively. The first result shows that `StackCoord` advice executes in the control flow of the join point it advises. The second result shows that `StackCoord` advice takes precedence over AspectJ advice at the same join point.

awesomew passed all these tests. We verified that **awesomew** weaved our input programs according to the COOLAJ semantics. The input programs provide a good coverage of the COOLAJ specification. Therefore, we conclude with high confidence that **awesomew** performs a correct weaving of COOLAJ programs.

8.3 Performance

Finally, we evaluated the runtime performance of the bytecode produced by the framework weavers. This is intended to verify that the quality of woven bytecode is unaffected by the improved design of the weaver. Specifically, we validated the following hypotheses:

[17] Coordinators never weave other coordinators.

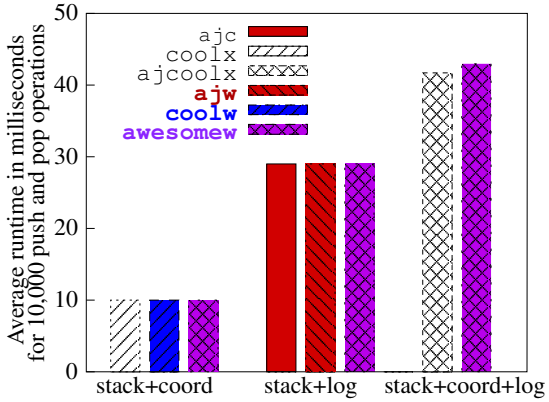

Figure 9. AWESOME performance

1. The runtime performance of **ajw**-woven bytecode is the same as `ajc`-woven;
2. The runtime performance of **coolw**-woven bytecode is the same as `coolx`-woven;
3. The runtime performance of **awesomew**-woven bytecode is the same as:
   (a) `ajc`-woven, for AspectJ programs;
   (b) `coolx`-woven, for source COOL programs;
   (c) `ajcoolx`-woven, for COOLAJ programs.

where `coolx` and `ajcoolx` are weaving algorithms for COOL and COOLAJ that are applied manually.

We ran a multi-threading COOL program that creates a `Stack` object and invokes its methods using ten writer-reader threads. A writer-reader thread performs 5000 pairs of push-pop operations. The test program reported the average running time of a thread (in milliseconds) over series of 10 runs. We also ran a single-threaded AspectJ program that created a `Stack` object, and invoked its `push` and `pop` methods 5000 times each. We measured the average running time of executing the operations over a series of 10 runs.

Figure 9 summarizes the measured execution times. Programs compiled with **ajw** and **coolw** are as efficient as those compiled with `ajc` and `coolx`. Programs compiled with **awesomew** are within 4% efficiency compared to (optimal) code woven mechanically using the `ajcoolx` algorithm and `ajc` as a back-end compiler. This indicates that the framework design overhead on the performance of the woven bytecode is negligible, i.e., there is almost no overhead to supporting the plugin architecture.

9. Related Work

The vision of domain specific aspect-oriented extensions dates back to early days of AOP (e.g., [8, 9]), but very few of the related works deal with making such extensions available concurrently. Those that do, leave much to be desired in terms of composability, customizability, and efficiency.

Reflex [37] and XAspects [36] do not support the level of composability or customizability that is necessary for resolving foreign advising. These frameworks implement the composition by translating source aspects in foreign extensions to aspects in a common target language. The translation introduces and exposes in the target aspects synthetic join points that do not exist in the source. However, in Reflex and in XAspects, foreign aspects cannot distinguish the synthetic from the genuine join points. Moreover, Reflex and XAspects provide no mechanism (or composition rules) for customizing the foreign advising behavior, thus preventing the integrator from being able to correct the faulty resolution of this feature interaction. As a result, aspect programs compiled in Reflex- and XAspects-based multi-extension weavers may exhibit incorrect behavior [22, 24].

In contrast, AWESOME provides automatically a default reasonable resolution of foreign advising, thus significantly simplifying the problem of resolving the interactions. The integrator may fine-tune the default behavior, but does not necessarily need to.

In Reflex there is ample support for configuring co-advising at the *aspect level*. A programmer can resolve interactions between aspects in a specific aspect program. AWESOME, on the other hand, supports customizability at the *language level*. A language designer can resolve the interactions between aspect extensions, thus affecting the behavior of all multi-extension programs. Extending AWESOME with aspect-level support for fine-tuning co-advising is a topic for future work.

Pluggable AOP [22] is a third-party composition framework that supports the composition of dynamic aspect mechanisms into an AOP interpreter. In Pluggable AOP, an aspect mechanism is a transformer of an AOP interpreter. Among the related frameworks, only Pluggable AOP addresses foreign advising by treating a foreign aspect mechanism as an open module [1] that can determine which join points within its aspects are advisable and which are hidden. However, there is no control in Pluggable AOP over how these join points are advised. Pluggable AOP is also restricted in its co-advising customizability. It allows the integrator to customize the co-advising behavior only indirectly by ordering the aspect mechanisms. Moreover, Pluggable AOP is not designed for efficiency. It is impractical for use in industrial settings.

In comparison, AWESOME supports flexible language-level customizability of both the co-advising and the foreign advising interactions; and employs an efficient compile-time weaving scheme.

AWESOME was demonstrated to successfully compose real-world extensions. Pluggable AOP, in contrast, uses "toy" languages as a proof of the concept. To the best of our knowledge, Reflex has not been shown to work with AspectJ. We only found a plugin that implements a limited subset of AspectJ. The plugin, however, does not advise AspectJ aspects correctly, emphasizing the general limitation of the Reflex framework to support foreign advising [24].

The composition of COOL [27] and AspectJ [20] presents an interesting case study with a representative complexity. The two are sufficiently different, thus demanding interesting design decisions to make them work together. Similar to AWESOME, XAspects too explored a composition of COOL and AspectJ. However, the XAspects weaver exhibits incorrect weaving behavior that may result in deadlock [22].

AWESOME is not limited to COOL and AspectJ. It can be generally applied to combine any reactive aspect mechanisms. To the best of our knowledge, all existing join point and advice aspect extensions fall into this category. Other more disparate aspect-oriented extensions are either non-reactive aspect mechanisms, which are not composable, e.g., Hyper/J [34, 38], or they are not "oblivious" [7, 13, 14] and can be composed trivially, e.g., Demeter [25, 26]. Aspect-oriented features other than advising, such as introductions, are easier to compose because it is easier to detect and resolve conflicts [16].

The problem of feature interactions in a multi-extension composition does not rise in the context of a single-extension AOP language. Therefore, related works on single-extension weavers [3, 5, 19, 29, 30, 39, 42] do not address or solve this problem.

10. Conclusion

This work studies the composition implementation problem in constructing multi-extension weavers. We present a practical third-party composition framework, named AWESOME, for composing multiple aspect extensions. The AWESOME framework was built systematically. It implements a specified set of composition requirements. It provides a default resolution of feature interactions in the composition. It also provides means for customizing the default resolution to comply with a given composition specification. AWESOME was tested and evaluated on real-world aspect languages. The runtime performance of compiled aspect programs is practically unaffected by the extensible design of the framework, making AWESOME also useful in practice.

AWESOME is unique in its approach to composing aspect extensions by assembling an aspect compiler. In AWESOME, an aspect mechanism is a plugin to the compile-time aspect weaver. The AWESOME framework simplifies the creation of new extensions, because writing a plugin is much simpler than writing a complete compiler. In order to evaluate our approach, we refactored the AspectJ `ajc` compiler and used it as a basis for our multi-weaver platform. But the refactoring of open source compilers is *not* a part of the integration methodology.

The **`ajw`**, **`coolw`**, and **`awesomew`** weavers themselves are also a modest contribution of this work. These AspectJ, COOL, and COOLAJ compilers do not just have a cool design, but also awesome performance.

References

[1] J. Aldrich. Open modules: Modular reasoning about advice. In *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP'05)*, number 3586 in Lecture Notes in Computer Science, pages 144–168, Glasgow, UK, July 25-29 2005. Springer Verlag.

[2] *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD'04)*, Mancaster, UK, Mar. 17-21 2004. ACM Press.

[3] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. An overview of CaesarJ. In Rashid and Aksit [35], pages 135–173.

[4] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: an extensible AspectJ compiler. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD'05)*, pages 87–98, Chicago, Illinois, USA, Mar. 14-18 2005. ACM Press.

[5] 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. In Rashid and Aksit [35], pages 293–334.

[6] J. Bonér. What are the key issues for commercial AOP use: how does AspectWerkz address them? In AOSD'04 [2], pages 5–6.

[7] C. Clifton and G. T. Leavens. Obliviousness, modular reasoning, and the behavioral subtyping analogy. In *Proceedings of the AOSD'03 Workshop on Software-engineering Properties of Languages for Aspect Technologies (SPLAT'03)*, Boston, Massachusetts, Mar. 18 2003. AOSD'03, ACM Press.

[8] K. De Volder, J. Brichau, K. Mens, and T. D'Hondt. Logic meta-programming, a framework for domain-specific aspect programming languages. Unpublished, 2001.

[9] M. D'Hondt and T. D'Hondt. Is domain knowledge an aspect? In *Proceedings of the ECOOP'99 International Workshop on Aspect-Oriented Programming*, Lisbon, Portugal, June 1999.

[10] *Proceedings of the 17th European Conference on Object-Oriented Programming (ECOOP'03)*, number 2743 in Lecture Notes in Computer Science, Darmstadt, Germany, July 21-25 2003. Springer Verlag.

[11] *Proceedings of the AOSD'05 Workshop on Foundations of Aspect-Oriented Languages (FAOL'05)*, Chicago, IL, USA, Mar. 14 2005. ACM Press.

[12] R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2005.

[13] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In P. Tarr, L. Bergmans, M. Griss, and H. Ossher, editors, *Proceedings of the OOPSLA 2000 Workshop on Advanced Separation of Concerns*. Department of Computer Science, University of Twente, The Netherlands, 2000.

[14] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In Filman et al. [12], pages 21–35.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing. Addison-Wesley, 1995.

[16] W. Havinga, I. Nagy, L. Bergmans, and M. Akşit. Detecting and resolving ambiguities caused by inter-dependent introductions. In *Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD'06)*, pages 214–225, Bonn, Germany, Mar. 20-24 2006. ACM Press.

[17] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In AOSD'04 [2], pages 26–35.

[18] *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, Los Angeles, California, May 1999. IEEE Computer Society.

[19] R. Jagadeesan, A. Jeffrey, and J. Riely. An untyped calculus for aspect oriented programs. In ECOOP'03 [10], pages 54–73.

[20] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01)*, number 2072 in Lecture Notes in Computer Science, pages 327–353, Budapest, Hungary, June 18-22 2001. Springer Verlag.

[21] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, number 1241 in Lecture Notes in Computer Science, pages 220–242, Jyväskylä, Finland, June 9-13 1997. Springer Verlag.

[22] S. Kojarski and D. H. Lorenz. Pluggable AOP: Designing aspect mechanisms for third-party composition. In *Proceedings of the 20th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'05)*, pages 247–263, San Diego, CA, USA, Oct. 16–20 2005. ACM Press.

[23] S. Kojarski and D. H. Lorenz. Modeling aspect mechanisms: A top-down approach. In *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*, pages 212–221, Shanghai, China, May 20-28 2006. ACM Press.

[24] S. Kojarski and D. H. Lorenz. Identifying feature interactions in aspect-oriented frameworks. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, Minneapolis, MN, May 20-26 2007. IEEE Computer Society.

[25] K. Lieberherr and D. H. Lorenz. Coupling Aspect-Oriented and Adaptive Programming. In Filman et al. [12], pages 145–164.

[26] K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS-Kent Publishing, 1996.

[27] C. V. Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, Northeastern University, 1997.

[28] D. H. Lorenz and S. Kojarski. Understanding aspect interactions, co-advising and foreign advising. In *Proceedings of ECOOP'07 Second International Workshop on Aspects, Dependencies and Interactions*, pages 23–28, Berlin, Germany, July 30 2007.

[29] H. Masuhara and G. Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In ECOOP'03 [10], pages 2–28.

[30] H. Masuhara, G. Kiczales, and C. Dutchyn. Compilation semantics of aspect-oriented programs. In FAOL'05 [11], pages 17–26.

[31] H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In *Compiler Construction*, number 2622 in Lecture Notes in Computer Science, pages 46–60, 2003.

[32] G. C. Murphy, R. J. Walker, and E. L. A. Baniassad. Evaluating emerging software development technologies: Lessons learned from assessing aspect-oriented programming. *IEEE Transactions on Software Engineering*, 25(4):438–455, 1999.

[33] G. C. Murphy, R. J. Walker, E. L. A. Baniassad, M. P. Robillard, A. Lai, and M. A. Kersten. Does aspect-oriented programming work? *Commun. ACM*, 44(10):75–77, Oct. 2001.

[34] H. Ossher and P. Tarr. Hyper/J: multi-dimensional separation of concerns for Java. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE'00)*, pages 734–737, Limerick, Ireland, June 2000. ICSE'00, IEEE Computer Society.

[35] A. Rashid and M. Aksit, editors. *Transactions on Aspect-Oriented Software Development I*, number 3880 in Lecture Notes in Computer Science. Springer Verlag, 2006.

[36] M. Shonle, K. Lieberherr, and A. Shah. XAspects: An extensible system for domain specific aspect languages. In *Companion to the 18th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 28–37, Anaheim, California, 2003. ACM Press.

[37] É. Tanter and J. Noyé. A versatile kernel for multi-language AOP. In *Proceedings of the 4th International Conference on Generative Programming and Component Engineering (GPCE'05)*, number 3676 in Lecture Notes in Computer Science, pages 173–188, Tallin, Estonia, Sept. 29-Oct. 1 2005. Springer Verlag.

[38] P. L. Tarr, H. L. Ossher, W. H. Harrison, and S. M. Sutton, Jr. N degrees of separation: Multi-dimentional separation of concerns. In ICSE'99 [18], pages 107–119.

[39] D. Walker, S. Zdancewic, and J. Ligatti. A theory of aspects. In *Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming*, pages 127–139, Uppsala, Sweden, Aug. 2003. ACM Press.

[40] R. J. Walker, E. L. A. Baniassad, and G. Murphy. Assessing aspect-oriented programming and design. In S. Demeyer and J. Bosch, editors, *Object-Oriented Technology, ECOOP'98 Workshop Reader*, number 1543 in Lecture Notes in Computer Science, pages 433–434, Brussels, Belgium, July 1998. Proceedings of the ECOOP'98 Workshops, Demos, and Posters, Springer Verlag.

[41] R. J. Walker, E. L. A. Baniassad, and G. C. Murphy. An initial assessment of aspect-oriented programming. In ICSE'99 [18], pages 120–130.

[42] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Trans. Prog. Lang. Syst.*, 26(5):890—910, Sept. 2004.