

# Identifying Feature Interactions in Multi-Language Aspect-Oriented Frameworks\*

Sergei Kojarski<sup>†</sup>     David H. Lorenz  
Department of Computer Science, University of Virginia  
Charlottesville, Virginia 22904-4740, USA  
{kojarski, lorenz}@cs.virginia.edu

## Abstract

*The simultaneous use of multiple aspect languages has the potential of becoming a significant one, as new aspect-oriented frameworks are developed and existing ones expand to incorporate features of others. A key challenge in combining multiple aspect-oriented languages is identifying and resolving adverse feature interactions. These interactions occur due to the incompatible and inconsistent treatment of aspects, join points, and advice across different languages. In this paper, we analyze the root cause of this feature interaction problem. We classify common features of aspect languages, describe how these features may interact when using different aspect languages in tandem, and concretely illustrate how these interactions may be resolved. Our work allows AOP users and tool developers to reason about the occurrence of such adverse and unexpected feature interactions, and to apply several patterns for resolving these problems.*

## 1. Introduction

The inherent goal of software design is to create sound, implementable artifacts that are also easily maintainable, quickly evolvable, and clearly readable. These characteristics emerge naturally in design artifacts that keep system concerns separate. Aspect-oriented programming (AOP) [6] provides support for separation of otherwise crosscutting concerns. As such, *aspect-oriented software engineering* has the potential of becoming an effective software design and development approach.

The available AOP languages vary in their potential effectiveness [18]. General purpose AOP languages, e.g., ASPECTJ [5] or ASPECTWERKZ [1], express imperatively a wide

range of crosscutting concerns, but do so at the price of complex aspect descriptions. In comparison, domain specific AOP languages, e.g., COOL [10], express domain concerns declaratively, but lack the expressiveness to tackle all cases of crosscutting.

The ability to concurrently use several AOP languages, whether general purpose or domain specific, can improve the overall effectiveness of AOP. Indeed, a current trend in language and tool support for AOSD [3] is to provide “multi-language” aspect-oriented frameworks which seamlessly integrate several AOP languages.<sup>1</sup> Some development environments, such as IBM’s CME, even strive to providing developers with “a common platform in which different AOSD tools can interoperate and integrate” [4].

Unfortunately, there is no methodology to facilitate the construction of multi-extension aspect-oriented frameworks. Concrete integrations are generally built ad-hoc. We examined the level of integration support and tested the quality of the integrated product in three state-of-the-art bodies of code:

- The latest version and compiler for ASPECTJ/5 [2], which is a composition of ASPECTJ 1.2 [5] and ASPECTWERKZ [1], but does not support COOL [10].
- Reflex [17], which is a multi-extension AOP kernel [16] based on an intermediate reflective representation for implementing AOP extensions and resolving aspect interactions. It includes a plugin for a subset of ASPECTJ [14], but currently not for ASPECTWERKZ nor for COOL.
- XAspects [15], which is a framework for composing aspects written in different AOP languages by translating them to ASPECTJ. It includes support for ASPECTJ and COOL, but currently not for ASPECTWERKZ.

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

<sup>†</sup>Sergei Kojarski is a PhD candidate at Northeastern University and a visiting graduate student at University of Virginia.

<sup>1</sup>The term *multi-language* is a misnomer since it really refers to multiple aspect extensions to the same base language.

We found that, from a *user perspective*, the available compositions exhibit obscure, unexpected, and even arguably incorrect behavior. We later show that Reflex handles advice erroneously (Listing 6); ASPECTJ/5 exhibits unexpected behavior [12]; and programs pre-processed in XAspects may behave incorrectly [7]. From a *tool developer perspective*, implementing such integrations in the respective frameworks is difficult, sometimes impossible. For example, the translation approach employed by Reflex and XAspects does not support a reasonable composition of ASPECTJ and COOL. Elsewhere [7] we show an example illustrating how translated aspect code might fail.

Based on these observations, we conclude that the design and development of an aspect-oriented framework that integrates several aspect extensions is an aspect-oriented software engineering challenge in and of itself. In this paper, we perform an analysis of the aspect extension composition problem. Our analysis reveals that the reason why implementing a multi-extension composition is so difficult is a fundamental *feature interaction problem*.

This paper is the first to analyze this feature interaction problem. We note that the *aspect extension composition problem* [7] is fundamentally different from the *aspect composition problem* [11]. The former concerns *extensions*, while the latter concerns *aspects*. The problem of composing extensions is the problem of defining the semantics of a new multi-extension language. In contrast, the problem of composing aspects is the problem of specifying the behavior of an AOP program under the pre-defined semantics of the AOP language.

In this work we focus on a dominant family of aspect extensions known as *join point and advice* (J&A). The J&A family is a practical space to investigate. It hosts many of the existing AOP languages, including ASPECTJ [5] ASPECTWERKZ [1], and COOL [10]. We use the following font, shape, and color convention. Code is displayed in Typewriter font, with keywords written in **Bold** series: **black** is used for Java; **blue** for COOL; **red** for ASPECTJ. Join point and advice types are written in Sans serif font; computations in *Slanted* shape.

## 2. Illustration of the Problem

Consider a composition of ASPECTJ and COOL. COOL is a domain specific aspect extension to Java for modularization of synchronization concerns. Its syntax is quite different from the syntax of ASPECTJ. COOL does not have pointcuts. Join points in COOL are implicit and are not reflected in the syntax. Advice in COOL is defined using multiple separate terms and expressions.

Aspects in COOL (called *coordinators*) synchronize Java methods. A synchronization policy for a method is expressed using several expressions. The expressions

Listing 1: BoundedStack.cool

```

1 coordinator BoundedStack {
2   condition empty = true, full=false;
3   int reads=0, writes=0;
4
5   selfex push, pop;
6   mutex {push, pop, top};
7
8   push: requires !full;
9   on_entry {writes++;}
10  on_exit {
11    empty = false;
12    if (size == capacity) full = true;
13  }
14
15  pop: requires !empty;
16  on_entry {reads++;}
17  on_exit {
18    full = false;
19    if (size == 0) empty = true;
20  }
21
22  top: requires !empty;
23  on_entry {reads++;}
24 }

```

**selfex**, **mutex**, and **requires** define pre-conditions on a method invocation. A thread may execute the method only if the pre-conditions are met. Otherwise, the thread suspends and waits on the method. The **on\_entry** and **on\_exit** expressions are executed immediately before and immediately after a method execution.

For example, BoundedStack (Listing 1) is a coordinator in COOL that advises a Java class (with the same name) whose code is not shown. The coordinator synchronizes thread access to the methods `push`, `pop`, and `top` of that class. The **selfex** declaration (line 5) specifies that neither `push` nor `pop` may be executed by more than one thread at a time. The **mutex** declaration (line 6) prohibits concurrent execution of `push`, `pop`, and `top`. Specifically, while a thread executes one of the three methods (e.g., `push`), no other thread may enter either one of the other two (e.g., `pop` or `top`). A **requires** expression specifies a pre-condition over the **condition** fields (line 2). For example, **requires !full** (line 8) guards the `push` method from being executed when the stack is full.

The **on\_entry** and **on\_exit** expressions update the aspect fields immediately before and after a method execution, respectively. The expressions may read instance variables of the coordinated Java object. For example, the **on\_exit** expression (lines 10-13) updates the `empty` and `full` coordinator variables, and reads from the `size` and `capacity` fields of the coordinated BoundedStack Java object.

The problem of composing ASPECTJ and COOL raises many questions including, e.g.:

- What are the join points in COOL, and how do they correspond to join points in ASPECTJ?
- Should the execution of an advice in COOL generate an advice execution join point in ASPECTJ?
- Should aspects in ASPECTJ be permitted to advise the expressions `requires`, `on_exit`, and `on_entry`, and if so, what precisely are the rules of engagement?
- Should the execution of `mutex` and `selfex` preconditions be advisable by aspects in ASPECTJ, and if so, what join points do such executions generate?
- Should COOL synchronize advice in ASPECTJ that apply to the same method, or just synchronize the body of the method?

There is no simple way to identify and answer these questions for the composition of COOL and ASPECTJ. Even if one knows how to put COOL and ASPECTJ together, it is difficult to evaluate whether the composition behaves as expected.

### 3. Analysis of the Problem

From a feature interaction perspective, an aspect extension introduces features that incrementally extend the base language specification. The problem of specifying the behavior of a multi-extension composition is the problem of identifying and resolving the interactions between features of the composed extensions.

Each aspect extension to the base language is specified in the context of a single-extension language. In a multi-extension language, however, a program contains aspects written in various aspect extensions. Aspects written in one extension generally advise not only code written in that extension and in the base language, but also code written in the foreign extensions. Moreover, aspects written in different extensions may collaboratively affect the same point in the execution (join point). These behaviors are not described in the specification of any of the composed extensions, but rather emerge in their composition.

Our focus in this paper is on the family of extensions whose high-level behavior evolves around join points and advice (J&A). We identify the *abstract features* that are common to all J&A extensions [9], and we refer to the feature model (the set of abstract features) as an *abstract extension*. A virtual composition of such abstract extensions provides us with a conceptual framework for reasoning about concrete compositions of concrete extensions. We refer to the interactions between abstract features in a virtual composition as *feature interaction patterns*.

We distinguish between three related definitions commonly associated with the term *join point*. A join point

*computation* is a “point” in the execution. A join point *type* is an extension-specific data structure for describing a computation. A join point *instance* is an internal description of a particular computation.

#### 3.1. Join Point Features

We identify two groups of interaction-related abstract features. The first group includes *join point features* that characterize the ability to *observe* a program execution. The features are: granularity, type, visibility, and history.

**Join Point Granularity.** The join point granularity feature specifies what kinds of computations may be intercepted by the extension. The granularity is the consolidation of base language computations (base granularity) and extension-specific computations (aspect granularity). In ASPECTJ, for example, the base granularity includes certain computations in Java (*method call*, *method execution*, etc.); the aspect granularity includes only *advice execution* computations. In COOL, the base granularity includes only *method invocation* computations in Java; and the aspect granularity is empty.

**Join Point Type.** The join point type feature defines the extension-specific data structure for describing computations. In ASPECTJ this structure includes the kind of computation (e.g., *method call*) and its dynamic, static, and lexical contexts (e.g., `this` and `target` objects, signature of the target member, and location in the source file). In COOL, the description contains only the signature of the target method.

**Join Point Visibility.** The join point visibility feature maps join point computations to join point instances. The feature classifies computations as either *visible* or *invisible*; and instantiates join points for the visible ones. The join point visibility feature of ASPECTJ, for example, filters out all the computations within the lexical scope of an `if` pointcut expression; and constructs join points for all the rest. All *method call*, *get*, and *set* computations within the `if` pointcut expression are invisible, although subcomputations in their control flow (e.g., executions of methods that are called from the pointcut expression) are visible.

**Join Point History and Genealogy.** The join point history and genealogy feature stores join point instances, and establishes their relation to provide a genealogy of join points. Intuitively, the feature builds an extension-specific representation of the program execution. For example, ASPECTJ represents a program execution as a join point stack. The stack defines a *dynamic context* relation over

join points: join points pushed on the top of the stack are said to be in the dynamic context of those below them. An *enclosing join point* relation associates each join point with its immediate parent on the stack.

### 3.2. Advice Features

The second group includes *advice features* that shape the capability to *modify* a program. The features are: advice type, join point advisability, advice ordering, and advice execution.

**Advice Type.** The advice type feature defines the sorts of advice weaving that an aspect extension supports. For example, ASPECTJ and ASPECTWERKZ support *before*, *after*, and *around* advice. Identifying the advice types in COOL, on the other hand, is more of a challenge. COOL defines two blocks of operations. The first block is weaved before a method invocation. It comprises operations defined by *mutex*, *selfex*, *requires*, and *on\_entry* expressions. We refer to this as the *lock* block. The second block is weaved after a method invocation and includes *on\_exit* expressions. We refer to this as the *unlock* block. The semantics of COOL specify that both the *lock* and *unlock* blocks are synchronized. A synchronized block is never executed concurrently with another synchronized block. Therefore, it is reasonable to identify *lock* and *unlock* as advice types in COOL.

**Join Point Advisability.** The join point advisability feature associates advice with a join point. The association logic is normally based on the join point history. For example, in ASPECTJ an advice is selected if its pointcut matches the join point stack. Besides the core advice selection logic, the feature may define advising constraints for various types of join points, or even for specific join points. For example, the join point advisability feature of ASPECTJ restricts the advisability of handler join points to *before* advice only; and filters *around* advice at initialization and preinitialization join points.

The join point visibility and join point advisability features are not interchangeable. In general, filtering a join point is not equivalent to filtering all the advice for that join point. In the first case, the join point imposes no effect on the program execution. In the second case, the join point is reflected in the join point history, and can potentially affect the execution at a future join point. For example, if ASPECTJ were to filter join points within *if* pointcut expressions by disallowing advice, then these join points could still be accessed via a *cflow* pointcut designator.

**Advice Ordering.** The advice ordering feature prescribes the semantics for sorting advice that is selected at a join

point into a specific execution order. For example, ASPECTJ orders pieces of advice according to their type, their lexical location in the aspect definition, and a precedence relationship over aspects.

**Advice Execution.** The advice execution feature controls how an extension observes the execution of its own advice. Specifically, the feature determines how advice computations are built. Once built, the computations can be recursively intercepted and transformed by the extension, thus allowing aspects to advise each other. For example, the advice execution feature of ASPECTJ executes an advice as an *advice execution* computation. The computation is then intercepted by the join point visibility feature and advised by ASPECTJ as an advice execution join point. On the other hand, the advice execution feature of COOL is empty, because a COOL coordinator can only advise Java methods.

## 4. Patterns of Interaction

We analyze the feature interaction problem by focusing on a composition of abstract extensions. Our analysis yields seven interaction patterns. We present each pattern by listing the interacting abstract features and discussing how the interaction can be resolved.

**Emergent Join Point Granularity and Type.** The emergent join point granularity feature normalizes and consolidates the individual granularities to enable a common reference for mutual interaction. Unfortunately, this process is generally ambiguous. The problem occurs because the join point granularity features of the individual extensions are given in different terms and at different levels of detail; and they may be normalized in more than one way. This ambiguity must be resolved in the composition.

The ASPECTJ granularity feature includes *method execution* and *method call* computations. The COOL granularity feature is less detailed and only includes *method invocation* computations. Thus, there are at least three ways to normalize the granularities of COOL and ASPECTJ. A *method invocation* can be equated with a *method call*, or a *method execution*, or it can be mapped to a computation between the call and the execution (e.g., nested within a *method dispatch* computation, but around a *method execution* computation).

Each of the alternative normalizations specifies a unique composition behavior. Equating *method invocation* with *method call* join point computations forces coordinators in COOL to synchronize ASPECTJ advice at *method execution* join points. It also allows the composition to choose whether or not the coordinators synchronize ASPECTJ advice at *method call* join points (by ordering the advice of COOL and ASPECTJ). In contrast, equating *method invo-*

ation with *method execution* join point computations prevents the COOL coordinators from synchronizing ASPECTJ advice at method call join points, and allows the composition to choose whether or not the coordinators synchronize ASPECTJ advice at method execution join points. Finally, mapping *method invocation* to a computation in-between ASPECTJ's *method call* and *method execution* prevents COOL from synchronizing ASPECTJ advice at method call join points, and forces the coordinators in COOL to synchronize ASPECTJ advice at method execution join points.

**Advice Execution Interaction.** An aspect extension introduces extension-specific terms and expressions. In a multi-extension AOP language, these extension-specific terms and expressions can be advised by foreign aspects. However, it is unspecified how these terms and expressions are observed by the foreign extensions.

Consider deploying the `BoundedStack` (Listing 1) coordinator in a multi-extension composition of COOL and ASPECTJ. An execution of the coordinator locks and unlocks a target Java method, reads and writes to the `empty`, `full`, `reads`, and `writes` instance variables of the coordinator, and reads from the `size` and `capacity` fields of the coordinated `BoundedStack` Java object. In COOL these operations are *not* join point computations because the granularity of COOL is limited to *method invocations* only. The composition of COOL and ASPECTJ has a finer join point granularity that might intersect with COOL coordinators. This poses the question: what join point computations in the emergent granularity does the `BoundedStack` coordinator contain?

The problem can be attributed to the interaction between the advice execution feature of an aspect extension and the emergent granularity feature of the multi-extension composition. The advice execution feature controls how the extension observes execution of its own aspects in the extension's granularity. In a multi-extension composition, the aspect can generally be advised by foreign extensions at the *emergent* granularity level. Because the emergent granularity is generally finer than the extension's granularity, it is undefined what join point computations in the emergent granularity the aspect contains.

To resolve this interaction, a composition designer must refine the advice execution feature to build join point computations in the emergent granularity domain. In general, however, the feature can be refined in multiple alternative ways. One point of variability is the inherent difference between base language and aspect language terms and concepts. In our example, the composition designer should decide whether COOL coordinators behave like Java classes; whether static initializations of the coordinator classes generate *static initialization* computations; whether instantiations of the coordinator objects generate *preini-*

Listing 2: `AJAdviceLogger.java`

```
1 public aspect AJAdviceLogger {
2   before(): adviceexecution()
3     && !cflow(within(AJAdviceLogger)) {
4     System.out.println("Advice: "+thisJoinPoint);
5   }
6 }
```

Listing 3: `AJAccessLogger.java`

```
1 public aspect AJAccessLogger {
2   before(): (get(* *.* ) || set(* *.* ))
3     && !within(AJAccessLogger) {
4     System.out.println("Access: "+thisJoinPoint);
5   }
6 }
```

*tialization*, *constructor execution*, and *initialization* computations; whether COOL `condition` fields can be treated as `boolean` Java fields; what join point computations are found in `on_entry`, `on_exit`, and `requires` expressions; and whether or not other COOL constructs (e.g., `mutex`) produce Java join point computations.

Another point of variability is dealing with join point computations that are specific to foreign extensions (e.g., *advice execution*). Consider deploying the `BoundedStack` coordinator together with the `AJAdviceLogger` (Listing 2) aspect. The `AJAdviceLogger` aspect logs all advice execution join points in the program. The composition specification should define how, if at all, `AJAdviceLogger` logs executions of the `BoundedStack` coordinator.

One option is to execute advice in COOL as *advice execution* computations. Another option is to evaluate only `on_entry` and `on_exit` as *advice execution* computations. Yet another option is to have no *advice execution* computations within COOL coordinators whatsoever.

**Join Point Visibility Interaction.** The join point visibility feature of an aspect extension maps join point computations to extension-specific join point instances. In a multi-extension composition the mapping is undefined over join point computations that are produced by foreign aspects.

Consider now the deployment of `BoundedStack` (Listing 1) together with `AJAccessLogger` (Listing 3) in a composition of COOL and ASPECTJ. `BoundedStack` accesses the fields of the coordinator and the coordinated object, and `AJAccessLogger` logs access to instance variables. Even if we assume that the advice execution feature of COOL is defined to generate field access (`get` and `set`) join point computations in executions of `requires`, `on_entry`, and `on_exit` expressions, it is still unspecified what join points `AJAccessLogger` logs in the `BoundedStack` coordinator.

Listing 4: AJStackExecScope.java

```

1 public aspect AJStackExecScope {
2   before():cflow(execution(* BoundedStack.*(..)))
3     && !cflow(within(AJStackExecScope)) {
4     System.out.println("Cflow: "+thisJoinPoint);
5   }
6 }

```

Listing 5: AJStackLogger.java

```

1 public aspect AJStackLogger {
2   before():execution(* BoundedStack.*(..)) {
3     System.out.println("Method: "+thisJoinPoint);
4   }
5 }

```

The problem can be attributed to the interaction between the join point visibility feature of ASPECTJ and the advice execution feature of COOL. The interaction is resolved by extending the join point visibility feature to classify and instantiate join points in coordinators. In general, the feature can be extended in many possible ways. In our example, the join point visibility feature can be defined to construct ASPECTJ join points for all join point computations in COOL coordinators; it can be defined to ignore *get* and *set* computations that access the `condition` fields of a coordinator; or it can be defined to ignore all COOL computations. In the first case, `AJAccessLogger` would log any *set* and *get* field access within the `BoundedStack` coordinator. In the second case, the aspect would not log *get* and *set* access to the `full` and `empty` fields. In the third case, the aspect would not advise the coordinator at all.

**Join Point History Interaction.** In a single extension, the join point history and genealogy feature establishes and maintains a relation over the join points found in the aspect and base programs. In a multi-extension composition the feature is unspecified over join points that the extension observes in foreign aspects. We call these *foreign join points*.

Consider the `AJStackExecScope` aspect (Listing 4). The *dynamic context* relation in ASPECTJ provides the semantics to the `cflow` pointcut. `AJStackExecScope`'s `before` advice logs all join points in the dynamic context (control flow) of the execution of a `BoundedStack` method (except for the join points in the control flow of the advice).<sup>2</sup>

Next, consider the `AJStackLogger` aspect (Listing 5). In ASPECTJ, advice is executed in the dynamic context of the join point it advises. `AJStackLogger` advises executions of `BoundedStack` methods. If `AJStackExecScope` and `AJStackLogger` are deployed together then the former will always advise join points inside the latter.

<sup>2</sup>Join points in the control flow of the advice are also in the control flow of the method execution, but they are excluded to prevent an infinite loop.

In a composition of ASPECTJ and COOL, however, the dynamic context relation of ASPECTJ is unspecified over join points found in coordinators. Let assume that aspects advise *get* and *set* join points in coordinators. The `BoundedStack` (Listing 1) coordinator runs COOL expressions on every execution of *pop*, *push*, and *top*. When the coordinator and the `AJStackExecScope` aspect are deployed simultaneously, the join point history and genealogy feature of ASPECTJ can treat the `BoundedStack` coordinator in one of several ways.

One option is to run the `requires`, `on_entry`, and `on_exit` expressions in the dynamic context of ASPECTJ's method execution join points. In that case, `AJStackExecScope` logs join points in the `BoundedStack` aspect. An alternative is to run the expressions outside the dynamic context of the method execution join points. In that case, the `AJStackExecScope` aspect does not advise the `BoundedStack` coordinator.

**Join Point Advisability Interaction.** The join point advisability feature selects advice at a join point. In a composition of multiple extensions, however, the feature is undefined over foreign join points.

The construction of foreign join points is resolved in the join point visibility interaction. The join point advisability interaction is therefore an interaction between the join point advisability and join point visibility features of the aspect extension. The interaction is resolved by extending the join point advisability feature to select advice at the foreign join points.

For example, assume that the join point visibility feature of ASPECTJ constructs *get* and *set* join points for every access to any field of a COOL coordinator or a coordinated object (from within the coordinator). The interaction between the join point advisability and join point visibility features of ASPECTJ then raises the question of how ASPECTJ advises these join points.

The interaction can be resolved in at least three reasonable ways. First, ASPECTJ may treat these join points as regular field access without constraining the kind of advice. A second option is to allow ASPECTJ aspects to advise only a field access that targets a field of a coordinated object (e.g., access to `size` and `capacity`), while hiding any access to an internal field of a coordinator (e.g., `empty`, `full`, `reads`, `writes`). This solution respects the differences between COOL and Java objects, while allowing ASPECTJ aspects to advise any access to a Java field. A third option is to allow ASPECTJ aspects to advise an access to a field of a coordinator with `before` advice only. In this manner, the internal synchronization logic of COOL coordinators cannot be overridden by ASPECTJ, while ASPECTJ aspects can still monitor the coordinator.

**Emergent Advice Type.** The advice type feature of the composition normalizes the set of advice types defined by the individual extensions. However, the normalization of advice types is generally ambiguous. Different normalizations yield different composition specifications. This ambiguity constitutes the unspecified behavior found in the composition.

The normalization of COOL and ASPECTJ advice types is a good example. The first challenge is to identify correctly the COOL advice types (recall Section 3.2). Once the advice types are identified, the problem of normalizing advice types in the composition of COOL and ASPECTJ is the problem of matching the lock and unlock advice types of COOL with the before, after, and around advice types of ASPECTJ. One option is to equate lock and unlock with before and after, respectively. Another option is to equate a pair of lock and unlock advice blocks with a single around advice. A third option is to consider COOL and ASPECTJ advice types distinct, and allow COOL advice to dominate over ASPECTJ advice (i.e., to always run lock advice before ASPECTJ's advice and unlock advice after ASPECTJ's advice). This ambiguity illustrates the general problem of advice type feature interaction in a composition of multiple aspect extensions.

**Emergent Advice Ordering.** A single join point computation can generally match advice written in several extensions. The advice ordering feature of the composition (not to be confused with the advice ordering feature of an individual extension) defines the ordering of selected multi-extension advice.

The ordering of multi-extension advice is unknown at the level of the individual extensions. Rather, this feature emerges in the combination of the various advice ordering features and is thus unspecified.

Consider a composition of ASPECTJ and ASPECTWERKZ. For this composition, the advice ordering feature interaction problem has many alternative solutions that exhibit considerably different behavior. For example, one option is (a) to execute before advice of both extensions first; then to wrap their around advice over each other; and finally execute the remaining after advice. Another option is (b) to run ASPECTWERKZ's advice only when ASPECTJ proceeds to the base program. In this case, ASPECTWERKZ advice are nested within the execution of ASPECTJ around advice.

Adding an extension to the composition generally increases a number of alternative solutions. For example, consider a composition of ASPECTJ, ASPECTWERKZ and COOL. Assuming that COOL method invocation join points are equated with method execution join points, and COOL lock and unlock advice types are equated with before and after advice types of ASPECTJ/ASPECTWERKZ, each ordering of ASPECTJ and ASPECTWERKZ advice corresponds to multiple combinations of ASPECTJ, ASPECTWERKZ, and COOL

advice. For example, if ASPECTJ and ASPECTWERKZ advice are ordered as suggested by option (a), the lock and unlock advice may be ordered to (i) synchronize all ASPECTJ/ASPECTWERKZ advice; (ii) synchronize only around advice; (iii) synchronize only Java methods, and so on.

## 5. Evaluation

In this section we evaluate our analysis and conceptual framework by using the feature interaction patterns to identify and resolve adverse feature interactions in the composition of ASPECTJ, ASPECTWERKZ and COOL. We then use the patterns to identify and explain the unexpected behavior we observed in ASPECTJ/5, Reflex, and XAspects.

### 5.1. Specifying New Compositions

Consider a composition of three extensions: ASPECTJ, ASPECTWERKZ and COOL. For this composition, the feature interaction patterns generate a set of 24 interactions to be analyzed. Concretely, the three emergent patterns generate one feature interaction each, the advice execution pattern yields three interactions, and each of the other three patterns generates six feature interactions, e.g., interactions between the join point visibility feature of COOL/ASPECTJ/ASPECTWERKZ and the advice execution features of ASPECTJ and ASPECTWERKZ (COOL and ASPECTWERKZ; ASPECTJ and COOL). The feature interactions pose questions, including:

- What join point computations are observed by ASPECTJ (ASPECTWERKZ) within the coordinators?
- How does ASPECTJ (ASPECTWERKZ) define the dynamic context relation over coordinator's join points?
- How does ASPECTJ (ASPECTWERKZ) advise join points that originate from the coordinators?

Although the questions are initially expressed in abstract terms, they are easily restated in concrete terms of COOL, ASPECTJ, and ASPECTWERKZ. Each question expands into several extension-specific questions. Due to space consideration, we list and answer only the most interesting ones:

**What is the granularity of the composition?** The composition has the same granularity as ASPECTJ. Corresponding join point types of ASPECTJ and ASPECTWERKZ are equated. *Method invocation* computations of COOL are equated with *method execution* computations of ASPECTJ.

**What join point computations are specified by a COOL coordinator?** The coordinator advises a target method

with lock and unlock advice. An execution of a COOL advice yields an *advice execution* computation. Within the advice, COOL exposes *get* and *set* computations as defined by *requires*, *on\_entry*, and *on\_exit* expressions.

**What join points does ASPECTJ (ASPECTWERKZ) instantiate for a coordinator’s computation?** An *advice execution* computation is mapped to an advice execution join point. A *get* (*set*) computation is represented by a *get* (*set*) join point.

**How does ASPECTJ (ASPECTWERKZ) advise COOL join points?** ASPECTJ (ASPECTWERKZ) may only impose *before* advice over *set* and *get* join points that target fields of the coordinator object. This decision protects the internal thread synchronization logic of the coordinator, while allowing ASPECTJ (ASPECTWERKZ) aspects to inspect its execution. An access to a field of the coordinated object is advised in the usual way. The advice execution join points (that represent executions of lock and unlock advice) can be advised by *before* and *after* advice only. This decision protects COOL advice from being overridden by ASPECTJ (ASPECTWERKZ) aspects.

**What join points are observed by COOL in ASPECTJ (ASPECTWERKZ) aspects?** COOL treats the ASPECTJ (ASPECTWERKZ) aspects as Java classes. It builds method invocation join points for *method execution* computations that originate from the aspects. Moreover, implicit executions of ASPECTWERKZ advice or pointcut methods are also observed by COOL as method invocations.

**What join points are observed by ASPECTJ aspects in ASPECTWERKZ aspects?** In ASPECTWERKZ an advice is an annotated method with a dual purpose (and a pointcut can also be defined using a method). The method may be executed implicitly when playing the role of an advice (or a pointcut). It may also be invoked explicitly as a regular Java method. Implicit executions of the advice methods of ASPECTWERKZ are exposed as advice execution join points in ASPECTJ. Implicit executions of the pointcut methods are hidden from ASPECTJ, including join points that lexically reside within the pointcut methods. Explicit executions of the pointcut and advice methods are treated as regular Java method executions.

The rest of the feature interactions between ASPECTJ, ASPECTWERKZ, and COOL are omitted. A complete list of questions and answers fully specifies the composition. The specification we have presented defines only one possible composition of ASPECTJ, ASPECTWERKZ and COOL.

## 5.2. Analyzing Existing Compositions

Our analysis is also useful for identifying and explaining adverse feature interactions in existing compositions of aspect extensions. Applying the analysis to ASPECTJ/5, Reflex, and XAspects reveals that they fail to adequately resolve feature interactions in multi-extension compositions. All three systems exhibit an unexpected behavior of aspects “misadvising” foreign aspects. ASPECTJ/5 fails to address a particular instance of the advice execution feature interaction; and Reflex and XAspects fail to address the feature interaction problem in general.

**ASPECTJ/5.** ASPECTJ/5 [2] is a merger of ASPECTJ and ASPECTWERKZ. ASPECTJ and ASPECTWERKZ are very similar semantically, diverging mainly at the syntactical level. ASPECTJ differentiates syntactically between a method, an advice, and a pointcut; whereas ASPECTWERKZ does not.

We tested ASPECTJ/5 using a set of ASPECTJ and ASPECTWERKZ aspects that address all the interactions that are generated from the interaction patterns. Running these aspects in ASPECTJ/5 revealed that:

- Explicit executions of either pointcut or advice methods in ASPECTWERKZ aspects do not generate method execution join points;
- Advice execution join points are generated for both implicit and explicit executions of the advice methods;
- No join points are generated within the body of a pointcut method, even if it is executed explicitly.

We attribute this behavior of ASPECTJ/5 to a failure to identify and address the interactions between the advice execution feature of ASPECTWERKZ and the emergent granularity feature of ASPECTJ/5. In our minds, ASPECTJ/5 generates unexpected advice execution join points without any actual advice execution; unexpectedly hides method execution join points at explicit executions of pointcut and advice methods; and unexpectedly hides all join points within pointcut methods at their explicit execution.

**Reflex.** Tanter *et al.* [16] present a framework for resolving aspect interactions. The Reflex framework is also used to address the aspect extension composition problem, by reducing the problem of composing multi-extension aspects to the problem of composing multiple aspects in Reflex. The framework applies a translation approach: it takes as input an aspect program written in a given language and outputs an aspect program in Reflex [17].

A designer of a multi-extension composition can use Reflex to specify composition rules that resolve *aspect-level* interactions in the translated program. Concretely, the designer can resolve “aspect interactions [that] occur when

Listing 6: AJNothing.java

```

1 aspect AJNothing {
2   before():execution(* *(..)) {}
3 }

```

several aspects affect the same program point” [17], namely ordering and nesting of aspects, mutual exclusion (an aspect should not apply whenever another aspect applies), and implicit cut (an aspect should apply whenever another aspect applies).

However, while the framework provides adequate support for resolving *aspect-level* interactions in a specific Reflex program, it does not provide any means for resolving *extension-level* interactions in the multi-extension AOP language. Most notably, the lack of support for resolving advice execution interactions renders the Reflex framework inapplicable for composing aspect extensions in general.

The source of the problem is that Reflex fills in the unspecified behavior in an ad-hoc manner. In particular, the translated aspects in Reflex may misadvise each other. When the framework translates an aspect from a source extension to Reflex, it introduces implementation-specific operations into the target Reflex aspect (realized in part as a metaobject class). The implementation-specific operations are not explicit in the code of the source aspect; they realize behavior of terms, expressions, and concepts that are specific to the source extension. For example, means of synchronizing threads that are implicit in a COOL coordinator might be translated into explicit `wait` and `notifyAll` method calls in a target Reflex aspect.

The implementation-specific operations introduce unexpected join points into a translated Reflex program. The program aspects cannot distinguish the bad (unexpected) from the good (expected) join points, and can erroneously advise the bad points. Advice executions at bad join points are not only unexpected to an aspect programmer, but may also render the program incorrect. For example, the empty advice in the `AJNothing` aspect (Listing 6) is supposed to do nothing but, when run in Reflex, results in an infinite loop. The source of the problem is that Reflex translates the `before` advice to a method in the target metaobject class. The aspect then erroneously advises executions of this method as method execution join points. This simple failure illustrates an inability of Reflex to deal with complex extension-level interactions in a multi-extension composition.

**XAspects.** Shonle *et al.* [15] present a framework for compiling aspects written in multiple domain-specific aspect extensions. XAspects uses a translation approach. It reduces all extensions to ASPECTJ. Given a set of programs

written in different extensions, XAspects produces a single program in ASPECTJ.

Similar to Reflex, the XAspects framework does not provide adequate support for resolving feature interactions in the multi-extension AOP language. We analyzed the support for COOL and ASPECTJ in XAspects. We found that the major problem of the composition lies in a failure to resolve the interaction between the advice execution feature of COOL and the emergent granularity feature of the composition. Because of that interaction, COOL coordinators expose unexpected join point computations that can be erroneously advised by ASPECTJ aspects. In some multi-extension programs this interaction causes an unexpected deadlock [7]. Most of the other interactions are also not addressed in XAspects.

## 6. Discussion and Other Related Work

As we have covered ASPECTJ/5, Reflex, and XAspects, we complete the discussion of related work with two studies that focus on the problem of composing aspect extensions and on the problem of composing aspects.

**Disciplined Aspect Extension Composition.** Pluggable AOP [7] is a study of the problem of undefined semantics for aspect extension compositions. The work presents a framework for third-party composition of arbitrary dynamic aspect mechanisms into an AOP interpreter. In the Pluggable AOP framework, the aspect mechanisms collaborate by hiding, delegating, and exposing expression evaluations.

Unlike other composition frameworks, Pluggable AOP resolves advice execution interactions in the composition. Specifically, the framework establishes a common granularity for the multi-extension AOP language, and provides design guidelines for resolving the advice execution interactions [13]. The framework, however, provides very restricted support for resolving advice ordering (other than ordering the mechanisms), and does not address other interactions [8].

In contrast, this paper studies feature interactions that occur in the composition. The results we present here are relevant for any composition of aspect extensions, not just for third-party compositions. Specifying the composition semantics is inherently complex. Our analysis simplifies some of that complexity by identifying what must be specified.

**Disciplined Aspect Composition.** Lopez-Herrejon *et al.* [11] study the problem of undefined semantics for aspect compositions in ASPECTJ. Although the problem of undefined semantics in a single-extension AOP language is orthogonal to our study, their algebraic model can be used to specify multi-extension aspect composition more flexibly.

## 7. Conclusion

Our work analyzes the feature interaction problem in the composition of aspect extensions. This problem is the reason why the task of composing multiple extensions is so complex, error prone, and unintuitive. Failing to address feature interactions normally results in unexpected behavior in the multi-language aspect-oriented framework.

One contribution of this paper is a road map for resolving the feature interaction problem in multi-language aspect-oriented frameworks. We analyze existing aspect extensions and derive a set of abstract features and their patterns of interaction. The value of our analysis is in providing users and tool developers with an appropriate abstraction for identifying feature interactions in a large set of compositions.

Our analysis is useful for specifying the behavior of new compositions of existing aspect extensions. We identify and resolve potential feature interaction problems in a hypothetical composition of ASPECTJ, ASPECTWERKZ, and COOL.

Another contribution of this paper is the understanding of the space of compositions. Our analysis is useful for verifying that feature interactions in existing frameworks are resolved properly. We identify and illustrate feature interaction problems in ASPECTJ/5, Reflex, and XAspects.

Finally, the set of features we identify contributes a vocabulary for documenting, understanding, and communicating designs of multi-language AOP frameworks.

**Acknowledgment** We thank Kim Hazelwood and the anonymous reviewers for their helpful comments.

## References

- [1] J. Bonér. What are the key issues for commercial AOP use: how does AspectWerkz address them? In *Proceedings of the 3<sup>rd</sup> International Conference on Aspect-Oriented Software Development (AOSD'04)*, pages 5–6, Manchester, UK, Mar. 2004. ACM Press.
- [2] A. Colyer. AOP@Work: Introducing AspectJ 5. *developer-Works*, July 2005.
- [3] R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2005.
- [4] IBM's concern manipulation environment, 2004. <http://www.research.ibm.com/cme>.
- [5] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of the 15<sup>th</sup> European Conference on Object-Oriented Programming (ECOOP'01)*, number 2072 in Lecture Notes in Computer Science, pages 327–353, Budapest, Hungary, June 2001. Springer Verlag.
- [6] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the 11<sup>th</sup> European Conference on Object-Oriented Programming (ECOOP'97)*, number 1241 in Lecture Notes in Computer Science, pages 220–242, Jyväskylä, Finland, June 1997. Springer Verlag.
- [7] S. Kojarski and D. H. Lorenz. Pluggable AOP: Designing aspect mechanisms for third-party composition. In *Proceedings of the 20<sup>th</sup> Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'05)*, pages 247–263, San Diego, CA, USA, Oct. 2005. ACM Press.
- [8] S. Kojarski and D. H. Lorenz. Comparing white-box, black-box, and glass-box composition of aspect mechanisms. In *Proceedings of the 9<sup>th</sup> International Conference on Software Reuse (ICSR9)*, number 4039 in Lecture Notes in Computer Science, pages 246–259, Torino, Italy, June 2006. Springer Verlag.
- [9] S. Kojarski and D. H. Lorenz. Modeling aspect mechanisms: A top-down approach. In *Proceedings of the 28<sup>th</sup> International Conference on Software Engineering (ICSE'06)*, pages 212–221, Shanghai, China, May 2006. ACM Press.
- [10] C. V. Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, Northeastern University, 1997.
- [11] R. Lopez-Herrejon, D. Batory, and C. Lengauer. A disciplined approach to aspect composition. In *Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation (PEPM'06)*, pages 68–77, Charleston, South Carolina, 2006. ACM Press.
- [12] D. H. Lorenz and S. Kojarski. Feature interaction in AspectJ/5. In *AOSD 2006 Workshop on Software-engineering Properties of Languages for Aspect Technologies*, Bonn, Germany, Mar. 2006.
- [13] D. H. Lorenz and S. Kojarski. Parallel composition of aspect mechanisms: Design and evaluation. In *AOSD 2006 Workshop on Open and Dynamic Aspect Languages*, Bonn, Germany, Mar. 2006.
- [14] L. Rodríguez, É. Tanter, and J. Noyé. Supporting dynamic crosscutting with partial behavioral reflection: a case study. In *Proceedings of the XXIV International Conference of the Chilean Computer Science Society (SCCC 2004)*, Arica, Chile, Nov. 2004. IEEE Computer Society Press.
- [15] M. Shonle, K. Lieberherr, and A. Shah. XAspects: An extensible system for domain specific aspect languages. In *Companion to the 18<sup>th</sup> Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 28–37, Anaheim, California, 2003. ACM Press.
- [16] É. Tanter. *From Metaobject Protocols to Versatile Kernels for Aspect-Oriented Programming*. PhD thesis, University of Nantes and University of Chile, Nov. 2004.
- [17] É. Tanter and J. Noyé. A versatile kernel for multi-language AOP. In *Proceedings of the 4<sup>th</sup> International Conference on Generative Programming and Component Engineering (GPCE'05)*, number 3676 in Lecture Notes in Computer Science, pages 173–188, Tallin, Estonia, Sept.-Oct. 2005. Springer Verlag.
- [18] R. J. Walker, E. L. A. Baniassad, and G. C. Murphy. An initial assessment of aspect-oriented programming. In *Proceedings of the 21<sup>st</sup> International Conference on Software Engineering (ICSE'99)*, pages 120–130, Los Angeles, California, May 1999. IEEE Computer Society.