Technical Report

Nr. TUD-CS-2011-0272
October 3rd, 2011

Modular Reasoning with Join Point Interfaces

Authors
Milton Inostroza (Universidad de Chile)
Éric Tanter (Universidad de Chile)
Eric Bodden (CASED)

Modular Reasoning with Join Point Interfaces

Milton Inostroza Éric Tanter
PLEIAD Laboratory
Computer Science Department (DCC)
University of Chile – Santiago, Chile
{minostro,etanter}@dcc.uchile.cl

Eric Bodden
Center for Advanced
Security Research Darmstadt
Darmstadt, Germany
eric.bodden@cased.de

***Abstract*—In current aspect-oriented systems, aspects usually carry, through their pointcuts, explicit references to the base code. Those references are fragile and give up important software engineering properties such as modular reasoning and independent evolution of aspects and base code. A well-studied solution to this problem consists in separating base code and aspects using an intermediate interface abstraction.**

In this work, we show that previous approaches fail at restoring modular reasoning because they do not provide modular type checking; programs can fail to compose when woven, even though their interfaces are compatible. We introduce a novel abstraction called Join Point Interfaces, which, by design, supports modular reasoning and independent evolution by providing a modular type-checking algorithm. Join point interfaces further offer polymorphic dispatch on join points, with an advice-dispatch semantics akin to multi-methods. As we show, our semantics solves important problems present in previous approaches to advice dispatch.

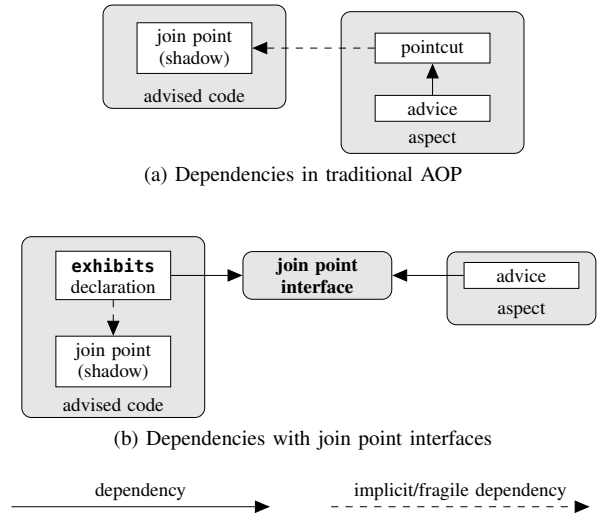
We have fully implemented join point interfaces as an open-source extension to the AspectBench Compiler. A study on existing aspect-oriented programs of varying sizes and domains supports our major design choices and reveals potential for exploiting polymorphism through non-trivial join-point type hierarchies.

***Keywords*-Aspect-oriented programming, modular reasoning, independent evolution, polymorphism**

I. Introduction

> "Modular reasoning means being able to make decisions about a module while looking only at its implementation, its interface and the interfaces of modules referenced in its implementation or interface. For example, the type-correctness of a method can be judged by looking at its implementation, its signature (i.e. interface), and the types (i.e. interfaces) of any other code called by the method." [1]

While Aspect-Oriented Programming (AOP) [2] aids in obtaining localized implementations of crosscutting concerns, its impact on modular reasoning is not that positive. Indeed, the emblematic mechanism of AOP is pointcuts and advice, where *pointcuts* are predicates that denote *join points* in the execution of a program where *advice* is executed. With such an implicit invocation mechanism, it is not usually possible to reason about an aspect or an advised module in isolation. As we show in Figure 1a, an aspect contains direct textual references to the base code via its pointcuts—with detrimental effects. These dependencies make programs



(a) Dependencies in traditional AOP

(b) Dependencies with join point interfaces

Figure 1: Dependencies in traditional AOP and with join point interfaces.

fragile, they hinder aspect evolution and reuse. Changes in the base code can unwittingly render aspects ineffective or cause spurious advice applications. Conversely, a change in a pointcut definition may cause parts of the base program to be advised without notice, breaking some implicit assumptions. The fact that independent evolution is compromised is particularly worrying considering that programming aspects requires a higher level of expertise, and is hence likely to be done by specialized programmers. Therefore, to be widely adopted, AOP is in great need of mechanisms to support separate development in a well-defined manner.

The above issues have been identified early on [3] and have triggered a rich discussion in the community [1], [4]. In particular, several proposals have been developed to enhance the potential for modular reasoning by introducing a notion of *interface* between aspects and advised code (e.g. [3], [5], [6], [7]). However, while they do enhance the situation over traditional AOP, none of these proposals manages to fully support independent evolution through modular type checking, mostly because the interfaces are not expressive enough. This is especially troublesome because the existence

of a concrete modular type checker is generally considered the first solid evidence of modular reasoning.

In order to enable fully modular type checking, we introduce *join point interfaces* (JPIs), an extension and refinement of the notion of join point types recently introduced by Steimann et al. [7]. Join point interfaces are contracts between aspects and advised code (Figure 1b). JPIs support a programming methodology where aspects only specify the types of join points they advise based on a JPI, not on concrete pointcuts. It is the responsibility of the programmer maintaining the advised code to specify, through an **exhibits** clause, which join points are of which type. Aspects and advised code can be developed and evolved independently.

Conversely to previous work [5], [6], [7], JPIs do allow for strict separate compilation thanks to modular type checking. When programmers in charge of advised code compile their module, they need to include JPI definitions but no aspect code. Likewise, when aspect experts compile their aspects, they only include the join point interface definitions but no base code. This is similar to what Java interfaces offer to support independent evolution of object-oriented code. The static semantics of JPIs gives the strong guarantee that programmers can always safely compose aspects and advised modules, even when they were separately developed and compiled. This is very different from other approaches, where errors can occur at integration (weave) time. The key observation of this work is that a pointcut is not a sufficiently expressive abstraction for such interfaces; return and exception types ought to be taken into account.

In addition, join point interfaces support a notion of subtyping, which helps in structuring and managing the complexity of the space of events-of-interest to aspects. Subtyping of JPIs supports join point polymorphism and advice overriding. We introduce a novel semantics of advice dispatch that avoids the pitfalls of other approaches, inspired by the well-established multiple-dispatch semantics [8].

The contributions of this paper are as follows[1]:

- We illustrate why Steimann's join point types fail to support modular type checking and join point polymorphism (Section II).
- We describe the static and dynamic semantics of JPIs that allow for modular type checking and proper join point polymorphism (Section III).
- We provide a complete implementation of JPIs for AspectJ, based on the AspectBench Compiler [10] (Section IV).
- We evaluate the benefits of JPIs based on the study of a set of existing AspectJ programs (Section V).

[1]The general idea of JPIs was first presented in the New Ideas track of ESEC/FSE [9]. This paper is the result of the development and maturation of that idea paper; both syntax and semantics have evolved, and both implementation and evaluation are completely new.

```
1  class ShoppingSession { ...
2    void checkOut(Item item, float price,
3                  int amount, Customer cus){ ... }
4  }
5  aspect Discount {
6   pointcut checkingOut(Item item, float price,
7                        int amount, Customer cus):
8      execution(* ShoppingSession.checkOut(..))
9        && args(item, price, amount, cus);
10  void around(Item item, float price, int amt, Customer cus):
11    checkingOut(item, price, amt, cus) {
12    double factor = cus.hasBirthday()? 0.95 : 1;
13    proceed(item, price*factor, amt, cus);
14  }
15 }
```

Listing 1: Shopping session with discount aspect

Sections VI and VII discuss related work and conclude.[2]

II. Example

We motivate our approach using a running example based on an e-commerce system. A customer can check out a product by either buying or renting the product. A business rule states that, on his/her birthday, the customer is given a 5% discount when checking out a product. We will be adding further rules later.

Listing 1 shows an implementation of the example where the business rule is defined as an aspect in AspectJ [11]. The around advice in lines 10–14 applies the discount by reducing the item price to 95% of the original price when proceeding on the customer birthday. Note how brittle the AspectJ implementation is with respect to changes in the base code. Most changes to the signature of the `checkOut` method, such as renaming the method or modifying its parameter declarations, will cause the `BirthdayDiscount` aspect to lose its effect. The root cause of this problem is that the aspect, through its pointcut definition in lines 6–9, makes explicit references to named entities of the base code—here to the `checkOut` method.

A. Why previous work fails to provide modular reasoning

The motivation of this work is to introduce a layer of abstraction to mediate between aspects and advised code so as to support modular reasoning and independent development and evolution. This idea was already proposed by others; most notably, Steimann et al. recently proposed *join point types* (JPTs) [7].[3] To facilitate an easy comparison of both approaches we use similar examples here. As we will show, the abstraction provided by JPTs does not actually support modular type checking.

[2]The implementation is available, along with documentation, test cases, and applications at http://bodden.de/jpi. We also include supplementary material, such as examples of the problems described with other approaches.

[3]We use "JPTs" to refer to Steimann's proposal of join point types.

```
1  joinpointtype CheckingOut {
2   Item item; float price; int amt; Customer cus;
3  }
4  class ShoppingSession exhibits CheckingOut {
5   pointcut CheckingOut: execution(* checkOut(..))
6    && args(item, price, amount, cus);
7    //remainder of code as before
8  }
9  aspect Discount {
10  void around(CheckingOut jp) {
11   double factor = cus.hasBirthday()? 0.95 : 1;
12   jp.price = jp.price * factor;
13   proceed(jp);
14  }
15 }
```

Listing 2: Shopping session example with JPTs

Listing 2 shows the previous example with JPTs. A JPT is defined as a plain data structure: lines 1–3 define the join point type `CheckingOut` along with the names and types of context parameters that join points of this type expose. The base class `ShoppingSession` is enhanced to declare that it exhibits join points of type `CheckingOut`. In lines 5–6 the class binds the join point type to concrete join points, using a regular AspectJ pointcut. Through the use of join point types, the aspect is completely liberated from pointcut definitions. Note that in line 10 the aspect refers directly to the join point type, obviating the need to explicitly refer to base code elements. Using this type-based abstraction, it may appear that there are no longer any hidden dependencies between aspects and advised code.

Unfortunately, however, Steimann's proposal of join point types does not eliminate all hidden dependencies between aspects and advised code, and is therefore unable to properly support modular reasoning: JPTs lack crucial information necessary for modular type checking. Indeed, notice that the join point type definition in lines 1–3 of Listing 2 specifies neither the return type of these join points, nor the checked exceptions that the execution of these join points may throw. Because this information is lacking, JPTs must defer checking of some conditions, such as return type compatibility, to weave time. This makes independent development error prone, hinders aspect reuse, and is highly undesirable in a collaborative working environment.

For example, consider that some requirements changed on the aspect side. To determine a customer's birthday the aspect now has to submit an SQL query:

```
void around(CheckingOut jp) throws SQLException {
        boolean hasBirthday = SQL.query(/*query omitted*/);
        double factor = hasBirthday ? 0.95 : 1;
        jp.price = jp.price * factor;
        proceed(jp);
}
```

SQL queries can cause checked `SQLExceptions`, and the aspect programmer, being careless, decides to just pass the exception on, by including `SQLException` in its **`throws`** clause. Nothing in the type system can prevent the programmer from doing so. Since the struct-based join point type definition contains no information about exceptions, a static type checker based on this definition cannot tell which exceptions are allowed or not. As a result, the type error remains latent until system integration time: weaving will fail and report an error. Since JPTs also lack information about the return type of a join point, similar issues arise. If the return type is changed, either on the side of the base code or on the side of an advice, such a change will go unnoticed on the other side, breaking the code at integration time.

To understand the gravity of the problem, one can make the analogy with interfaces in statically-typed languages like Java. Interfaces allow for modular type checking and independent development. However, if interfaces only included type information about method parameter types, eg.:

```
interface Printer {  print(Document d);  }
```

it would be easy to see that the type system would not be able to maintain soundness due to the lack of information about the return type and checked exception types of method `print`. Both implementors and clients of this interface would have to make informed guesses about those types, resulting in fragile code due to hidden dependencies.

As we showed, the JPT proposal ultimately falls short when it comes to providing important software engineering properties. As Steimann et al. write, they "have not yet explored whether and how [their] approach could open the door for separate compilation." [7, p.36]. But should not the primary goal of a type declaration be to give static guarantees such that modular checking and independent development can be done in a sound manner? In this paper we propose a stronger notion of interfaces for aspect-oriented programming, called *join point interfaces*, which address these fundamental shortcomings.

B. Modular reasoning with join point interfaces

Join point interfaces (JPIs for short) share with JPTs the idea of abstracting from join points through types; but in contrast to JPTs, join point interfaces are type declarations that actually do allow for modular type checking. Crucially, join point interfaces define join point types through method signatures instead of structs. This choice is for a good reason. In a recent piece of work, Bodden showed that one can avoid many semantic pitfalls by regarding join points as typed closures [12]. Importantly, because a JPI definition is akin to a method signature, it *does* specify return type and checked exception types, just like Java interfaces do. Listing 3 shows the shopping cart example in the syntax we propose, including the JPI definition in line 2. Similar

```
1  jpi void CheckingOut(Item item, float price, int amount,
2                       Customer cus);
3  class ShoppingSession {
4    exhibits void CheckingOut(Item i, float price,
5                              int amount, Customer c):
6      execution(* checkOut(..))
7      && args(i, price, amount, c);
8    ...
9  }
10 aspect Discount {
11   void around CheckingOut(Item item, float price,
12                           int amt, Customer cus){
13     double factor = cus.hasBirthday()? 0.95 : 1;
14     proceed(item, price*factor, amt, cus);
15   }
16 }
```

Listing 3: Shopping session example using a JPI

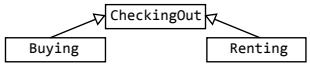


Figure 2: Inheritance between Join Point Interfaces

```
1  jpi void Renting(Item item, float price,
2                   int amount, Customer c)
3    extends CheckingOut(item, price, amount, c);
4
5  aspect Discount {
6   void around CheckingOut(Item item, float price, int amt,
        Customer cus) { /* as before */ }
7   void around Renting(Item item, float price,
8                       int amt, Customer cus) {
9    double factor = (amt > 5) ? 0.85 : 1;
10   proceed(item, price*factor, amt, cus);
11  }
12 }
```

Listing 4: Advice overriding

to JPTs, the programmer attaches pointcut definitions to classes. This is done through an **exhibits** declaration (lines 4–7).[4] Lines 11–15 show the advice declaration. As with JPTs, advices in our approach only refer to join point types, not directly to pointcuts. However, since we model join points and their types through method signatures, programmers access the exposed context information of join points through formal parameters of the advice (as in AspectJ), rather than field-like members (eg. line 12 in Listing 2).

Let us come back to the the case of introducing `SQLExceptions` on the aspect side to show how join point interfaces restore modular reasoning to AOP. The aspect programmer extends the advice signature similar to before:

```
void around CheckingOut(Item item, float price, int amt,
     Customer cus) throws SQLException { ...
```

With join point interfaces, however, this mistake is caught by the type system. The JPI declaration in line 2 of Listing 3 states that join points of this type cannot throw any checked exceptions. Hence, the JPI type system flags the modified advice as erroneous, notifying the aspect programmer that part of the JPI contract is now broken. The aspect programmer can now either handle `SQLExceptions` in the advice itself, or, if that is an option, update the JPI to:

```
jpi void CheckingOut(Item item, float price, int amount,
     Customer cus) throws SQLException;
```

When choosing the second option, the contract between aspect and base code is modified. Therefore, as soon as the base code programmer updates to the new interface, the type checker reports that exceptions of type `SQLException` must be caught at all code locations of join points of type `CheckingOut`, as specified by the **exhibits** declarations.

[4]While one could in principle infer the signature for the **exhibits** declaration from the referenced JPI declaration (line 2). We decided not to, so that the class definition is more self contained. After all, we envision JPI definitions to reside in their own separate package, shared with the aspect programmer. This is in stark contrast to JPTs: in line 6 of Listing 2, the programmer has to remember how the JPT declaration named its arguments.

C. Join Point Polymorphism

Join point interfaces abstract from join points through types. In the same way that object interfaces in languages like Java support a flexible form of subtype polymorphism, JPIs enable polymorphic join points. A join point can be seen as providing multiple JPIs, and advice dispatch at that join point can take advantage of this polymorphism.

Which JPIs does a join point provide? This is the role of pointcuts. As we have seen, a class defines the pointcuts that expose certain join points in its execution, following a given JPI. For instance, in Listing 3, class `ShoppingSession` defines a pointcut that gives the type `CheckingOut` to all join points that are executions of the `checkOut` method. Because a join point can be matched by several pointcuts, a join point can have multiple types. For instance, an execution of `checkOut` can be seen as a `CheckingOut`, and could additionally be seen as a `LoggableEvent` (a JPI whose definition is left to the imagination of the reader).

In addition, like interfaces in Java, JPIs support subtyping. Consider two subtypes of `CheckingOut`, `Buying` and `Renting` (Figure 2), and the following business rule: the customer gets a 15% discount when *renting* at least 5 products of the same kind; this promotion is not compatible with the birthday discount.

Listing 4 shows an implementation of this additional rule using the subtyping relationship on JPIs. First, we declare the JPI `Renting` as extending `CheckingOut`. The semantics of this subtyping relationship implies that `Renting` represents a subset of the join points that `CheckingOut` represents. The aspect `Discount` now declares two advices. The first one is the same as in the previous example. It applies to all

`CheckingOut` join points. The second advice *overrides* the first advice for all join points that are of type `Renting`.

Steimann's proposal also supports join point subtyping. However, join point polymorphism is not properly handled. To briefly illustrate why, let us consider that we introduce two subtypes of `Buying`: `BuyingBestSeller` and `BuyingEcoFriendly`. With JPTs, whenever a book is bought that is both a best seller and an eco-friendly print, the `Discount` advice for `CheckingOut` is executed twice! This implies that the side effects of the advice (*e.g.* sending a notification email) are duplicated for a single book purchase. The reason is that JPTs do not support polymorphic join points: instead, in the case above, *two separate* join point instances are generated, one of each type. Because both instances are subtypes of `CheckingOut`, the advice executes twice. To the best of our knowledge, our approach is the first to provide a clear and natural mechanism to address advice overriding. It supports true join point polymorphism: a single join point can have multiple types, and a given advice only executes *at most once* for a given join point.

III. Join Point Interfaces

We begin this section by describing the syntax of JPIs, designed as an extension to AspectJ (Section III-A). Join point interfaces allow for modular reasoning about aspect-oriented programs by precisely mediating the dependencies between aspects and base code (recall Figure 1b). The most fundamental contribution of JPIs therefore lies in the static type system that supports modular checking: we informally describe it in Section III-B. Our proposal of JPIs also innovates over previous work in the way it supports join point polymorphism. The dynamic semantics of JPIs are described in Section III-C. The actual implementation of JPIs in the AspectBench Compiler is described in Section IV.

A. Syntax

We have introduced the syntactic extension to AspectJ in order to support join point interfaces in the previous section. We provide the complete definition online.

Type declarations, which normally include classes, interfaces and aspects, are extended with a new category for JPI declarations. A **`jpi`** declaration specifies the full signature of a join point interface: the return type at the join points, the name of the join point interface, its arguments, and optionally, the checked exception types that may be thrown. A join point interface declaration can also specify a super interface, using **`extends`**. In that case the name of the extended JPI is given, and the arguments of the super interface are bound to the arguments of the declared JPI.

Classes and aspects can have a new kind of member declaration, for specifying the join point interfaces that are exhibited. An **`exhibit`** declaration specifies a join point interface signature and the associated pointcut expression that denotes the exhibited join points.

Finally, with JPIs, the way advices are declared is changed. Instead of directly referring to a pointcut expression, advices instead refer to a join point interface. The information about return type, argument types and checked exception types that the JPI specifies becomes part of the advice signature.

B. Static Semantics

We next describe the JPI type system. It supports modular type checking of both aspects and classes; only JPIs declarations are needed to type-check either side. This is similar to type checking Java code based on the interfaces it relies upon. We first discuss how JPIs are used to type check aspects. Then we turn to type checking base code that exhibits certain JPIs. We finally discuss type checking JPIs themselves, in particular considering JPI inheritance.

1) Type checking aspects: An aspect is type checked just like an AspectJ aspect, save for its advices. There are two facets of type checking an advice: checking its signature and checking its body. Type checking the signature of an advice is simple. Each advice declares an advised JPI in its signature; the signature must directly resemble the JPI definition. More precisely, both return and argument types must be exactly the same as those of the JPI.[5] Checked exceptions are dealt with similarly. The advice must declare the same exception types as the JPI it advises. It cannot declare any additional exception, as this could lead to uncaught checked exceptions on the side of the base code.

Type checking the advice body is similar to type checking a method body, with the additional constraint of considering calls to **`proceed`**. As it turns out, a join point interface is identical to a method signature (except for the **`extends`** clause, used for join point subtyping). In fact, a JPI specifies the signature of **`proceed`** within the advice body, thereby abstracting away from the specific join points that may be advised. This is a fundamental asset of JPIs, and the key reason why interfaces for AOP ought to be represented as method signatures (including return and exceptions types), and not as structures like JPTs. JPIs *fully* specify the behavior of advised join points, thereby allowing safe and modular static checking of advice.

2) Type checking base code: On the other side of the contract is base code, which can exhibit join points. Also this base code must obey the contract specified by join point interfaces. Part of this contract has to be fulfilled by the pointcut associated with the **`exhibit`** declaration: the pointcut has to bind all the arguments in the signature, using pointcut designators such as **`args`**, **`this`** and `target`.

[5]Note that the invariance on types in the advice signature is motivated by simplicity: it could be relaxed to allow covariance on the return type for instance, but that may be more confusing than helpful. The advice body can in any case always return a subtype of the declared return type.

The second part of the contract from the base-code point of view is to respect the return type and exception types of the JPI. This has to be checked at each join point potentially matched by the pointcut associated to the **exhibit** declaration. More precisely, the pointcut is matched against all join point shadows[6] in the lexical scope of the declaring class. Whenever the pointcut matches a join point shadow, the type system checks that this shadow is of the same type as the return type of the JPI. Similarly, it checks that the declared exceptions of the shadow are exactly the same as those of the JPIs. To understand this invariance requirement, consider the case of declared exceptions. Let us denote T_S the checked exception thrown at a join point, T_I the exception type declared in the JPI, and T_A the type thrown by the advice. If $T_S < T_I$[7], this means that the context of the shadow is not prepared to handle T_A if $T_S < T_A <: T_I$. Conversely, if $T_S > T_I$, the advice is not prepared to handle T_S when invoking **proceed**. Therefore $T_S = T_I$ by necessity. The same reasoning holds for the return type [14].

These type checks at the join point shadows are the fundamental contribution of JPIs from the point of view of type checking base code. Other approaches, like JPTs, are not able to perform these checks modularly, simply because the specification of return and exception types are not part of their interfaces. Those approaches have to defer these type checks to weave time, or even worse, to runtime. With JPIs, conformance can be checked statically and modularly, prior to linking and weaving.

3) Type checking JPIs: Finally, some type checking is performed at the level of JPIs themselves, to ensure that inheritance is used properly. The extending interface declares how arguments of the super interface are bound to its own arguments. The types of the arguments must coincide, as must the return type. Again we impose invariance on these types. A sub-interface can declare more arguments than its super interface. However, the exception interface of a sub-JPI must be the same as that of the extended JPI to preserve type soundness.

C. Dynamic Semantics

The dynamic semantics of JPIs differ slightly from that of a traditional aspect language. Briefly, the traditional model is as follows [15]. All aspects (pointcuts and associated advices) are present in a global environment. At each evaluation step, a join point representation is built and passed to all defined aspects. More precisely, the pointcuts of an aspect are given the join point in order to determine if the associated advices should be executed or not.

With JPIs, aspects do not have pointcuts. They advise JPIs, and base code defines the pointcuts that denote the join points that provide these JPIs. The global environment contains aspects (with their advices). Conceptually, at each evaluation step, a join point representation is passed *only* to the pointcuts defined in the current class. If a pointcut matches, then the join point is tagged with the corresponding JPI. Then, the advices that advise one of the tagged JPIs are executed. In presence of join point polymorphism and inheritance among JPIs, it is necessary to specify which advice is executed.

We write A_T to denote an advice of aspect A that advises JPI T; we write jp^T to denote a join point jp tagged with JPI T. The semantics of advice dispatch closely mimics the semantics of message dispatch in multiple dispatch languages like CLOS [16] and MultiJava [8]. Indeed, an aspect with its multiple advices (each declared to advise a specific JPI) can be seen as a generic function with its multiple methods. Once a join point jp is tagged with interfaces $T_1, \ldots, T_n$ we select, for each aspect A, all *applicable* advices. An advice A_S is *applicable* to $jp^{T_1,\ldots,T_n}$ if there exists an i such that $T_i <: S$. In order to support overriding, among all applicable advices $A_{S_1}, \ldots, A_{S_k}$, we invoke only the *most-specific* ones, defined as the A_{S_j} such that for all i, either $S_j <: S_i$ or $S_i \not<: S_j$.

In this work we are dealing with implicit invocation, which inherently supports multiple reactions to an event. This differs from multiple dispatch, which requires exactly one method to execute. The difference manifests in two ways in the semantics. First, if there are no applicable advice, then nothing happens; no advice executes. In contrast, a multiple-dispatch language throws a *message-not-understood* error if no applicable method can be found. Also, message dispatch requires that there is a *unique* most-specific applicable method, otherwise an *ambiguity* error is raised[8]. In our case, we execute all the most-specific applicable advices, in the precedence order imposed by regular AspectJ when multiple advices of a same aspect match the same join point [17].

In the above, we have overlooked one specificity of AspectJ and most aspect languages: the fact that advices can be of different *kinds*—before, after, or around. The advice overriding scheme we described is kind-specific: an advice may override another advice only if it is of the same kind. (In Section V, we will show cases where this is useful.) For instance, consider an aspect that defines two advices A_{T_1} and A_{T_2}, with $T_2 <: T_1$. If one is a before advice and the other is an after advice, both are executed upon occurrence of a join point jp^{T_2}. Conversely, if both are around advices, only the most specific (A_{T_2}) executes, as explained above.

In practice, we found that advice overriding is not always desirable (see Section V-B4). We support the possibility to declare an advice as **final**, meaning it will always execute if applicable, regardless of whether there exists a more specific

[6] A join point *shadow* is the expression in program text that corresponds to a (set of) dynamic join point(s) at runtime [13].

[7] We use $<:$ for subtyping, and $<$ for strict (*i.e.* non-reflexive) subtyping.

[8] In a statically-typed language like MultiJava, both cases can be ruled out by the type system.

applicable advices; in such a case, both execute, following the standard AspectJ composition rules.

A fundamental asset of the dispatch semantics presented here is that it gives the guarantee that a given advice executes *at most once* for any given join point. This is in stark contrast with the semantics of JPTs, where the same advice can surprisingly be executed several times for the same join point, as discussed in Section II-C.

IV. Implementation

We have fully implemented join point interfaces as an extension to the AspectBench Compiler (abc) [10]. Our abc extension extends Bodden's implementation of Closure Join Points [12]. This allows our approach to not only support implicit announcement (pointcuts select exhibited join points), but explicit announcement as well. We come back to this in Section VI.

The most interesting aspect of our implementation is how we assure the correct dispatch semantics for advices referring to JPIs. Remember that syntactically our advice declarations do not at all refer to any pointcut. Instead they refer to a JPI declaration which in turn may be bound to pointcuts by one or more **exhibits** declarations. To allow for maximal reuse of existing functionality in the abc compiler, we decided to implement our dispatch semantics through a transformation that would compute for each such advice a single pointcut, based on the referenced JPI, its type hierarchy and the **exhibits** declarations of those types. Let a be the advice to compute the pointcut for, *as* the set of other advices in the same aspect and *es* the set of all **exhibits** declarations in the program. Then we compute the pointcut for a as follows:

$$
\begin{aligned}
pc(a, as, es) &= pc^{+}(a.jpi, es) \wedge \neg pc^{-}(a, as, es) \\
pc^{+}(jpi, es) &= \bigvee_{e \in es,\ e.jpi <: jpi} e.pc \\
pc^{-}(a, as, es) &= \bigvee_{a' \in as,\ a' \sqsubset a} pc^{+}(a'.jpi, es)
\end{aligned}
$$

The equation[9] for pc^{+} implements polymorphism: if a refers to $a.jpi$ then a will match not only on join points for $a.jpi$ itself but also for all subtypes. The equation for pc^{-} implements advice overriding within the same aspect: if an advice a' has the same kind as a but refers to a more specific JPI type, then a' overrides a, which means that a will not execute for the join points of this JPI. For advice that has been declared **final**, pc^{-} is simply skipped, so as to avoid overriding.

To implement the above equation, our implementation has to overcome a few technical obstacles. JPI declarations can rename formal arguments of their super types in their **extends** clauses. Our implementation undoes this renaming in the back-end. Further, the pointcut $pc^{-}(a, as, es)$ is used under negation. This raises an issue with argument-binding pointcuts, like **this**(a), because they cannot be negated. Fortunately, abc supports a way to close such pointcuts so that they do not have any free variable anymore, by rewriting them to (λa.**this**(a)); such a pointcut can be negated, and if a is of type A, the negation is equivalent to !**this**(A), which yields the semantics we need.

[9] $\sqsubset$ denotes kind-specific subtyping for advices: $a' \sqsubset a$ means that a' and a are of the same kind and $a'.jpi < a.jpi$.

V. Evaluation

A. Benefits of Joint Point Interfaces

Join point interfaces establish a clear contract between base code and aspects, such that separate development can be supported. This is in essence similar to crosscutting programming interfaces (XPIs, see Section VI) and join point types (JPTs). The benefits of XPIs and JPTs on modularity have been empirically demonstrated elsewhere [6], [7]. Because JPIs are an extension of these approaches, JPIs enjoy the same modularity benefits, and more.

As we argued and illustrated in Section II, JPIs refine the notion of JPTs with information that is crucial for robust separate development: return and checked exceptions types are part of the interface. Therefore modules can be compiled separately, and no type errors occur at weave time. Because software inevitably has to evolve, it is an important task for a modular type checker to ensure that once a module conforms to an interface, composing it with other modules that also rely on the interface never raises an error at integration time. It is obviously crucial that changes in the type of the return at a join point, or for an advice, be detected modularly.

The same necessity happens with checked exception types. Robillard and Murphy [18] report on an effort to design robust programs with exceptions. They report that focusing on specifying and designing the exceptions from the very early stages of development of a system is not enough; exception handling is a global phenomenon that is difficult and costly to fully anticipate in the design phase. Thus, inevitably, the exceptions that can be thrown from modules are bound to evolve over time, as development progresses and this global phenomenon is better understood. The support that JPIs provide to report exception conformance mismatch between aspects and advised code in a modular fashion is therefore particularly necessary: as modules change their exception interface, immediate and local feedback is crucial to decide if these changes must be promoted to the actual contract between aspects and advised code. This avoids errors before system integration time.

B. Join Point Polymorphism

To evaluate JPIs in practice, we have converted a set of existing AspectJ applications from the corpus of Khatchadourian et al. [19]. These rewritten projects are available online. Then, to assess our semantics for join point

polymorphism, we have closely inspected a set of interesting subjects from this corpus: AJHotDraw, an aspect-oriented version of JHotDraw, a drawing application; Glassbox, a diagnosis tool for Java applications; SpaceWar, a space war game that uses aspects to extend the game in various respects; and LawOfDemeter, a small set of aspects checking the compliance to programming rules.

The first three projects were selected because of their comparatively large size and number of aspects. LawOfDemeter is a rather small project that showcases an interesting use of pointcuts, as discussed further below.

We inspected the programs using both the Eclipse AspectJ Development Tools [20] and AspectMaps [21]. These tools allowed us to easily identify which advices advise which join point shadows. In particular, we focused on the shadows that are advised by more than one advice, as this hints at potential for subtyping. We also systematically investigated all pointcut expressions used in these projects and looked for potential type hierarchies. Our investigation revealed several interesting example hierarchies and clearly supports the usefulness of our semantics of join point subtyping. We now discuss a few representative examples.

1) Subtyping Patterns: We identify two patterns that programmers use to "emulate" subtyping with pointcuts.

LawOfDemeter contains the following pointcuts:

```
pointcut MethodCallSite(): scope() && call(* *(..));
pointcut MethodCall(Object thiz, Object targt):
   MethodCallSite() && this(thiz) && target(targt);
pointcut SelfCall(Object thiz, Object targt):
   MethodCall(thiz,targt) && if(thiz == targt);
```

These pointcuts form an instance of a pattern that we call *subtype by restriction*. `MethodCall` restricts the join points exposed by `MethodCallSite` to instance methods, through additional **`this`** and **`target`** pointcuts. `SelfCall` restricts this set further by identifying self calls using an additional **`if`** pointcut. A programmer could model this join point type hierarchy with JPIs as follows:

```
1 jpi Object MethodCallSite();
2 jpi Object MethodCall(Object thiz, Object targt)
3     extends MethodCallSite();
4 jpi Object SelfCall(Object thiz, Object targt)
5     extends MethodCall(thiz, targt);
```

The example shows that it is useful to allow subtypes to expose more arguments than their super types: `MethodCall` exposes `thiz` and `targt`, while `MethodCallSite` exposes nothing at all.

SpaceWars includes various instances of the *subtype by restriction* pattern, but also features instances of the dual pattern, *super type by union*. Consider the following:

```
pointcut syncPoint():
        call(void Registry.register(..))   ||
        call(void Registry.unregister(..)) ||
        call(SpaceObject[] Registry.getObjects(..)) ||
        call(Ship[] Registry.getShips(..));
pointcut unRegister(Registry registry):
        target(registry) &&
        (call(void register(..)) || call(void unregister(..))
             );
```

Here the pointcut `unRegister` matches a subset of the join points matched by `syncPoint` because `syncPoints` includes additional join points by disjunction (set union). Here also, the subtype induced by `unRegister` exposes an additional argument.

2) Depth of Subtyping Hierarchies: Glassbox proved to be a very interesting case study in that it provides over 80 aspects, and more than 200 pointcut definitions, with potential for non-trivial join point type hierarchies. Due to space limitations, we only show one of the most interesting examples in Figure 3. The figure shows a hierarchy formed by 11 pointcuts within the aspect `ResponseTracer`. We have added `Stats` as a root type that the aspect does not contain explicitly, but which could be introduced to abstract the common parts of the `StartStats` and `EndStats` pointcuts.

3) Per-Kind Advice Overriding: AJHotDraw contains the following definitions:

```
pointcut commandExecuteCheck(AbstractCommand acommand) :
        this(acommand)
        && execution(void AbstractCommand+.execute()) ..
        && !within(*..JavaDrawApp.*);
before(AbstractCommand acommand):
        commandExecuteCheck(acommand) {..}
pointcut commandExecuteNotify(AbstractCommand acommand) :
        commandExecuteCheck(acommand)
        && !within(org.jhotdraw.util.UndoCommand) ..
        && !within(org.jhotdraw.contrib.zoom.ZoomCommand);
after(AbstractCommand acommand):
        commandExecuteNotify(acommand) {..}
```

This is another instance of the *subtype by restriction* pattern, with `commandExecuteNotify` refining the pointcut `commandExecuteCheck`. This example validates our semantics to consider advice kinds separately when resolving advice overriding (Section III-C). In the example, the first pointcut is advised with a before advice, while the second is advised by an after advice. Assume now that we had abstracted from those pointcuts using JPIs as in the following code:

```
before CheckingView(AbstractCommand acommand){..}
after NotifyingView(AbstractCommand acommand){..}
```

In this example, `NotifyingView` is a JPI subtype of `CheckingView`. If we did *not* separate advices by advice kind when determining advice overriding then only the `NotifyingView` would execute at a `NotifyingView` join point, leading to an altered semantics compared to the original

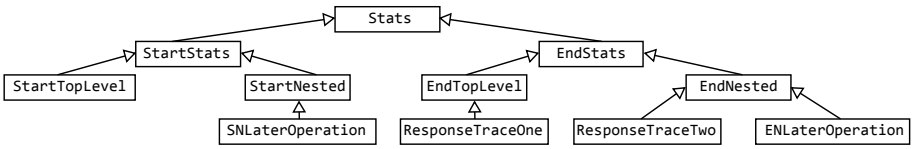


Figure 3: A Potential Hierarchy of Join Point Interfaces in Glassbox.

AspectJ program. Conversely, because we *do* separate advices by kind, when encountering a `NotifyingView` join point, the `CheckingView` advice is executed before the join point and the `NotifyingView` afterwards.

4) Overriding and the Multiple-Execution Bug: Glassbox showcases both the bug of JPTs with multiple execution of the same advice (Section II-C), and the interest of being able to declare some advice as **`final`** to avoid overriding.

An aspect in charge of system initialization advises the execution of TestCase object constructors. Some test classes implement the InitializedTestCase interface (ITC for short), and some implement the ExplicitlyInitializedTestCase interface (EITC for short); some classes implement both (these interfaces are added via inter-type declarations). More precisely, 5 classes implement only ITC, 5 classes EITC only, and 1 class implements both. The aspect defines four before advices on these constructors, discriminating between different categories using pointcuts. These pointcuts correspond to four join point types $T_1 \ldots T_4$, where T_1 is a super type of the three others.

With JPTs, T_1 never executes, because it is always overridden. The solution is to refactor the aspect to move the advice for T_1 in a separate aspect. But then, for all classes that implement only one of the two interfaces (10 out of 11), the advice associated to T_1, which initializes a factory object, is executed twice. Whenever a join point is of sibling join point types and there is an advice associated to the common super type, the advice executes as many times as there are siblings involved. This example shows that it can be non-trivial to reason about when such a multiple-execution bug can happen, because the hierarchy of join point types does not align with the hierarchy of classes and interfaces. For users of an aspect-oriented framework, this can be particularly hard to track down. The dispatch semantics of JPIs avoids this problem. In addition, with JPIs it is possible to declare the advice for T_1 as **`final`**, and therefore it is not necessary to introduce a separate aspect.

C. Flexibility of the Type System

The typing rules currently enforce invariance on both return and exception types of join point interfaces (Section III-B). This is the simplest way to ensure soundness, but can be too rigid in practice. If a pointcut matches a large number of shadows, invariance forces us to define different JPIs (and advices) for the different types of shadows. In applications that rely on wide-matching pointcuts, such as LawOfDemeter, this is clearly problematic: during conversion, the number of advices increased from 6 to 26. On the other hand, in an application like AJHotDraw, in which most pointcuts are very specific, we found that only 3 advices are affected by the rigidity of the current typing rules. The total number of advices increased from 48 to 51. This issue could be addressed with parametric advice types as in StrongAspectJ [14]; we leave such an extension to future work.

VI. Related Work

There is a very large body of work that is concerned with modularity issues raised by the form of implicit invocation with implicit announcement provided by aspect-oriented programming languages like AspectJ, starting with Gudmundson and Kiczales [3]. In the AOP literature, many proposals have been formulated, some aiming at providing more abstract pointcut languages (e.g. [22]), and others—as we do here—introducing some kind of interface between aspects and advised code. A detailed discussion of all these approaches is outside the scope of this paper, so we concentrate on the most salient and most related proposals. An exhaustive treatment of this body of work and neighbor areas can be found in [7].

In their ICSE 2005 paper, Kiczales and Mezini argue that when facing crosscutting concerns, programmers can regain modular reasoning by using AOP [1]. Doing so requires an extended notion of interfaces for modules, called aspect-aware interfaces, that can only be determined once the *complete* system configuration is known. While the argument points at the fact that AOP provides a better modularization of crosscutting concerns than non-AOP approaches, it does not do anything to actually enable modular type checking and independent development. Aspect-aware interfaces are the conceptual backbone of current AspectJ compilers and tools, which resort to whole program analysis, and perform checks at weave time.

Sullivan et al. [6] formulated a lightweight approach to alleviate coupling between aspects and advised code. Crosscutting interfaces, XPIs for short, are design rules that aim at establishing a contract between aspects and base code by means of plain AspectJ. With the XPI approach, aspects advising the base code only define advices, no pointcuts. The pointcuts, in turn, are defined in another aspect representing the XPI. Griswold et al. argue that this additional layer of indirection improves the system evolution because the resulting XPI is a separate entity and hence can be agreed

upon as a contract. The authors also show how parts of such a contract can be checked automatically using static crosscutting or contract-checking advices in the XPI aspect itself. However, without a language-enforced mechanism, XPIs cannot provide any strong guarantees on modularity.

In the same period, Aldrich formulated the first approach for language-enforced modularity, Open Modules [5]. Here, modules are properly encapsulated and protected from being advised from aspects. A module can then open up itself by exposing certain join points, described through pointcuts that are now part of the module's interface. The advantage is that aspects now rely on pointcuts for which the advised code is explicitly responsible. Aldrich formally proves that this allows replacing an advised module with a functionally equivalent one (but with a different implementation) without affecting the aspects that depend on it. Ongkingco et al. have implemented a variant of Open Modules for AspectJ [23].

Steimann's join point types (JPTs) [7], which we have extensively discussed in this paper, can be seen as a further (and the most recent) step in the line of Open Modules. Still being a language-enforced mechanism, JPTs provide a higher level of abstraction than pointcuts: join point types, which can be organized in a subtype hierarchy, provide a more natural way to deal with complexity, just like interfaces in Java help classify object behaviors. Also, join point types open opportunities for advice overriding. JPTs do not handle polymorphism properly; we fix this through a proper dispatch semantics inspired by multi-methods.

A major contribution of our work is to realize that all the above proposals—XPIs, Open Modules, JPTs—rely on *insufficiently expressive interfaces* to really allow separate development and modular type checking. XPIs and Open Modules use pointcuts; and JPTs, although aesthetically different from named pointcuts, do not include any additional information. Both lack information about the return type and checked exception types of join points.

Open Modules and JPTs have different takes on an interesting design decision: should classes be aware of the join points they expose? With Open Modules, classes themselves do not declare their exposed join points; it is the task of the module. The argument is that the maintainer of the module is in charge of all the classes inside the module, and therefore, has sufficient knowledge to maintain classes in sync with the pointcuts in the module interface. Steimann and colleagues, on the other hand, argue for class-local exhibit clauses: each class is responsible for what it exhibits. In JPTs, even nested classes are not affected by the exhibited pointcuts of their enclosing classes. Our current proposal is actually half-way between both standpoints. We do not extend Java with a new notion of modules (this is left for future work), but we do support nested classes in the sense that the exhibited pointcut of a class match join points in nested classes as well. This means that we can use class nesting as a structuring module mechanism—although this approach to modules is certainly not as well supported in Java as it would be in other languages, such as Newspeak [24], where modules are objects, supported by a very flexible virtual class system.

Another answer to the design question above is to have join points raised explicitly in the code, as proposed by Hoffman and Eugster [25], the Ptolemy language [26], and also supported by Steimann's proposal. Ptolemy introduces event types, which are similar to JPTs in the sense that they are struct-like specifications; they include information about the return type, but not about checked exceptions. Also, event subtyping is not supported. Ptolemy supports behavioral contracts, called translucid contracts [27], to specify and verify control effects induced by event handlers. These verification techniques go beyond more lightweight interfaces like Java interfaces and JPIs. EScala [28] is an approach to modular event-driven programming in Scala, combining implicit and explicit events. EScala does not support **`around`** advice, so event definitions need not declare return types; however exception types are missing. EScala treats both events and handlers as object members, subject to encapsulation and late binding. Aspects are scoped with respect to event owners rather than event types.

It is important to highlight that join point interfaces are not tied to implicit announcement; in fact, although not the focus of this paper, our implementation also supports explicit join points. More precisely, we have integrated JPIs with closure join points [12]. Closure join points address many issues related to control and data flow common to all other approaches to explicit join points.

VII. Conclusions and Future Work

Join point interfaces enable fully modular type checking of aspect-oriented programs by establishing a clear contract between aspects and advised code. Like interfaces in statically-typed object-oriented languages, JPIs support independent development in a robust and sound manner. Key to this support is the specification of JPIs as method-like signatures with return types and checked exception types. JPIs can be organized in hierarchies to structure the space of join points in a flexible manner, enabling join point polymorphism and dynamic advice dispatch. We have implemented JPIs as an AspectJ extension, and have rewritten several existing AspectJ programs to take advantage of JPIs. This study supports our major design choices.

Our study motivates the need to relax the invariance requirements imposed on JPIs. We have also started to study more advanced mechanisms for polymorphism and reuse: multiple inheritance of JPIs, flexible support for advice overriding, as well as a mechanism similar to **`super`** calls. Such mechanisms should improve the support for refinement of both pointcuts and advices for join point subtypes. The latter is not trivial because of the interaction between **`super`** and **`proceed`** in around advices [9].

REFERENCES

[1] G. Kiczales and M. Mezini, "Aspect-oriented programming and modular reasoning," in *Proceedings of the 27th international conference on Software engineering (ICSE 2005)*. St. Louis, MO, USA: ACM, 2005, pp. 49–58.

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

[3] S. Gudmundson and G. Kiczales, "Addressing practical software development issues in AspectJ with a pointcut interface," in *Proceedings of the Workshop on Advanced Separation of Concerns*, 2001.

[4] F. Steimann, "The paradoxical success of aspect-oriented programming," in *Proceedings of the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2006)*. Portland, Oregon, USA: ACM, Oct. 2006, pp. 481–497.

[5] J. Aldrich, "Open modules: Modular reasoning about advice," in *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP 2005)*, ser. LNCS, A. P. Black, Ed., no. 3586. Glasgow, UK: Springer-Verlag, Jul. 2005, pp. 144–168.

[6] K. Sullivan, W. G. Griswold, H. Rajan, Y. Song, Y. Cai, M. Shonle, and N. Tewari, "Modular aspect-oriented design with XPIs," *ACM Transactions on Software Engineering and Methodology*, vol. 20, no. 2, Aug. 2010.

[7] F. Steimann, T. Pawlitzki, S. Apel, and C. Kästner, "Types and modularity for implicit invocation with implicit announcement," *ACM Transactions on Software Engineering and Methodology*, vol. 20, no. 1, pp. 1–43, 2010.

[8] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein, "MultiJava: Modular open classes and symmetric multiple dispatch in java," in *Proceedings of the 15th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2000)*. Minneapolis, Minnesota, USA: ACM, Oct. 2000, pp. 130–145.

[9] M. Inostroza, É. Tanter, and E. Bodden, "Join point interfaces for modular reasoning in aspect-oriented programs," in *ESEC/FSE '11: Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Sep. 2011, New Ideas Track.

[10] 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 ACM International Conference on Aspect-Oriented Software Development (AOSD 2005)*. Chicago, Illinois, USA: ACM, Mar. 2005, pp. 87–98.

[11] R. Laddad, *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Press, 2003.

[12] E. Bodden, "Closure joinpoints: block joinpoints without surprises," in *Proceedings of the 10th ACM International Conference on Aspect-Oriented Software Development (AOSD 2011)*. Porto de Galinhas, Brazil: ACM, Mar. 2011, pp. 117–128.

[13] H. Masuhara, G. Kiczales, and C. Dutchyn, "A compilation and optimization model for aspect-oriented programs," in *Proceedings of Compiler Construction (CC2003)*, ser. LNCS, G. Hedin, Ed., vol. 2622. Springer-Verlag, 2003, pp. 46–60.

[14] B. De Fraine, M. Südholt, and V. Jonckers, "StrongAspectJ: flexible and safe pointcut/advice bindings," in *Proceedings of the 7th ACM International Conference on Aspect-Oriented Software Development (AOSD 2008)*. Brussels, Belgium: ACM, Apr. 2008, pp. 60–71.

[15] M. Wand, G. Kiczales, and C. Dutchyn, "A semantics for advice and dynamic join points in aspect-oriented programming," *ACM Transactions on Programming Languages and Systems*, vol. 26, no. 5, pp. 890–910, Sep. 2004.

[16] A. Paepcke, Ed., *Object-Oriented Programming: The CLOS Perspective*. MIT Press, 1993.

[17] AspectJ Team, "The AspectJ programming guide," http://www.eclipse.org/aspectj/doc/released/progguide/, 2003.

[18] M. Robillard and G. Murphy, "Designing robust Java programs with exceptions," in *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '00/FSE-8)*, 2000, pp. 2–10.

[19] R. Khatchadourian, P. Greenwood, A. Rashid, and G. Xu, "Pointcut rejuvenation: Recovering pointcut expressions in evolving aspect-oriented software," in *International Conference on Automated Software Engineering (ASE 2009)*. IEEE/ACM, Nov. 2009, pp. 575–579.

[20] "Eclipse AspectJ Development Tools," http://www.eclipse.org/ajdt/.

[21] J. Fabry, A. Kellens, and S. Ducasse, "Aspectmaps: A scalable visualization of join point shadows," in *Proceedings of 19th IEEE International Conference on Program Comprehension (ICPC2011)*. IEEE Computer Society Press, Jul 2011, pp. 121–130.

[22] K. Gybels and J. Brichau, "Arranging language features for more robust pattern-based crosscuts," in *Proceedings of the 2nd ACM International Conference on Aspect-Oriented Software Development (AOSD 2003)*, M. Akşit, Ed. Boston, MA, USA: ACM Press, Mar. 2003, pp. 60–69.

[23] N. Ongkingco, P. Avgustinov, J. Tibble, L. Hendren, O. de Moor, and G. Sittampalam, "Adding open modules to aspectj," in *Proceedings of the 5th ACM International Conference on Aspect-Oriented Software Development (AOSD 2006)*. Bonn, Germany: ACM, Mar. 2006, pp. 39–50.

[24] G. Bracha, P. Ahe, V. Bykov, Y. Kashai, W. Maddox, and E. Miranda, "Modules as objects in newspeak," in *Proceedings of the 24th European Conference on Object-oriented Programming (ECOOP 2010)*, ser. LNCS, T. D'Hondt, Ed., no. 6183. Maribor, Slovenia: Springer-Verlag, Jun. 2010, pp. 405–428.

[25] K. Hoffman and P. Eugster, "Bridging Java and AspectJ through explicit join points," in *Proceedings of the 9th International Symposium on Principles and Practice of Programming in Java (PPPJ 2007)*. ACM, 2007, pp. 63–72.

[26] H. Rajan and G. T. Leavens, "Ptolemy: A language with quantified, typed events," in *Proceedings of the 22nd European Conference on Object-oriented Programming (ECOOP 2008)*, ser. LNCS, J. Vitek, Ed., no. 5142. Paphos, Cyprus: Springer-Verlag, july 2008, pp. 155–179.

[27] M. Bagherzadeh, H. Rajan, G. T. Leavens, and S. Mooney, "Translucid contracts: expressive specification and modular verification for aspect-oriented interfaces," in *Proceedings of the 10th ACM International Conference on Aspect-Oriented Software Development (AOSD 2011)*. Porto de Galinhas, Brazil: ACM, Mar. 2011, pp. 141–152.

[28] V. Gasiunas, L. Satabin, M. Mezini, A. Núñez, and J. Noyé, "EScala: modular event-driven object interactions in Scala," in *Proceedings of the 10th ACM International Conference on Aspect-Oriented Software Development (AOSD 2011)*. Porto de Galinhas, Brazil: ACM, Mar. 2011, pp. 227–240.