Technical Report

Nr. TUD-CS-2012-0106
May 18th, 2012

Safe and Practical Decoupling of Aspects with Join Point Interfaces

NOTE:
The contents of this document have been superseded by the following publication.

Joint Point Interfaces for Safe and Flexible Decoupling of Aspects
(Eric Bodden, Éric Tanter, Milton Inostroza)
In ACM Transactions on Software Engineering and Methodology (TOSEM), 2013.

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

A

Safe and Practical Decoupling of Aspects with Join Point Interfaces

Eric Bodden, Technische Universität Darmstadt
Éric Tanter, University of Chile
Milton Inostroza, University of Chile

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. In this work, we introduce a novel abstraction called Join Point Interfaces, which, by design, supports modular reasoning and independent evolution by decoupling aspects from base code and by providing a modular type-checking algorithm. Join point interfaces can be used both with implicit announcement through pointcuts, and with explicit announcement, using closure join points. Join point interfaces further offer polymorphic dispatch on join points, with an advice-dispatch semantics akin to multi-methods. In this work, we show how our proposal solves a large number of problems observed in previous related approaches. We have implemented join point interfaces as an open-source extension to AspectJ. A first study on existing aspect-oriented programs supports our initial design in general, but also highlights some limitations, which we then address by introducing parametric polymorphism and a more permissive quantification mechanism. As a result, join point interfaces are a safe and practical way of decoupling aspects.

Categories and Subject Descriptors: D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms: Design, Languages

Additional Key Words and Phrases: aspect-oriented programming, modularity, typing, interfaces, implicit announcement, explicit announcement, join point polymorphism, advice dispatch

1. 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." [Kiczales and Mezini 2005]

While Aspect-Oriented Programming (AOP) [Filman et al. 2005] 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* are 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 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

Eric Bodden is supported by the German Federal Ministry of Education and Research (BMBF) within EC SPRIDE and by the Hessian LOEWE excellence initiative within CASED. Éric Tanter is partially funded by FONDECYT Project 1110051.
Author's addresses: Eric Bodden, Secure Software Engineering Group, EC SPRIDE, Mornewegstr. 30, 64289 Darmstadt, Germany; email:bodden@acm.org — É. Tanter and M. Inostroza, PLEIAD Laboratory, Computer Science Department (DCC), University of Chile, Blanco Encalada 2120, Santiago, Chile; email:{etanter,minostro}@dcc.uchile.cl

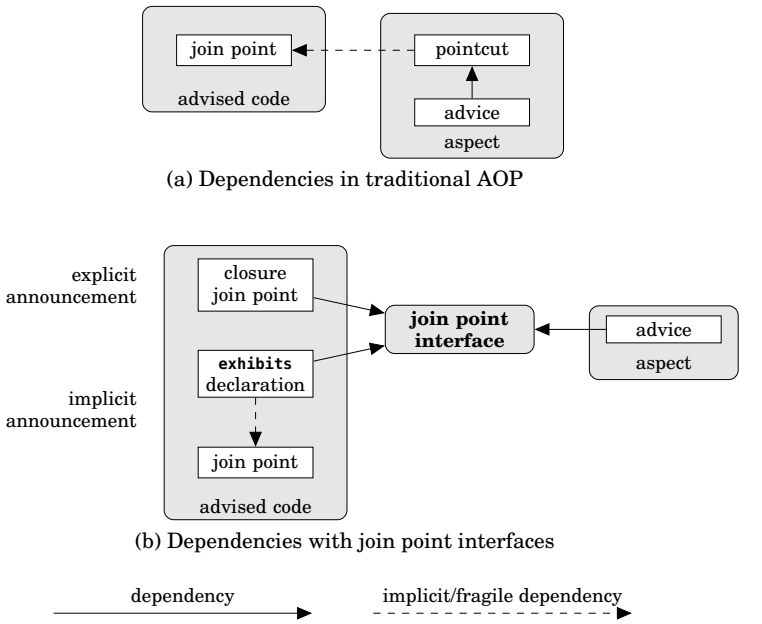

Fig. 1: Dependencies in traditional AOP and with join point interfaces.

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 [Gudmundson and Kiczales 2001] and have triggered a rich discussion in the community [Kiczales and Mezini 2005; Steimann 2006]. 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. [Gudmundson and Kiczales 2001; Aldrich 2005; Sullivan et al. 2010; Steimann et al. 2010]). However, as we show in this work, while those proposals 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.

The main contribution of this paper is to enable fully modular type checking for aspects through *join point interfaces* (JPIs), which are type-based contracts between aspects and advised code (Figure 1b). JPIs support a programming methodology where aspects only specify the types of join points they advise, but do not comprise any pointcuts. It is the responsibility of the programmer maintaining the advised code to specify, through an **`exhibits`** clause, which join points her code exposes, and of which type those join points are. In addition to implicit announcement through pointcuts that quantify over join points, JPIs are integrated with Closure Join Points [Bodden 2011] a mechanism for explicit event announcement.

Aspects and advised code can be developed and evolved independently. Conversely to previous work [Aldrich 2005; Sullivan et al. 2010; Steimann et al. 2010], 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.

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 [Clifton et al. 2000].

After a first thorough evaluation of JPIs on a corpus of AspectJ projects, we identified two major limitations of our approach, which led us to extend our design in two directions. First, we noticed that join point interfaces comprising return and exception types can easily become too specific to allow for flexible quantification through pointcuts, causing code duplication on the side of the aspect. To mitigate the problem, we introduce *generic JPIs*, using type parameters. As we show, these type parameters yield a solution that is both type safe and flexible. Second, we observed that JPIs are not well-suited for aspects that are highly crosscutting, such as profiling or data race detection, because they require programmers to define corresponding **`exhibits`** clauses in each advised classes. To mitigate this problem, we introduce a mechanism called *global pointcuts*, which makes it possible to balance the design trade-off between unrestricted quantification as in AspectJ and per-class **`exhibits`** clauses. A revisited evaluation validates the practical interest of these extensions.

Our project's web page contains a detailed documentation of our language extension, download links to our implementation (as open source), as well as all subject programs used for our case studies: `http://bodden.de/jpi/`

This paper presents the following original contributions:

— the design of join point interfaces, a novel mechanism for decoupling aspect definitions from base code, implemented as an extension of AspectJ;
— a detailed explanation of why join point interfaces are the first abstraction to allow for modular type checking in combination with implicit announcement;
— a novel semantics for advice dispatch in presence of join point polymorphism, based on multiple dispatch;
— the design and implementation of generic JPIs, to improve on flexibility while retaining type safety;
— the design and implementation of a controlled global pointcut mechanism, which can be used to support highly-crosscutting aspects;
— an extensive case study of existing aspect-oriented projects that is used to justify and assess our design decisions.

The notion of JPIs was first presented in the New Ideas track of ESEC/FSE [Inostroza et al. 2011]. This work is the result of the development and maturation of that idea; syntax and semantics have evolved, and both implementation and evaluation are completely new, as are the ideas of generic JPIs and global pointcuts.

Outline. The remainder of this paper is structured as follows. We next introduce join point interfaces through a running example. Section 3 explains our type system and the dynamic semantics in full generality. We discuss our implementation in Section 4, and present a first evaluation of join point interfaces in Section 5. As this evaluation shows, the core version of join point interfaces is provides type safety but lacks flexibil-

```
1  class ShoppingSession { ...
2    void checkOut(Item item, float price, int amount, Customer cus){
3      cart.add(item, amount);   //fill shopping cart
4      cus.charge(price);      //charge customer
5      totalValue += price;    //increase total value of session
6    }
7  }
8  aspect Discount {
9    pointcut checkingOut(float price, Customer cus):
10     execution(* ShoppingSession.checkOut(..)) && args(*, price, *, cus);
11
12   void around(float price, Customer cus):
13     checkingOut(price, cus) {
14     double factor = cus.hasBirthday()? 0.95 : 1;
15     proceed(price*factor, cus);
16   }
17 }
```

Listing 1: Shopping session with discount aspect

ity. In Sections 6 and 7, we thus explain how generic JPIs and global pointcuts restore this flexibility. We discuss related work in Section 8 and conclude in Section 9.

2. A FIRST TOUR OF JOIN POINT INTERFACES

We begin by defining an introductory example in plain AspectJ [Laddad 2003], which we will then improve using join point interfaces. We assume an e-commerce system, in which 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 plain AspectJ. The around advice in lines 12–16 applies the discount by reducing the item price to 95% of the original price when proceeding on the customer's 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 9–10, makes explicit references to named entities of the base code—here to the `checkOut` method.

2.1. Defining Join Point Interfaces

For this example, programmers could use the following join point interface definition to successfully decouple the aspect definition from the advised base code:

```
jpi void CheckingOut(float price, Customer cus);
```

In our core language design, join point interfaces are, except for the **`jpi`** keyword, syntactically equivalent to method signatures. This is for a good reason: methods are designed to be modular units of code that can be type-checked in a modular way, and we wish to inherit this property for join point interfaces. The method signature chosen for a join point interface typically coincides with the signature chosen for the advice that handles the appropriate join point (see Line 12 in Listing 1).

It is important to note that join point interface signatures are assumed to be complete with respect to the checked exceptions they define. The above JPI definition declares no exception, which gives both the aspect and the base code the guarantee that the respective other side of the interface cannot throw any checked exceptions at join points of this type. We will come back on the details in Section 3, and as we discuss in Section 8, our system is the first to correctly handle exceptions in that manner.

2.2. Advising JPIs

The following piece of code shows how programmers would advise join points defined by a join point interface:

```
aspect Discount {
    void around CheckingOut(float price, Customer c) {
        double factor = c.hasBirthday()? 0.95 : 1;
        proceed(price*factor, c);
    }
}
```

Crucially, advices in our approach only refer to join point types, not to pointcuts. Because of this, advices are completely decoupled from the code they advise; their dependencies are defined just in terms of a JPI declaration. As shown in the example, similar to AspectJ, an advice can use the formal parameters of a JPI to obtain context information exposed at join points. As we will explain in Section 3, we impose strict typing constraints on argument and return types. In plain AspectJ, this is not the case. In particular, an AspectJ can use `Object` in the return-type position as a wildcard that just denotes "any type". In addition, AspectJ uses unsafe co-variant typing through **`this`**, **`target`** and **`args`** pointcuts (more on this in Section 4).

2.3. Implicit Announcement with Pointcuts

Programmers have two different ways to raise join points declared through join point interfaces, through implicit and through explicit announcement. We first explain implicit announcement, in which join points are raised automatically at certain program events captured by AspectJ pointcuts. In our running example, the class `ShoppingSession` can raise the appropriate `CheckingOut` join points as follows:

```
class ShoppingSession {
    exhibits void CheckingOut(float price, Customer c):
        execution(* checkOut(..)) && args(*, price, *, c);
    ...
}
```

In this piece of code, the programmer raises join points implicitly, through an **`exhibits`** clause. Programmers will usually use this construct whenever wishing to concisely expose multiple join points from the same class. Within our JPI language, a pointcut attached to an **`exhibits`** clause only matches join points that originate from a code fragment lexically contained within the declaring class. This is to avoid an overly complex semantics with respect to subclassing, as previously observed by others [Steimann et al. 2010]. As for advice, our type system checks that the necessary constraints are satisfied such that weave-time and runtime errors are avoided (details in Section 3).

```
1 class ShoppingSession {
2     void checkingOut(final Item item, float price, int amount, Customer cus) {
3        totalValue = exhibit CheckingOut(float alteredPrice, Customer c) {
4                         sc.add(item, amount);
5                         cus.charge(alteredPrice);
6                         return totalAmount + alteredPrice;
7                     } (price,cus);
8     }
9 }
```

Listing 2: Example from Figure 1 with closure join points

2.4. Explicit Announcement with Closure Join Points

Pointcuts suffer from the *fragile pointcut problem* [Gybels and Brichau 2003; Stoerzer and Graf 2005]: when software evolves, the patterns of a pointcut can accidentally miss join points or match unintended ones, resulting in surprising and unintended behavior. With join point interfaces, even when using implicit announcement, this problem is much less severe because the scope of quantification is restricted: pointcuts can only match join points within the same class, and hence programmers can easily see how a pointcut affects a given class. Nevertheless, we offer programmers a language construct for *explicit announcement* as well, which allows programmers to explicitly mark expressions, statements or sequences to be advised by aspects. Explicitly announced join points are useful whenever pointcuts are either not expressive enough, too awkward, or too concrete to conveniently describe exactly which part of the execution should be advised. For instance, Steimann et al. [2010] recently showed that out of 484 advices in an aspect-oriented version of BerkelyDB [2010], 218 applied to some statements in the middle of a method. Single statements may be hard to select without explicit join points, yielding bloated and fragile pointcuts. Sequences of statements need to be extracted into a method so that they can be advised using pointcuts.

We support explicit announcement through a language construct called *closure join points*, a mechanism that allows programmers to explicitly mark any Java expression or sequence of statement as "to be advised". Closure join points are explicit join points that resemble labeled, instantly called closures, i.e., anonymous inner functions with access to their declaring lexical scope. Closure join points were first proposed independently of JPIs [Bodden 2011]; this work integrates both approaches.

Listing 2 shows the shopping-session example from Listing 1 adopted to the proposed syntax. Instead of using a pointcut in an **`exhibits`** clause, the programmer exposes a sequence of statements (lines 4–6) using a closure join point of the `CheckingOut` JPI. Section 3 gives the syntax and semantics of closure join points. For a further discussion of our design and implementation of closure join points we refer the interested reader to Bodden [2011]. Finally, note that an advice is oblivious to the fact whether join points are announced implicitly or explicitly; only their types matter.

2.5. Join Point Polymorphism

Join points raised explicitly through closure join points can only provide a single join point interface, the one specified in the closure's header. In the case of implicit invocation, however, JPIs are assigned to join points through pointcuts, and different pointcuts can overlap, i.e., match common join points. As we have seen, a class defines the pointcuts that expose certain join points in its execution, following a given JPI. For instance, in our running example, class `ShoppingSession` defines a pointcut that gives the type `CheckingOut` to all join points that are executions of the `checkOut` method. Be-

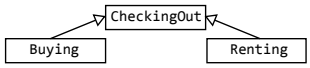

Fig. 2: Inheritance between Join Point Interfaces

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

Listing 3: Advice overriding

cause 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` join point, and could additionally be seen as a `LoggableEvent` join point (a JPI whose definition is left to the imagination of the reader).

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, increasing the potential for advice reuse. 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 3 shows an implementation of this additional rule using sub typing on JPIs. First, we declare the JPI `Renting` as extending `CheckingOut`. The semantics of this sub typing relationship implies that any join point of type `Renting` is also a join point of type `CheckingOut`. The **`extends`** clause defines how arguments are passed to super join points, similar to primary constructors in Scala [Odersky et al. 2008]. In the example, the first and third arguments of `Renting` join points become the first respectively second argument when this join point is seen through the `CheckingOut` interface.

This effect can be seen in the aspect `Discount`, which now declares two advices. The first one is the same as in the previous example. In general, it applies to all `CheckingOut` join points, and if this advice was the only advice in the aspect, then it would indeed execute also for join points of type `Renting`. In the example, however, the aspect defines a second advice specifically for `Renting`. In this case, an overriding semantics applies: the more specific advice *overrides* the first advice for all join points that are of type `Renting`. As a result, the first advice only executes for join points of other subtypes of `CheckingOut`, i.e., for `CheckingOut` itself, or for `Buying`. Section 3.3 describes advice dispatch in full detail.

3. CORE SEMANTICS OF JPIS

We begin this section by describing the syntax of JPIs, designed as an extension to AspectJ (Section 3.1). 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

TypeDecl ::= ... | *JPITypeDecl*

JPITypeDecl ::= "**jpi**" *JPISig* [*JPIExt*] ";"

JPISig ::= *Type ID* "(" [*ParamList*] ")" [*ThrowsList*]

JPIExt ::= "**extends**" "(" [*ArgList*] ")"

ClassMember ::= ... | *ExhibitDecl*

AspectMember ::= ... | *ExhibitDecl*

ExhibitDecl ::= "**exhibits**" *Type Name* "(" [*ParamList*] ")" [*ThrowsList*] ":" *PointcutExpr*

AdviceDecl ::= ... | *JPIAdviceDecl*

JPIAdviceDecl ::= [*Modifiers*] [*Type*] *AdviceKind ID* "(" [*ParamList*] ")" [*ThrowsList*] *Block*

Expr ::= ... | *ClosureJoinpoint*

StmtExpr ::= ... | *ClosureJoinpoint*

ClosureJoinpoint ::= "**exhibit**" *ID* "(" [*ParamList*] ")" *Block* "(" [*ArgList*] ")" | "**exhibit**" *ID Block*

Fig. 3: Syntactic extension for Join Point Interfaces (AspectJ syntax is shown in gray)

static type system that supports modular checking: we informally describe it in Section 3.2. Our proposal of JPIs also innovates over previous work in the way it supports join point polymorphism. The dynamic semantics of JPIs is described in Section 3.3.

3.1. Syntax

Figure 3 presents our syntactic extension to AspectJ to support join point interfaces and closure join points.

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 **exhibits** declaration specifies a join point interface signature and the associated pointcut expression that denotes the exhibited join points.

Further, our extension allows advices to be bound to JPIs. Opposed to normal advices, instead of directly referring to a pointcut expression, such 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.

Closure join points can be used in any place in which an expression can be used. A closure join point comprises the keyword **exhibit**, an identifier (the name of the JPI used to type the exposed join point), and a block, plus optionally a list of formal and actual arguments. If those lists are omitted, this is equivalent to specifying empty lists. The block effectively defines a lambda expression, while the formal parameter list defines its λ-bound variables. Opposed to a regular lambda expression, however, a closure join point must be followed by an actual parameter list: the closure is always immediately called; it is *not* a first-class object.

Our language extension is backwards compatible with AspectJ, i.e.,any valid AspectJ syntax is still valid in our language. This allows for a gradual migration of AspectJ programs to join point interfaces. It would be simple, though, to include a

"strict" mode that instructs the parser to disallow AspectJ's regular pointcut and advice definitions.

3.2. Static Semantics

We next describe the JPI type system, a key contribution of this work. The type system supports modular type checking of both aspects and classes; only knowledge of shared JPI declarations is required 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. Finally, we discuss type checking JPIs themselves, in particular considering JPI inheritance.

3.2.1. 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, but requires care. Each advice declares an advised JPI in its signature; the signature must directly correspond to the JPI declaration: both return and argument types must be exactly the same as those of the JPI. The crucial requirement here is that the typing of all types in the signature is *invariant*. To see why this is the case, assume that we allowed for the following advice definition, which declares that it cannot just accept `Customer` objects but instead objects of the super type `Person`:

```
void around CheckingOut(float price, Person p) {
    double factor = p.hasBirthday()? 0.95 : 1;
    //p = new Administrator();
    proceed(price*factor, p);
}
```

The contra-variant argument definition would not raise any issues with this particular advice definition. The problem is, however, that nothing would prevent the advice from including the commented statement `p = new Administrator()`, assuming that `Administrator` is a subtype of `Person`. If this statement were included, then the advice would invoke the original join point with an `Administrator` object. Assuming that `Administrator` is not a subtype of `Customer`, this code breaks crucial assumptions on the base-code side, usually manifested in the form of a `ClassCastException`. Conversely, if we allowed for co-variant arguments then the original join point could pass arguments to the advice that are of a type that the advice is not prepared to handle.

The opposite argument applies to return types. If we allowed for co-variant return types, then calls to **`proceed`** could return objects of a type that the advice is not prepared to handle. If we allowed for contra-variant return types, then the advice could return objects of a type that the base code is not prepared to handle.

The essence of the problem observed here is that the advice signature (and our JPI signature) is used to define both the join points intercepted by an advice, and the join point that an advice can call through **`proceed`**. Both calls have exactly opposite variance requirements, which means that the only sound typing must be invariant. In Section 5 we will see that in practice this can be too rigid; Section 6 will then explain how we relax this requirement with parametric polymorphism through generic types, an idea first proposed in the context of the StrongAspectJ language [De Fraine et al. 2008].

Checked exceptions also require some care. The advice must handle or declare to throw all exception types declared by the JPI it advises. We analyze the advice body to determine which types of exceptions the advice does or does not handle. The advice must not declare any additional exception, as this could lead to uncaught checked

exceptions on the side of the base code. It can, of course, declare fewer exceptions, or declare additional exceptions (e.g. `EOFException`) provided they are subtypes of expected exceptions (e.g. `IOException`) that are also declared.

The exceptions that are declared in the advice interface are typed invariantly with respect to the JPI. This strict invariance requirement for exceptions is required for the same reason as for return and argument types. Let us denote T_S the checked exception thrown at a join point shadow[1], T_I the exception type declared in the JPI, and T_A the type of checked exception thrown by the advice. If $T_S < T_I$[2], 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.

In our example, the JPI declaration for `CheckingOut` contains no checked exceptions. This imposes the following restrictions and guarantees:

— The base code can rely on the fact that an advice handling `CheckingOut` cannot throw any checked exceptions. The type system ensures that this is indeed the case.
— Likewise, an advice can rely on the fact that a join point of type `CheckingOut` cannot throw checked exceptions when the advice invokes it through a call to **`proceed`**. The type system forbid programmers from declaring join points that could throw any checked exceptions to be of type `CheckingOut` .

If needed, the programmer can relax these restrictions by declaring checked exceptions in the join point interface:

```
jpi void CheckingOut(float price, Customer cus) throws SQLException;
```

In this example, both the advised join point and the handling advice are allowed to throw `SQLExceptions`, but must also take care to handle (or forward) the exceptions appropriately.

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). JPIs *fully* specify the behavior of advised join points, thereby allowing safe and modular static checking of advice.

3.2.2. Type checking base code. On the other side of the contract is base code, which can exhibit join points. The base code must also obey the contract specified by join point interfaces. Part of this contract has to be fulfilled by the pointcut associated with the **`exhibits`** clause used for implicit announcement: the pointcut has to bind all the arguments in the signature, using pointcut designators such as **`this`**, **`target`** and **`args`**. To comply with our invariant semantics, those pointcuts must match invariantly, i.e., a pointcut such as **`this`**`(A)` must only match join points with declared type `A`. As we will explain in Section 4, this semantics is different from the one of AspectJ.

Because pointcuts do not account for return and exception types, our type system checks these types at each join point shadow matched by the pointcut associated with

[1]A join point shadow is the source expression whose evaluation produces a given join point [Masuhara et al. 2003; Hilsdale and Hugunin 2004].
[2]We use $<:$ for subtyping, and $<$ for strict (i.e., non-reflexive) subtyping.

the **exhibits** declaration. More precisely, the pointcut is matched against all join point shadows in the lexical scope of the declaring class. Whenever the pointcut matches a join point shadow, the type system checks that the return type of this shadow and the JPI coincide. If the shadow has a different return type, our type checker raises an error message stating that the selected join point shadow is incompatible with the JPI in question. Similarly, the type system validates that the declared exceptions of the shadow are the same as those of the JPIs; if they are not, the type checker raises an error. Here again, type compatibility is invariant.

One may wonder why we raise an error message when pointcuts select incompatible join point shadows and not simply restrict the matching process to exclude incompatible shadows instead. However, our chosen design is in line with AspectJ. In the following example, AspectJ raises an error because the **execution** pointcut selects a join point (in line 2), whose return type is incompatible with that of the around advice:

```
1 void around(): execution(int foo()) { ... }
2 int foo () { return 0; }
```

Our type checks at the join point shadows are the fundamental contribution of JPIs from the point of view of type checking base code. Most previous approaches, such as [Steimann et al. 2010], 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 type checks to weave time, or even worse, to runtime. With JPIs, conformance can be checked statically and modularly, prior to weaving.

Type checking closure join points. Type checking closure join points follows the Java static semantics for methods defined within inner classes. Code within a closure join point has access to its parameters, to fields from the declaring class and to local variables declared as **final**. Access to non-**final** local variables is forbidden because, by exposing a block of code to aspects in the form of a closure join point, aspects may choose to execute the closure in a different thread, or may choose to not execute the closure at all. In this case, write access to local variables may cause data races on those variables or may leave their value undefined if the write is never actually executed [Bodden 2011].

Other than that, we type check closure join points according to the same rules as implicitly-announced join points. To keep the code as concise as possible, closure join points do not define return and exception types; instead those are inferred from the join point interface declaration they reference. Because of this, those parts of the closure join point's signature are automatically invariant with respect to the referenced join point interface. Arguments are deliberately not inferred. This is because we want to allow users to give arguments within the closure join point their own local name, independent of what was specified in the JPI declaration and independent of the context in which the closure join point is declared. This is in line with method definitions and lexical scoping in Java. The independence from the closure join point's declaring context is important to distinguish the values of such variables that are declared in this context from the potentially altered values that an advice may pass as the closure join point's arguments. (As we will explain in Section 8, previous approaches did not treat this problem in a semantically sound manner.) For the types of those arguments, we check that the declaration in the closure join point's header obeys an invariant typing semantics. Further, we check that the body of a closure join point only throws checked exceptions of types that the referenced JPI declares, and conversely that the declaring context of the closure join point is prepared to handle (or declares to forward) all checked exceptions of these types.

3.2.3. Type checking JPIs. Primarily to facilitate reuse on the side of the aspect side, join point interfaces support a subtyping relationship. We employ type checking at the level of JPIs to ensure that subtyping relationships do not break type soundness. As we exemplified in Section 2, any JPI can declare to extend another JPI. During this process, the sub-JPI can add context parameters; JPIs thus support *breadth subtyping*. On the other hand, all arguments that do coincide have to be of the exact same type as the respective arguments in the JPI super type; JPIs do not support *depth subtyping*. This is due to the same reasons for which we use invariant argument typing in all other situations. For example, the following code would raise a type error, even if `B` were a subtype of `A`:

```
jpi void Base(A a);
jpi void Sub(B b) extends Base(b);
```

In addition, JPI subtypes must declare the same return type as their super-JPI, and must declare the same exceptions to be thrown.

3.3. Dynamic Semantics

The dynamic semantics of JPIs differ slightly from that of a traditional aspect language. Briefly, the traditional model is as follows [Wand et al. 2004]: 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 join points that are of these types, either using pointcuts (implicit announcement) or using closure join points (explicit announcement). The global environment contains aspects with their advices. With implicit announcement, conceptually, a join point representation is passed *only* to the pointcuts defined in the current class, at each evaluation step. If a pointcut matches, then the join point is tagged with the corresponding JPIs. Then, the advices that advise one of the tagged JPIs are executed.

In presence of join point polymorphism and inheritance among JPIs, it is interesting to ask 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 [Paepcke 1993] and MultiJava [Clifton et al. 2000]. 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 $S_i \not<: S_j$.

Aspect-oriented programming, like any publish-subscribe mechanism, inherently supports *multiple reactions* to a single 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[3]. In our case, we execute all the most-specific applicable advices, in the precedence order

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

imposed by regular AspectJ when multiple advices of a same aspect match the same join point [AspectJ Team 2003].

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 5, 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 5.2.3). 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 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 join point types [Steimann et al. 2010], where the same advice can surprisingly be executed several times for the same join point (we come back to this in Section 8).

4. IMPLEMENTATION BASED ON ASPECTJ

With this paper we provide a full implementation of join point interfaces as an extension to the AspectBench Compiler (abc) [Avgustinov et al. 2005]. Our implementation is maintained within abc's own code base.[4]

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`** clauses. To allow for maximal reuse of existing functionality in the abc compiler, we decided to implement our dispatch semantics through a transformation that computes for each such advice a single pointcut, based on the referenced JPI, its type hierarchy and the **`exhibits`** clauses 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`** clauses 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[5] 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

[4] abc can be downloaded at: `http://aspectbench.org/`

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

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 by inlining pointcuts. 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: if a pointcut does not match, there is no value that a could be bound to. Fortunately, abc supports a way to close such pointcuts so that the variables do not appear free any longer. This is done by rewriting a pointcut such as **this**(a) 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.

4.1. Invariant Pointcut Designators

As noted in Section 3, we must ensure that join point interfaces are invariantly typed. As it turns out, in AspectJ it is not straightforward to ensure invariant typing for arguments. The problem is that the standard pointcuts **this**, **target** and **args** come with a variant semantics. In the following AspectJ example, the pointcut matches the call to foo, although the declared type of foo's argument is Integer, not Number:

```
aspect A{
  public static void main(String[] args){
      foo(new Integer(2));
  }

  public static void foo(Integer a){}

  void around(Number n): call(void *(..)) && args(n) {
      proceed(new Float(3)); // will raise a ClassCastException
  }
}
```

As this example shows, the variant matching semantics of AspectJ is problematic: while the advice is well-typed in AspectJ, the call to **proceed** will cause a ClassCastException, as it calls foo with a Float argument.

One way to address this problem is to re-define the matching semantics of the **this**, **target** and **args** pointcuts such that they match a join point only if the declared type at the join point is exactly the same as the declared type used within the pointcut. This design, however, would give up backward-compatibility with AspectJ. Since our overall language design can be integrated as a fully backward-compatible extension, we opted for another design, such that existing AspectJ applications can be easily migrated to our language.

In our implementation, we support three additional invariant pointcuts, **This**, **Target** and **Args**. These pointcuts come with an invariant matching semantics. For instance, the pointcut **Args**(n) would not match in the example above, since n has type Number but the argument value at the join point shadow has declared type Integer. Due to the introduction of these novel pointcuts, the existing semantics of **this**, **target** and **args** remains unchanged. When programmers use one of those pointcuts within an **exhibits** clause, our compiler issues a warning, notifying the programmer that **This**, **Target** or **Args** should be used instead, as otherwise type soundness is not guaranteed.

4.2. Static Overloading

In addition to overriding, we also support static overloading of JPIs. For instance, one could write the following definition:

```
jpi void CheckingOut(float price, Customer cus) throws SQLException;
jpi void CheckingOut(float price, int amt, Customer cus) throws SQLException;
```

Similar to overloaded methods in Java, overloading is resolved completely at compile time. Overloaded JPI definitions are thus treated exactly as if they had different names, as in:

```
jpi void CheckingOut1(float price, Customer cus) throws SQLException;
jpi void CheckingOut2(float price, int amt, Customer cus) throws SQLException;
```

Therefore, overloading is just a means to allow the programmer to document that two JPIs are inherently related; it has no semantic impact.

4.3. Reuse of implementation for closure join points

Interestingly, our abc extension for join point interfaces extends and completely reuses the original implementation for closure join points [Bodden 2011]. abc uses the JastAdd [Ekman and Hedin 2007] compiler front-end, which allows for truly modular language definition. That way, our JPI extension to abc can modularly define how the extension for closure join points needs to be refined to match the correct syntax and semantics that they require when used in combination with join point interfaces.

5. A FIRST EVALUATION OF JPIS

We first discuss the benefits of join point interfaces based on previous studies. Then we report on a case study where we converted several AspectJ applications to use JPIs. This study brings a number of interesting insights related to join point polymorphism. Finally, we discuss the limitations revealed by this study, which motivated the extensions to JPIs that we will describe in the following sections.

5.1. 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 [Sullivan et al. 2010]) and join point types (JPTs [Steimann et al. 2010]), see Section 8. The benefits of XPIs and JPTs on modularity have been empirically demonstrated in previous editions of TOSEM [Sullivan et al. 2010; Steimann et al. 2010].

Recently, Dyer et al. [2012] report on an exploratory study of the design impact of different approaches to aspect interfaces. They consider the evolution of aspect-oriented software using different approaches, namely plain AspectJ, annotation-style, and quantified, typed events [Rajan and Leavens 2008]. While they do not consider JPIs in their study, their key results support our design of join point interfaces:

— The use of inter-type declarations is prevalent. By integrating JPIs in AspectJ, we inherit this mechanism for free.
— Quantification failure due to the need to advise join points that cannot be denoted using the pointcut language occurred on 5% of advised join point shadows. Explicit announcement addresses these cases nicely.
— The lack of quantification support with quantified typed events was problematic because it required to keep track of design rules that affect all members of a given class

(e.g. make all methods **`synchronized`**). Implicit announcement with **`exhibits`** clauses supports these cases.
— Almost a fifth of changes to pointcuts was due to the fragile pointcut problem. Any abstraction mechanism for denoting join points is immune to this issue.
— There were several instances where the fact that context information is restricted to join point specific attributes (**`this`**, **`target`**, **`args`**) was problematic, yielding additional complexity. Explicit-announcement approaches make it easy to expose arbitrary context information.[6]

To evaluate the importance of a sound treatment of checked exceptions in aspect interfaces, it is instructive to look at the lessons learned by Robillard and Murphy in an effort to design robust programs with exceptions [Robillard and Murphy 2000]. 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.

5.2. Join Point Polymorphism

To evaluate JPIs in practice, we have converted a set of existing AspectJ applications from the corpus of Khatchadourian et al. [2009]. These rewritten projects are available online at our project's web page. 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 the Law of Demeter programming rules [Lieberherr et al. 1988].

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 AspectJ Development Tools [AJDT 2012] and AspectMaps [Fabry et al. 2011]. 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.

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

LawOfDemeter contains the following pointcuts:

[6]Context-aware aspects [Tanter et al. 2006] are a general mechanism that supports arbitrary context information with implicit announcement; a similar flexibility is found in AspectScript [Toledo et al. 2010].

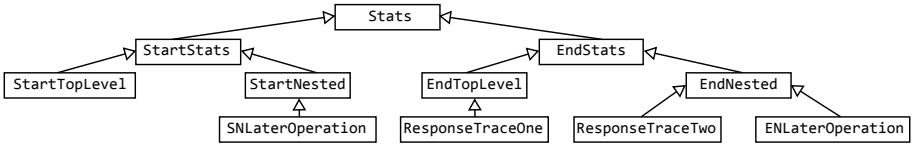

Fig. 4: A Potential Hierarchy of Join Point Interfaces in Glassbox.

```
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) extends MethodCallSite();
3 jpi Object SelfCall  (Object thiz, Object targt) extends MethodCall(thiz, targt);
```

The example shows that it is useful to allow subtypes to expose more arguments than their super types (a.k.a. breadth subtyping): `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.

5.2.2. 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. Here, we only show one of the most interesting examples in Figure 4. 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.

5.2.3. Advice Overriding. Glassbox showcases 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 $T_1 \dots T_4$, where T_1 is a super type of the three others. Due to overriding, the advice for T_1 never executes since it is always overridden. The solution would then be to refactor the program to move the advice for T_1 in a separate aspect. A simpler solution is to declare the advice for T_1 as **final**, which guarantees that it will not be overridden and therefore always execute.

5.2.4. 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 3.3). 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 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 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 afterward.

5.3. Limitations

The typing rules for JPIs currently enforce invariance on both return and exception types of join point interfaces (Section 3.2). 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 multiple JPIs (and advices) for the different types of

shadows. In an application like AJHotDraw, in which most pointcuts are very specific anyway, this rigidity is less problematic than in applications that rely on wide-matching pointcuts, such as LawOfDemeter. In AJHotDraw, we found that only 3 advices were concerned, and as a result the total number of advices in this project increased from 49 to 77, almost twice a much. In LawOfDemeter the number of advices was increased more than 10 times, going from 6 to 68! This observation strongly motivates the need for introducing more flexibility in the type system. In Section 6, we introduce *generic* interfaces to address this issue.

On a related front, aspects that use wide-matching pointcuts, like in LawOfDemeter, or dynamic analysis aspects like data race detectors and profilers, suffer from the class-local quantification approach followed by JPIs and others [Steimann et al. 2010]. These aspects require modifying many (if not all) classes to add the corresponding **`exhibits`** clauses. Since aspects are also extremely helpful in these application scenarios, it seems necessary to make some pragmatic decision with respect to the scope of quantification. We address this issue in Section 7 with a novel mechanism for controlled global quantification.

6. GENERIC JOIN POINT INTERFACES

As revealed by the case study, JPIs easily cause code duplication in aspect definitions when programmers implement highly-crosscutting aspects, such as monitoring, dynamic analysis, access control, exception handling, etc. The problem is that highly-crosscutting aspects have pointcuts that match a large number of shadows that typically have unrelated types. AspectJ deals with this issue by abandoning soundness. JPIs retain soundness, but at a cost: in the language design presented so far, programmers have to define different JPIs and advices for all the different shadow types. To mitigate this problem, we introduce *generic JPIs*. A generic JPI is a JPI that includes type parameters in its signature. Type parameters allow for the sound use of polymorphic advice. We first dive into a concrete example from the case study, then shows how a generic JPI solves the problem. We then briefly discuss the semantics and implementation of generic JPIs, before reporting more precisely on their impact on our case study.

6.1. Motivation

To illustrate the situation, let us consider the `Check` aspect defined in the LawOfDemeter (LoD) project, which checks that an object only sends messages to a certain set of closely related objects according to the Law of Demeter [Lieberherr et al. 1988]. Figure 5 shows an advice that registers the LoD violations within an aspect called `Check`. The important thing to note in this aspect definition is the extreme quantification used (**`call`**`(* *(..))`), which is typical of dynamic analysis aspects.

To migrate the `Check` aspect to use JPIs, conversely to what we suggested in Section 5.2.1, we *cannot* use a single JPI like the following:

```
1 jpi Object MethodCall(Object thiz, Object targt);
```

The reason is that, in order to ensure type soundness, such a JPI could only match join points where the actual types at the shadows are `Object`. As explained earlier, co-variant matching of shadows is unsound. This means that we need to define one JPI per possible combination of types for this, target and the return type! In the LoD project, just handling the `Check` aspect for the examples included in the project required us to write 21 JPIs:

```
1 jpi void    MethodCall_1(TemporalQueue thiz, TemporalQueue targt);
```

```
aspect Check {
        private IdentityHashMap objectViolations = new IdentityHashMap();

        public pointcut scope():
          !within(lawOfDemeter..*) && !cflow(withincode(* lawOfDemeter..*(..))) ;

        pointcut methodCalls(Object thiz, Object targt) :
           scope() && call(* *(..)) && this(thiz) && target(targt);

        after(Object thiz,Object targt): methodCalls(thiz,targt) {
                if (!ignoredTargets.containsKey(targt) &&
                    !Pertarget.aspectOf(thiz).contains(targt)) {
                        objectViolations.put(thisJoinPointStaticPart,
                                             thisJoinPointStaticPart);
                }
        }
}
```

Fig. 5: LoD Check aspect (excerpt)

```
aspect Check {
        void registerViolation(Object thiz, Object targt, JoinPoint.StaticPart jp) {
                if (!ignoredTargets.containsKey(targt) &&
                     !Pertarget.aspectOf(thiz).contains(targt)) {
                        objectViolations.put(jp, jp);
                }
        }
        // one advice per type combination...
        after MethodCall_1(TemporalQueue thiz, TemporalQueue targt) {
                this.registerViolation(thiz, targt, thisJoinPointStaticPart);
        }
        after MethodCall_2(TemporalQueue thiz, TQ_N targt) {
                this.registerViolation(thiz, targt, thisJoinPointStaticPart);
        }
        after MethodCall_3(TemporalQueue thiz, TemporalQueue targt) {
                this.registerViolation(thiz, targt, thisJoinPointStaticPart);
        }
        // etc.
}
```

Fig. 6: LoD Check aspect with (non-generic) JPIs

```
jpi int     MethodCall_2(TemporalQueue thiz, TQ_N targt);
jpi PQ_Node MethodCall_3(TemporalQueue thiz, TemporalQueue targt);
//etc.
```

In addition to this tedious and fragile list of JPI definitions, the `Check` aspect itself has to contain one advice for each such JPI. As Figure 6 shows, all advice bodies are exact copies (note that for **`after`** advice, the return type of the JPI is simply ignored). While this approach is "correct" in that all expected join points are matched,

```
aspect Check {
       // ... registerViolation ...
        <T, U> after MethodCall(T thiz, U targt){
            this.registerViolation(thiz, targt, thisJoinPointStaticPart);
       }
}
```

Fig. 7: LoD Check aspect with a generic JPI

and type soundness is preserved, it is highly cumbersome for programmers and, most importantly, it does not scale. As soon as one wants to apply the LoD aspects to other projects, more JPIs and advices have to be defined.

6.2. Defining and Using a Generic JPI

We therefore extended our syntax and type system to allow programmers to express a whole range of possible type combination through a single *generic* JPI:

```
<R, A, B> jpi R MethodCall(A thiz, B targt);
```

This generic interface abstracts away the specific types involved. Now the programmer can define a generic version of the `Check` aspect using a polymorphic advice (Figure 7). Note that the specific type variables used in the JPI and the corresponding advice do not need to match; they are independent abstract parameters.

Our support for generics includes type *bounds* as in Java. For instance, the following JPI characterizes join points where an argument is of a subtype of `Number`:

```
<T extends Number> jpi void JP(T arg);
```

In such a case, an **`exhibits`** clause on the base code side must also be generic, and it must use the same type bounds (if present). For instance:

```
class A {
  <N extends Number> exhibits void JP(N n) : call(void *(..)) && args(n);
  // ...
}
```

Note that this is much more flexible than the same definition using `Number` directly as the type of `n`. The invariant matching semantics means that only methods where the argument is specifically of type `Number` would match. Using the type parameter `N` makes it possible to match uniformly all join points where the argument is a subtype of `Number`.

6.3. Static Semantics

Generic JPIs are a simple extension of JPIs with basic support for bounded polymorphism. We only support upper bounds for type variables (declared with **`extends`**), since they are enough to handle the needs for polymorphism that we have observed in practice through the case study. We leave to future work the exploration of the other features found in object-oriented languages with parametric polymorphism (lower bounds, wild-cards, type constraints as in Scala, etc.).

The semantics of type checking in presence of parametric polymorphism is exactly the same as that of Java with generics, more precisely as defined in Featherweight

Generic Java [Igarashi et al. 2001]. A generic JPI is like a generic method signature. A generic advice is type-checked just like a generic method, where **`proceed`** is given the type of the JPI (modulo renaming of type variables). When checking base code, a type at a shadow matches a type variable in the JPI if it is a subtype of the bound of that variable. As explained above, this is where flexibility is gained, allowing generic JPIs to be practical in the face of wide crosscutting. Note that soundness is not compromised thanks to parametricity [Reynolds 1983]: only the bound of the type variable can be directly relied upon.

6.4. Implementation

We introduce generic JPIs in our AspectJ extension by reusing the capabilities of JastAdd to deal with Java 5 generics. This allows us to reuse all of the functionality needed to use type variables in our language and only focus on specific changes in the parser and in the static type system. Syntactically, to support generic advice, we allow type variables to be declared and used in JPIs, advice signature, **`exhibits`** clauses and pointcuts. Closure join points do not need type parameters, as they are actual instances of join points, binding to the type parameters of the corresponding join point interface.

6.5. Evaluation

	AspectJ	Non-generic JPI	Generic JPI
LoD	6	68	6
AJHotDraw	49	77	49

Table I: Number of advices defined in each version

To assess the impact of generic JPI, we re-implemented the AJHotDraw-JPI and LoD-JPI projects. As Table I shows, generic JPIs completely eliminate the problem of repeated advice declarations in both projects. While the number of advice declarations in the (non-generic) JPI version was considerably increased (more than 10x for LoD), the version using generic JPIs presents the exact same number of advices. This clearly validates the practical positive impact of extending JPIs to support parametric polymorphism.

7. CONTROLLED GLOBAL QUANTIFICATION

As revealed by the case study from Section 5, if JPIs are used in combination with highly-crosscutting aspects, developers may be forced to modify several (if not all) existing classes to introduce **`exhibit`** clauses. The problem is that such aspects use wide-matching pointcuts to capture several join points within different classes. To overcome this problem, we introduce *global pointcuts*, which can be attached directly to JPI definitions. Aspect programmers can use global pointcuts for *controlled global quantification*. The quantification is global, as it is evaluated over all classes in the system. At the same time the quantification is controlled because classes and aspects have the option to restrict or widen quantifications within themselves, or to seal themselves off against any kind of global quantification. We first revisit the motivating example of Section 6, then show how a global pointcut solves the problem. We then discuss the semantics and the implementation of global pointcuts. Finally, we report the impact of global pointcuts on our case study.

```
import jpis.*;

public class TemporalQueue extends PriorityQueue
{
  <R, T, I> exhibits R MethodCall(T thiz, I targt) :
    call(R *(..))
    && This(thiz)
    && Target(targt)
    && !within(lawOfDemeter..*)
    && !cflow(withincode(* lawOfDemeter..*(..)));
  ...
}

public class Repository extends Entity
{
  <R, T, I> exhibits R MethodCall(T thiz, I targt) :
    call(R *(..))
    && This(thiz)
    && Target(targt)
    && !within(lawOfDemeter..*)
    && !cflow(withincode(* lawOfDemeter..*(..)));
  ...
}
```

Fig. 8: Base code advised by the generic JPI version of Check aspect

7.1. Motivation

Let us consider the situation that a programmer wishes to apply the generic JPI version of the `Check` aspect (Section 6, Figure 7) to the following two arbitrarily chosen classes `TemporalQueue` and `Repository`. First, the programmer would introduce the **`exhibits`** clauses, as shown in Figure 8. We note that this approach is fragile and does not scale: programmers must modify every single class to introduce the **`exhibits`** clauses. Moreover, this redundancy is unnecessary, as in most cases all classes will bind the same JPI to the same pointcut expression. In the migration of LoD to its generic JPI version, this problem forced us to modify 21 of 23 classes to include the corresponding **`exhibits`** clauses.

the clauses.

7.2. Defining and Using a Global JPI

We therefore extended our syntax and type system to allow programmers to introduce globally defined templates for **`exhibits`** clauses, defined in the form of a *global pointcut*, attached to a JPI:

```
    <T, U, V> jpi T MethodCall(U thiz, V targt) :
        call(T *(..))
        && This(thiz)
        && Target(targt)
        && !within(lawOfDemeter..*)
        && !cflow(withincode(* lawOfDemeter..*(..)));
```

Those templates affect all existing classes. Global pointcuts use an inlining semantics: by default, i.e., if a class (or aspect) states nothing about the JPI `MethodCall`, then the class (or aspect) will automatically inherit an appropriate **`exhibits`** clause based on the global pointcut definition. Such implicitly introduced **`exhibits`** clauses are then treated exactly as if programmers had introduced them by hand. As we will next show, though, those implicitly defined **`exhibits`** clauses are just templates that every class (or aspect) can choose to refine.

7.3. Refining global pointcuts

The JPI `MethodCall` exposes every single method invocation to the `Check` aspect, which in turn detects whether such calls violate the Law of Demeter or not. The Law of Demeter states that classes should only access the public interface of other classes. In particular, they should not invoke methods on objects referenced through another class' public fields. But now consider a case where we use the class `Output` to print some messages to the standard output stream.

```
1  class Output {
2      public static void print(String message){
3          System.out.println("Message: " +message);
4      }
5
6      public static void print(float value){
7          System.out.println("Message: " +value);
8      }
9      ...
10 }
```

As can be seen, every call to `System.out.println` cause s the `Check` aspect to register a violation of the law. But such violations are not interesting to us. A naïve attempt to address this issue is to modify the global pointcut in the JPI, but this is fragile.

Instead, we propose a refinement mechanism that allows classes and aspects to control the way in which they are advised through global pointcuts: Whenever a class (or aspect) *does* define an **`exhibits`** clause for a JPI that also comprises a global pointcut, this **`exhibits`** clause *overrides* the one that would normally be implicitly inherited. At this point, the programmer can use a special primitive pointcut **`global`** to refer to the global pointcut, if needed.

```
1  class Output {
2
3      <T, U, V> exhibits T MethodCall(U thiz, V targt):
4          global(thiz,targt)
5          && !call(T java..*.*(..));
6      ...
7  }
```

Fig. 9: **`global`** pointcut designator

As an example, Figure 9 shows how the class `Output` can refine the global pointcut to restrict matching within itself.

7.4. Protecting from Global Quantification

When refining a global pointcut with an **exhibits** clause, programmers may provide no pointcut at all. In that case, the class or aspect is *sealed* against the global pointcut of the respective JPI. This approach to controlled global quantification is useful in that it allows programmers to refine pointcuts for JPIs that they are aware of. But in a code-evolution scenario, programmers may wish to protect classes and aspects from being advised altogether, even as new global pointcuts are introduced. To this end, we introduce a new modifier that allows programmers to mark those classes and aspects in the form of **sealed class** `Output {...}`. Sealing a class or aspect completely disables the effects of global pointcuts within this scope. Note, however, that even a sealed class can still opt to expose join points selectively by using **exhibits** clauses or closure join points.

7.5. Static Semantics

Global pointcuts are a simple extension of JPIs, also in terms of their semantics. Our type checks enforce that the **global** pointcut designator is only used in **exhibits** clauses that bind to a JPI declaring a global pointcut. The signature of the **global** pointcut is defined through the signature of the JPI that defines this pointcut. Our type-checking rules for classes, aspects and their exhibit clauses remain unchanged.

7.6. Implementation

We introduce global pointcuts to our implementation by modifying the parser, the static type system and by introducing some AST rewrites. Syntactically, we allow pointcut expression to be declared in JPIs. We further alter type checks to recognize the new built-in pointcut designator called **global** in the scope of an **exhibits** clause, and also allow **exhibits** clauses to be defined without any pointcut expression at all.

We then implement the correct runtime semantics for global pointcut by inlining global pointcuts into every aspect and class, replacing occurrences of the **global**`(..)` pointcut by the global pointcut definition. In case a class (or aspect) contains no refinement for a JPI such as **void** `JP()` defining a global pointcut, then we treat it as if the class (or aspect) actually defined an **exhibits** clause of the following form:

```
exhibit void JP(): global();
```

Figure 10 shows how our implementation treats such refinements through inlining within our previous example.

```
1  class Output {
2
3      <T, U, V> exhibits T MethodCall(U thiz, V targt):
4          call(T *(..))
5          && This(thiz)
6          && Target(targt)
7          && !within(lawOfDemeter..*)
8          && !cflow(withincode(* lawOfDemeter..*(..)));
9          && !call(T java..*.*(..));
10     ...
11 }
```

Fig. 10: Inlined exhibits clause which use **global**

7.7. Evaluation

	Generic JPI version	Global JPI version
LoD	130	0
AJHotDraw	46	46

Table II: Number of exhibits clauses defined in each version

To assess the impact of global pointcuts, we re-implemented the generic-JPI versions of both AJHotDraw and LoD with global pointcuts. As Table II shows, global pointcuts significantly decrease the amount of scattered **`exhibits`** clauses in projects such as LoD, which implement highly-crosscutting aspects. While the number of **`exhibits`** clauses in the (generic) JPI version of LoD was 130, the version using global JPIs in fact goes without any **`exhibits`** clauses. For application such as AJHotDraw, however, which comprise highly specific pointcuts, global quantification cannot help, as different classes must expose different join point shadows for matching to the respective aspects.

7.8. Discussion

There is a clear tension between global quantification and class-level **`exhibits`** clauses. On the one hand, **`exhibits`** declarations at the class level demand more code annotations but have the positive effect of allowing for truly local reasoning at the class level. On the other hand, global pointcuts allow programmers to eliminate many code annotations at the cost of losing the ability to reason about advising locally.

We believe—and our case study shows—that none of the two mechanisms is ideal for every aspects; highly generic aspects do benefit from global quantification. Many of them, like LoD, or more generally dynamic analyses, cause no harm, because they are only observing the base code execution. For other aspects that are meant to directly affect the base execution, global quantification may be the wrong choice. Our design reflects our intent to be pragmatic and support both families of aspects. With our approach, programmers can make the choice of which mechanism to use, in a sound and well-typed setting.

8. 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 [Gudmundson and Kiczales 2001]. In the AOP literature, many proposals have been formulated, some aiming at providing more abstract pointcut languages (e.g. [Gybels and Brichau 2003]), 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. A recent exhaustive treatment of this body of work and neighbor areas can be found in [Steimann et al. 2010].

Aspect-aware interfaces. In their ICSE 2005 paper, Kiczales and Mezini argue that when facing crosscutting concerns, programmers can regain modular reasoning by using AOP [Kiczales and Mezini 2005]. 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.

Crosscutting interfaces. Sullivan et al. [Sullivan et al. 2010] 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. Sullivan 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.

Open modules. In the same period, Aldrich formulated the first approach for language-enforced modularity, Open Modules [Aldrich 2005]. 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 [Ongkingco et al. 2006].

Join point types. Join point types (JPTs) [Steimann et al. 2010] are a further (and the most recent) step in the line of Open Modules. Also 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.

At first sight, JPTs are very similar to JPIs, and were actually the starting point of our work. The general idea of introducing a typed interface between base and aspects with support for both implicit and explicit announcement is the same, but the realization differs in a number of fundamental ways, which make JPIs both safe (while JPTs are not), and more flexible than JPTs.

First, as an example, consider Listing 4, which corresponds to the birthday discount example, implemented using JPT. As we can see, join point types do not resemble method signatures; rather they are data structures with mutable fields (line 13). This has a number of problems: most importantly, it makes advice and base code fragile, as both depend on the actual name of a JPT field. On the side of the base code, this dependency is even prohibitively strong: the example in Listing 4 only works because the programmer has named the local variables `price` and `c` equal to those names used in the join-point type definition. Our join point interfaces instead are like method signatures, so name dependencies for context information are avoided; matching of arguments happens purely by argument position, just like standard procedural abstraction.

Second, mutation of JPT fields can yield incorrect or even undefined semantics. In line 7, the code updates the variable `totalAmount`. The type checker for JPTs allows those updates also in the case where `totalAmount` is a local variable. But this is unsound: any aspect advising `CheckingOut` join points may opt to execute the original join point in another thread, thus causing a data race on the *local* variable `totalAmount`,

```
1  class ShoppingSession {
2    void checkingOut(final Item item, float price, int amount, Customer cus) {
3      //assume price==100 && c.hasBirthday()
4      exhibit new CheckingOut(price, c) {
5        sc.add(item, amount); //then here price==95...
6        cus.charge(price);
7        totalAmount += amount;
8      }; //... and here price==100 again
9    }
10 }
11
12 aspect BonusProgram {
13   joinpointtype CheckingOut{ float price; Customer c; }
14   void around(CheckingOut jp) {
15         if(c.hasBirthday())
16           jp.price = 0.95 * jp.price;
17     proceed(jp);
18   }
19 }
```

Listing 4: Shopping example with join point types (JPTs)

breaking the important guarantee of Java and AspectJ programs that local variables cannot cause data races. Or even worse, the aspect may choose not to **proceed** at all, in which case the value of `totalAmount` may be totally undefined. Bodden's earlier work on closure join points discusses those issues in even further detail [Bodden 2011].

Third, join point types are unsound because they do not specify return and exception types at join points. This means that both weave time and runtime errors can occur whenever aspects and base code do not coincidentally agree on these types. JPIs make these assumptions explicit and therefore ensure type safety.

Fourth, JPTs do not handle polymorphism properly. To briefly illustrate why, let us consider that we introduce two subtypes of `Buying` from our example of Section 2: `BuyingBestSeller` and `BuyingEcoFriendly`. With JPTs, whenever a book is bought that is both a bestseller and an eco-friendly print, the same `Discount` advice for `CheckingOut` is executed twice. This implies that any 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 really 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. Note that we did find an occurrence of this scenario in the case study; more precisely, in the Glassbox example described in Section 5.2.3. For all classes that implement only one of the two interfaces `ITC` and `EITC`—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. The dispatch semantics of JPIs avoids this problem: a given advice is guaranteed to execute *at most once* for any given join point (Section 3.3).

Fifth, Steimann's proposal is not symmetric in that aspects cannot exhibit join points on their own, to be advised by other aspects. The justification of this choice is to avoid infinite loops due to aspects advising themselves. As explained by Tanter [2010], this is however no solution. Avoiding infinite loops requires a dynamic semantic construct,

like execution levels. As a matter of fact, infinite loops can happen with JPTs[7]. Our proposal of join point interfaces is symmetric: aspects can exhibit join points. Integrating execution levels in AspectJ has already been done [Tanter et al. 2010] and would be helpful for JPIs as well, although this is an orthogonal concern.

Finally, the two additions to our design that were directly motivated by the case study, namely generic JPIs and global pointcuts, also contribute to making JPIs a much more practical approach to aspect interfaces. JPTs do not support type variables nor global quantification.

Ptolemy. Ptolemy [Rajan and Leavens 2008] supports explicit announcement of events (an idea initially proposed by Hoffman and Eugster [Hoffman and Eugster 2007]). Events are defined with event types. Event types 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. Originally, Ptolemy did not support event subtyping. It was recently extended with a form of subtyping, which allows for depth subtyping in event types [Fernando et al. 2012]; this is possible only because Ptolemy does not support the specification of alternative arguments to **`proceed`**, otherwise depth subtyping would be unsound, as explained earlier in this paper. Further, because Ptolemy only supports explicit announcement, emitted join points have a single most specific type, simplifying advice dispatch. Ptolemy supports behavioral contracts, called translucid contracts [Bagherzadeh et al. 2011], to specify and verify control effects induced by event handlers. These verification techniques go beyond more lightweight interfaces like Java interfaces and JPIs.

A major contribution of our work is to realize that the above proposals rely on *insufficiently expressive interfaces* to really allow separate development and modular type checking. As mentioned above, the contribution of our work has already been reflected in recent enhancements to the Ptolemy language.

EScala. EScala [Gasiunas et al. 2011] is an approach to modular event-driven programming in Scala, which, like JPTs and JPIs, also combines implicit and explicit events. EScala does not support **`around`** advice, so event definitions need not declare return types; exception types are missing, but this reflects the design philosophy of the Scala language. 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.

Type soundness and aspects. Soundness issues with the type system of AspectJ were first reported by Wand et al. [2004], although no solution was proposed. Jagadeesan et al. [2006] formulate a sound approach in which an advice type may depend on explicitly-declared type variables. Like in our approach, they use the same signature for **`proceed`** and the corresponding advice, and therefore require invariance as well. Some flexibility is regained because the type variables from the signatures can be instantiated for each join point. Due to parametricity, it is difficult to express arbitrary replacement advice.

MiniMAO_1 [Clifton and Leavens 2006] and StrongAspectJ [De Fraine et al. 2008] both consider different signatures for advice and **`proceed`**, thereby allowing more liberal pointcut/advice bindings while maintaining soundness. StrongAspectJ is more flexible in that it support signature ranges for pointcuts and type variables for generic advices. As recognized by the authors, however, the more expressive typing constructs of StrongAspectJ result in quite complicated syntax forms.

[7]Examples available online: http://pleiad.cl/research/scope/levels/iiia-loops

Aspectual Caml [Masuhara et al. 2005] is an aspect-oriented extension of OCaml. An interesting feature of Aspectual Caml is that it uses type information to influence matching, rather than for reporting type errors. More precisely, they infer the type of pointcuts from the associated advices, and only match join points that are valid according to these inferred types. It is unclear if the inference approach may be ported to an object-oriented language with subtype polymorphism.

All the above approaches are formulated in a traditional pointcut-advice setting, without relying on interfaces to uncouple base and aspect code. Our design of generic JPIs basically follows the proposal of Jagadeesan et al. This choice is pragmatic: while we have clear evidence of the interest of this approach for JPIs in our case study (Section 6), we have not faced a single case where the extra flexibility of StrongAspectJ was required. This may change in the future as we experiment with more programs. In any case, it is unclear how to provide the extra flexibility of StrongAspectJ while retaining a simple enough syntax. This is definitely an area where more study is needed.

Perspectives on quantification. While AspectJ allows global quantification (i.e. pointcuts can match join points that occur anywhere), Open Modules and JPTs have very different takes on quantification: 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, and obtain module-local quantification. Of course, this approach to modules is certainly not as well supported in Java as it would be in other languages, such as Newspeak [Bracha et al. 2010], where modules are objects, supported by a very flexible virtual class system.

In addition, in order to make it possible for AspectJ programmers that adopt JPIs to balance the quantification design trade-off, we have introduced global pointcuts (Section 7). This combination of local and global quantification, which preserves the possibility for a given class to be sealed from unwanted advising, is in itself a novel answer to the quantification design question.

9. CONCLUSION

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. The use of type variables in JPI definitions, along with support for controlled global quantification, allow for concise definitions of JPIs with minimal programmer effort. We have implemented JPIs as a publicly available AspectJ extension, and have rewritten several existing AspectJ programs to take advantage of JPIs. This study supports our major design choices.

Table III summarize the main features of our language extension and the design decisions that justify this language design. When defining the JPI mechanism, we al-

Feature	Purpose
Join point interfaces	Decoupling aspects from base code
JPIs as method signatures	Preserve procedural abstraction
Class-local pointcut matching	Limit pointcut fragility
Closure join points	Support explicit announcement
Invariant typing of arguments, return type and checked exceptions Invariant pointcuts	Preserve type soundness
Join point polymorphism	Support expressive modeling and handling of events
Parametric polymorphism	Enhance the flexibility of the type system
Controlled global pointcuts	Support wide quantification

Table III: Summary of features and their purpose

ways opted for type safety over flexibility. As we explained earlier, in a first approach, we opted for totally invariant typing, which is safe but sometimes too inflexible. Join point polymorphism, parametric polymorphism and global pointcuts restore the necessary flexibility without jeopardizing type soundness. As a result, we have designed a language that has strong typing guarantees, allows for modular reasoning, minimizes the amount of extra code required, and can accommodate a smooth transition path from plain AspectJ.

REFERENCES

AJDT 2012. Eclipse AspectJ Development Tools. http://www.eclipse.org/ajdt/.

ALDRICH, J. 2005. Open modules: Modular reasoning about advice. In *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP 2005)*, A. P. Black, Ed. Number 3586 in LNCS. Springer-Verlag, Glasgow, UK, 144–168.

ASPECTJ TEAM. 2003. The AspectJ programming guide. http://www.eclipse.org/aspectj/doc/released/progguide/.

AVGUSTINOV, P., CHRISTENSEN, A. S., HENDREN, L., KUZINS, S., LHOTÁK, J., LHOTÁK, O., DE MOOR, O., SERENI, D., SITTAMPALAM, G., AND TIBBLE, J. 2005. abc: An extensible AspectJ compiler. In *Proceedings of the 4th ACM International Conference on Aspect-Oriented Software Development (AOSD 2005)*. ACM, Chicago, Illinois, USA, 87–98.

BAGHERZADEH, M., RAJAN, H., LEAVENS, G. T., AND MOONEY, S. 2011. 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)*. ACM, Porto de Galinhas, Brazil, 141–152.

BerkelyDB 2010. Oracle Berkeley DB. http://www.oracle.com/technetwork/database/berkeleydb/.

BODDEN, E. 2011. Closure joinpoints: block joinpoints without surprises. In *Proceedings of the 10th ACM International Conference on Aspect-Oriented Software Development (AOSD 2011)*. ACM, Porto de Galinhas, Brazil, 117–128.

BRACHA, G., AHE, P., BYKOV, V., KASHAI, Y., MADDOX, W., AND MIRANDA, E. 2010. Modules as objects in newspeak. In *Proceedings of the 24th European Conference on Object-oriented Programming (ECOOP 2010)*, T. D'Hondt, Ed. Number 6183 in LNCS. Springer-Verlag, Maribor, Slovenia, 405–428.

CLIFTON, C. AND LEAVENS, G. T. 2006. MiniMAO$_1$: An imperative core language for studying aspect-oriented reasoning. *Science of Computer Programming 63*, 312–374.

CLIFTON, C., LEAVENS, G. T., CHAMBERS, C., AND MILLSTEIN, T. 2000. 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)*. ACM, Minneapolis, Minnesota, USA, 130–145.

DE FRAINE, B., SÜDHOLT, M., AND JONCKERS, V. 2008. StrongAspectJ: flexible and safe pointcut/advice bindings. In *Proceedings of the 7th ACM International Conference on Aspect-Oriented Software Development (AOSD 2008)*. ACM, Brussels, Belgium, 60–71.

DYER, R., RAJAN, H., AND CAI, Y. 2012. An exploratory study of the design impact of language features for aspect-oriented interfaces. In *Proceedings of the 11th International Conference on Aspect-Oriented Software Development (AOSD 2012)*, É. Tanter and K. J. Sullivan, Eds. ACM, Potsdam, Germany, 143–154.

EKMAN, T. AND HEDIN, G. 2007. The JastAdd extensible java compiler. In *OOPSLA*. 1–18.

FABRY, J., KELLENS, A., AND DUCASSE, S. 2011. Aspectmaps: A scalable visualization of join point shadows. In *Proceedings of 19th IEEE International Conference on Program Comprehension (ICPC2011)*. IEEE Computer Society Press, 121–130.

FERNANDO, R. D., DYER, R., AND RAJAN, H. 2012. Event type polymorphism. In *Proceedings of the 11th Workshop on Foundations of Aspect-Oriented Languages (FOAL 2012)*. ACM, Potsdam, Germany, 33–38.

FILMAN, R. E., ELRAD, T., CLARKE, S., AND AKŞIT, M., Eds. 2005. *Aspect-Oriented Software Development*. Addison-Wesley, Boston.

GASIUNAS, V., SATABIN, L., MEZINI, M., NÚÑEZ, A., AND NOYÉ, J. 2011. EScala: modular event-driven object interactions in Scala. In *Proceedings of the 10th ACM International Conference on Aspect-Oriented Software Development (AOSD 2011)*. ACM, 227–240.

GUDMUNDSON, S. AND KICZALES, G. 2001. Addressing practical software development issues in AspectJ with a pointcut interface. In *Proceedings of the Workshop on Advanced Separation of Concerns*.

GYBELS, K. AND BRICHAU, J. 2003. 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. ACM Press, Boston, MA, USA, 60–69.

HILSDALE, E. AND HUGUNIN, J. 2004. Advice weaving in AspectJ. In *Proceedings of the 3rd ACM International Conference on Aspect-Oriented Software Development (AOSD 2004)*, K. Lieberherr, Ed. ACM, Lancaster, UK, 26–35.

HOFFMAN, K. AND EUGSTER, P. 2007. 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, 63–72.

IGARASHI, A., PIERCE, B. C., AND WADLER, P. 2001. Featherweight java: a minimal core calculus for java and gj. *ACM Transactions on Programming Languages and Systems 23,* 3, 396–450.

INOSTROZA, M., TANTER, É., AND BODDEN, E. 2011. 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*. New Ideas Track.

JAGADEESAN, R., JEFFREY, A., AND RIELY, J. 2006. Typed parametric polymorphism for aspects. *Science of Computer Programming 63*, 267–296.

KHATCHADOURIAN, R., GREENWOOD, P., RASHID, A., AND XU, G. 2009. Pointcut rejuvenation: Recovering pointcut expressions in evolving aspect-oriented software. In *International Conference on Automated Software Engineering (ASE 2009)*. IEEE/ACM, 575–579.

KICZALES, G. AND MEZINI, M. 2005. Aspect-oriented programming and modular reasoning. In *Proceedings of the 27th international conference on Software engineering (ICSE 2005)*. ACM, St. Louis, MO, USA, 49–58.

LADDAD, R. 2003. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Press.

LIEBERHERR, K., HOLLAND, I., AND RIEL, A. 1988. Object-oriented programming: An objective sense of style. In *Proceedings of the 3rd International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 88)*, N. Meyrowitz, Ed. ACM, San Diego, California, USA, 323–334. ACM SIGPLAN Notices, 23(11).

MASUHARA, H., KICZALES, G., AND DUTCHYN, C. 2003. A compilation and optimization model for aspect-oriented programs. In *Proceedings of Compiler Construction (CC2003)*, G. Hedin, Ed. LNCS Series, vol. 2622. Springer-Verlag, 46–60.

MASUHARA, H., TATSUZAWA, H., AND YONEZAWA, A. 2005. Aspectual Caml: an aspect-oriented functional language. In *Proceedings of the 10th ACM SIGPLAN Conference on Functional Programming (ICFP 2005)*. ACM, Tallin, Estonia, 320–330.

ODERSKY, M., SPOON, L., AND VENNERS, B. 2008. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, USA.

ONGKINGCO, N., AVGUSTINOV, P., TIBBLE, J., HENDREN, L., DE MOOR, O., AND SITTAMPALAM, G. 2006. Adding open modules to aspectj. In *Proceedings of the 5th ACM International Conference on Aspect-Oriented Software Development (AOSD 2006)*. ACM, Bonn, Germany, 39–50.

PAEPCKE, A., Ed. 1993. *Object-Oriented Programming: The CLOS Perspective*. MIT Press.

RAJAN, H. AND LEAVENS, G. T. 2008. Ptolemy: A language with quantified, typed events. In *Proceedings of the 22nd European Conference on Object-oriented Programming (ECOOP 2008)*, J. Vitek, Ed. Number 5142 in LNCS. Springer-Verlag, Paphos, Cyprus, 155–179.

REYNOLDS, J. C. 1983. Types, abstraction, and parametric polymorphism. In *Information Processing 83*, R. E. A. Mason, Ed. Elsevier, 513–523.

ROBILLARD, M. AND MURPHY, G. 2000. Designing robust Java programs with exceptions. In *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '00/FSE-8)*. 2–10.

STEIMANN, F. 2006. 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)*. ACM, Portland, Oregon, USA, 481–497.

STEIMANN, F., PAWLITZKI, T., APEL, S., AND KÄSTNER, C. 2010. Types and modularity for implicit invocation with implicit announcement. *ACM Transactions on Software Engineering and Methodology 20,* 1, 1–43.

STOERZER, M. AND GRAF, J. 2005. Using pointcut delta analysis to support evolution of aspect-oriented software. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*. 653–656.

SULLIVAN, K., GRISWOLD, W. G., RAJAN, H., SONG, Y., CAI, Y., SHONLE, M., AND TEWARI, N. 2010. Modular aspect-oriented design with XPIs. *ACM Transactions on Software Engineering and Methodology 20,* 2.

TANTER, É. 2010. Execution levels for aspect-oriented programming. In *Proceedings of the 9th ACM International Conference on Aspect-Oriented Software Development (AOSD 2010)*. ACM, Rennes and Saint Malo, France, 37–48.

TANTER, É., GYBELS, K., DENKER, M., AND BERGEL, A. 2006. Context-aware aspects. In *Proceedings of the 5th International Symposium on Software Composition (SC 2006)*, W. Löwe and M. Südholt, Eds. LNCS Series, vol. 4089. Springer-Verlag, Vienna, Austria, 227–242.

TANTER, É., MORET, P., BINDER, W., AND ANSALONI, D. 2010. Composition of dynamic analysis aspects. In *Proceedings of the 9th ACM SIGPLAN International Conference on Generative Programming and Component Engineering (GPCE 2010)*. ACM, Eindhoven, The Netherlands, 113–122.

TOLEDO, R., LEGER, P., AND TANTER, É. 2010. Aspectscript: Expressive aspects for the web. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development (AOSD 2010)*. ACM, 13–24.

WAND, M., KICZALES, G., AND DUTCHYN, C. 2004. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Transactions on Programming Languages and Systems 26,* 5, 890–910.