Types and Modularity for Implicit Invocation with Implicit Announcement

FRIEDRICH STEIMANN AND THOMAS PAWLITZKI
Fernuniversität in Hagen
SVEN APEL
Universität Passau
and
CHRISTIAN KÄSTNER
Universität Magdeburg

Through implicit invocation, procedures are called without explicitly referencing them. Implicit announcement adds to this implicitness by not only keeping implicit which procedures are called, but also where or when — under implicit invocation with implicit announcement, the call site contains no signs of that, or what it calls. Recently, aspect-oriented programming has popularized implicit invocation with implicit announcement as a possibility to separate concerns that lead to interwoven code if conventional programming techniques are used. However, as has been noted elsewhere, as currently implemented it establishes strong implicit dependencies between components, hampering independent software development and evolution. To address this problem, we present a type-based modularization of implicit invocation with implicit announcement that is inspired by how interfaces and exceptions are realized in JAVA. By extending an existing compiler and by rewriting several programs to make use of our proposed language constructs, we found that the imposed declaration clutter tends to be moderate; in particular, we found that for general applications of implicit invocation with implicit announcement, fears that programs utilizing our form of modularization become unreasonably verbose are unjustified.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features – abstract data types, polymorphism, control structures.
General Terms: Design, Languages
Additional Key Words and Phrases: implicit invocation; event-driven programming; publish/subscribe; aspect-oriented programming; modularity; typing.

1. INTRODUCTION

Implicit invocation, which was first discussed by [Reiss 1990; Sullivan and Notkin 1990; Sullivan and Notkin 1992], is both an architectural style and a programming paradigm (the latter also known as *event-driven programming* (EDP) or *publish/subscribe* (P/S) [Eugster et al. 2003]). Garlan and Shaw have characterized it succinctly as follows:

> The idea behind implicit invocation is that instead of invoking a procedure directly, a component can announce (or broadcast) one or more events. Other components in the system can register an interest in an event by associating a procedure with the event. When the event is announced the system itself invokes

Authors' addresses: F. Steimann and T. Pawlitzki, Lehrgebiet Programmiersysteme, Fakultät für Mathematik und Informatik, Fernuniversität in Hagen, D-58084 Hagen; S. Apel, Lehrstuhl für Programmierung, Fakultät für Informatik und Mathematik, Universität Passau, D-94030 Passau; C. Kästner, Institut für Technische und Betriebliche Informationssysteme, Fakultät für Informatik, Universität Magdeburg, D-39016 Magdeburg.

Table I. Terminology, rough equivalences

EVENT-DRIVEN PROGRAMMING, PUBLISH/SUBSCRIBE	ASPECT-ORIENTED PROGRAMMING, THIS PAPER
event	join point
event handler	advice
event type	join point type
publishes declaration	exhibits declaration
subscribes declaration	advises declaration
implicit announcement	pointcut, join point type predicate

> all of the procedures that have been registered for the event. Thus an event announcement implicitly causes the invocation of procedures in other modules. [Garlan and Shaw 1994, p. 9]

A special form of implicit invocation is *implicit invocation with implicit announcement* of events (hereafter abbreviated as IIIA), in which events are not published through a dedicated statement, but are instead specified declaratively. IIIA permits "events to be announced as a side effect of calling a given procedure" [Garlan and Scott 1993], which is considered "attractive because it permits events to be announced without changing the module that is causing the announcement to happen" [Notkin et al. 1993]. According to [Garlan and Scott 1993; Notkin et al. 1993], prominent applications of IIIA are database triggers (allowing the interception of database operations and their enhancement with stored procedures [Eswaran 1976]) and wrapper functions in the Common Lisp Object System (CLOS) [Bobrow et al. 1988].

Aspect-oriented programming (AOP) [Kiczales et al. 1997; Elrad et al. 2001] can be viewed as a contemporary form of IIIA [Xu et al. 2004]. Indeed, the most popular AOP language to date, ASPECTJ, has a powerful, declarative pointcut language that allows one to select from certain points of execution in a program, called join points, those with which certain events can be associated.[1] By binding pointcut expressions to methods called advice, implicit invocation of these methods takes place whenever the corresponding pointcut fires (matches). The announcement of the corresponding event can therefore be considered implicit. Table I gives an overview of how the concepts of the event world (IIIA) and AOP relate.

More recently, concerns have been raised that IIIA à la AOP compromises modularity by establishing a strong, implicit coupling between the components of a system [Gudmundson and Kiczales 2001; Clifton and Leavens 2003; Rajan and Sullivan 2003; Xu et al. 2004; Störzer and Graf 2005; Sullivan et al. 2005; Aldrich 2005; Dantas and Walker 2006; Griswold et al. 2006; Ongkingco et al. 2006; Steimann 2006]. Especially the absence of explicit interfaces, or other hints in the places where behaviour may get changed, is thought to hamper independent development. While similar concerns had already been raised for implicit invocation alone, namely that "when a component announces an event, it has no idea what other components will respond to it" and that "reasoning about correctness can be problematic, since the meaning of a procedure that announces events will depend on the context of bindings in which it is invoked" [Garlan and Shaw 1994, p. 10], it should be clear that all these objections are in stark contrast to the expected benefit of IIIA, namely the easing of software evolution.

To alleviate the modularity problems in AOP, several mechanisms have been proposed. One breed of works suggest improvements in pointcut languages that aim at rais-

[1] The set of possible join points in a program is determined by the so-called join point model of the aspect language. ASPECTJ also offers static introductions (injection of members into classes), which we do not consider here.

ing the level of abstraction of joint point specifications, thereby decoupling the implementations of the base program and its aspects (e.g., pattern-based pointcuts [Eichberg et al. 2004], structural and behavioral property-based pointcuts [Gybels and Brichau 2003; Masuhara and Kawauchi 2003; Rho et al. 2006], test-based pointcuts [Sakurai and Masuhara 2008]), and model-based pointcuts [Kellens et al. 2006]. Another breed, more in line with our own work, proposes various forms of interfaces that make the coupling between aspects and the advised code more explicit (e.g., the aspect-implied interfaces of [Kiczales and Mezini 2005], the crosscutting interfaces of [Griswold et al. 2006], or the open modules of [Aldrich 2005; Ongkingco et al. 2006]). However, all of these solutions have drawbacks: they either require a whole-program analysis, or they rely on conventions that cannot be enforced by the compiler, or they merely state dependencies, without achieving greater decoupling. Also, they all introduce language constructs that do not align well with the other constructs of their host language, including its type system.

1.1 Contribution

In this paper, we present a simple solution to the problems of IIIA that restores full modularity of involved components. It evolved out of our own prior work on avoiding accidental recursion in ASPECTJ by introducing type levels [Forster and Steimann 2006; Bodden et al. 2006], and of our criticism of AOP as well as the solutions to its problems as suggested in the literature to date [Steimann 2006]. Our approach is based on the novel concepts of *join point types* as the types of events that can be implicitly announced, and *polymorphic pointcuts* as their intensional specifications that are defined as parts of the classes exhibiting join points. Our solution, which we present as an ASPECTJ-based extension to the JAVA programming language, blends naturally with JAVA's native programming concepts; in particular, it bears some similarities with its type-based notions of interfaces and exceptions. More concretely:

- We interpret join points as runtime instances of user-declared join point types, with fields of join point types representing the context of a join point instance.
- We interpret pointcuts as the type predicates (characteristic functions) of these join point types.
- We require that classes exhibit join points explicitly, as declared by an `exhibits <join point type>` clause.
- We define pointcuts polymorphically by requiring classes that declare to exhibit join points of a certain type, to define their branch of the corresponding pointcut (type predicate) locally. The complete pointcut is thus defined as a disjunction of its class-local branches.
- We allow the explicit creation of join points at runtime via an `exhibit new <join point type constructor> {<statement>}` expression. This is useful in cases in which a suitable pointcut is difficult or impossible to formulate using the given point
- We make the dependencies of aspects explicit, by requiring them to declare — through an `advises <join point type>` clause — instances of which join point types they intend to advise.

Thus, our join point types are like JAVA interfaces (analogies in parentheses)

- in that they are abstract, i.e., provide no instances of their own, but take them from the exhibiting (implementing) classes;
- in that they specify *what* the exhibiting (implementing) classes must provide, namely the values of the fields that are declared in the join point type, while leaving the *how*, the pointcuts establishing the bindings to the context, to the classes; and
- in that they allow the creation of anonymous inner join point types (anonymous inner classes) via `exhibit new <join point type constructor> {<statement>}`.

As a result, each class can specify the sets of join points it exhibits individually by providing its own intensional specification of the join point type (the class-local pointcut), and the extension of each join point type is the union of the sets of join points of that type as specified by each class. This is analogous to interfaces, which let classes define the method implementations individually, and whose extension is the union of those of its implementing classes.

At the same time, our join point types are like exceptions (analogies in parentheses)

- in that their instances may either come into existence at some implicitly specified point of program execution within the lexical scope of the `exhibits` (`throws`) clause, or are explicitly created with an `exhibit new <join point type constructor> {<statement>}` expression (`throw new <exception constructor>` expression); and
- in that their occurrence is handled in some place remote from, and unknown to, where they occurred.

Most strikingly, our dealing with join points resembles dealing with exceptions in that it avoids code tangling, but not scattering — each scope in which a join point may occur must be explicitly marked with the corresponding join point type. While this may raise fears that programs are becoming impracticably verbose (when compared to IIIA without explicit interfaces), first evidence we have collected by applying our approach to several programs suggests that the imposed "declaration clutter" is moderate, and likely outweighed by better readability and increased safety against programming errors.

1.2 Outline

The remainder of this paper is organized as follows. We begin with an introductory example which demonstrates the problem we are attacking, namely the lack of explicit interfaces between the code hosting IIIA and the code being invoked. In Section 3 we introduce the basic concepts of our solution, namely join point types and polymorphic pointcuts. This solution naturally extends to explicit join points interpreted as anonymous inner subtypes, which in turn leads to subtyping of join point types. Section 4 presents syntax and semantics of our conception of IIIA by describing its compiler. Section 5 summarizes our findings obtained by rewriting two mid-size applications, namely BERKELEY DB and JHOTDRAW, to IIIA. A comprehensive discussion of related work completes our contribution.

One further remark before we begin. This paper is at the intersection of OOP, EDP (or P/S), and AOP. This imposes a terminological problem, namely which labels to use for the concepts we rely on. Since we are using the join point model of ASPECTJ and also relevant parts of its pointcut language to specify IIIA, we decided to stick with the jargon of AOP, ASPECTJ in particular. For readers unfamiliar with AOP and better acquainted with EDP, Table I should help through the paper.

2. A MOTIVATING EXAMPLE

We develop our proposed language for IIIA step by step, resorting to a simple example. The example consists of a class `ShoppingSession` and three referenced classes `ShoppingCart`, `Invoice`, and `Log`. The referenced classes all offer methods for adding an amount of items, the only difference being that `Invoice` takes a customer's personal rebate into account, which is why its `add` method receives the customer as an additional parameter. The corresponding JAVA program is shown in Fig. 1.

```
01  package application;
02
03  class ShoppingSession {
04    ShoppingCart sc = new ShoppingCart();
05    Invoice inv = new Invoice();
06    Log log = new Log("buys");
07    Customer cus = customerLogOn();
08
09    void buy(Item item, int amount) {
10      sc.add(item, amount);
11      inv.add(item, amount, cus);
12      log.add(item, amount);
13    }
14  }
15
16  class ShoppingCart {
17    void add(Item item, int amount) {…}
18  }
19
20  class Invoice {
21    void add(Item item, int amount, Customer cus) {…}
22  }
23
24  class Log {
25    void add(Item item, int amount) {…}
26  }
```

Fig. 1. A standard shopping cart example.

```
01  package aspects;
02
03  aspect BonusProgram {
04
05    pointcut buying(Item item, int amount):
06      execution(* *.add(Item, int)) && args(item, amount);
07
08    void around (Item item, int amount): buying(item, amount) {
09      if (item.category == Item.BOOK)
10        amount += amount / 2;
11      proceed(item, amount);
12    }
13  }
```

Fig. 2. Aspect extending the shopping cart application with additional behaviour.

Now assume that after the design of the classes has been finalized, the marketing department wants to have installed a temporary customer bonus program “buy 2 books, get 1 for free”. Using ASPECTJ, this added behaviour can be realized, without changing the original classes, by installing an aspect `BonusProgram` (Fig. 2) which adapts the amount of books for all `add` transactions except that for `Invoice`. It does so by providing a (named) pointcut[2], `buying` (lines 5–6), which specifies the condition that leads to the implicit invocation of the advice (lines 8–12). Note that the specification provided by the pointcut is highly economical in that it specifies an open number of locations in the source code (here lines 10 and 12 in Fig. 1), a property sometimes referred to as *quantification* [Filman and Friedman 2004].

The problem with this approach is that only the aspect `BonusProgram` contains hints that, and where or when, implicit invocation takes place. From a software engineering perspective this poses a serious modularity issue: while `BonusProgram` implicitly specifies on what it depends (through the pointcut `buying` it defines), the targets `ShoppingCart` and `Log` contain no hints of this coupling, a property referred to as *obliviousness* in

[2] A pointcut is a predicate that selects from a set of join points (i.e., points in the execution of a program) those that are considered to be worth noting, so that advice can be called (the implicit invocation, implicitly announced by way of the predicate evaluating to true).

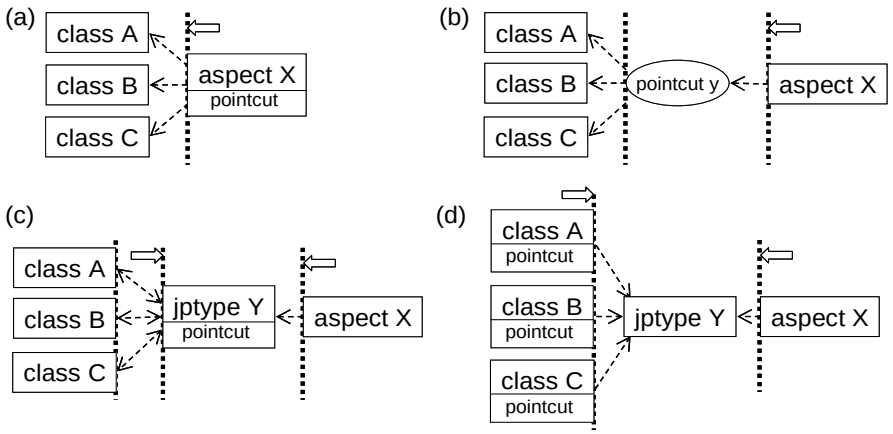

Fig. 3. From standard aspects to typed and modular IIIA (dashed arrows indicate referencing and change dependency, vertical dotted bars represent interfaces, and hollow arrows indicate the direction from which they are programmed against). (a) Aspect with local pointcut. (b) Same aspect with pointcut moved in proximity of targets. (c) Pointcut encapsulated by a join point type (abbreviated as jptype) and classes declaring to exhibit corresponding join points (Section 3.1). (d) Pointcut split and branches moved into targets ("polymorphic pointcut"; Section 3.2).

[Filman and Friedman 2004]. In particular, the lack of an explicit interface on the side of the target means that whenever one wishes to change the implementation of that target, one does not know which interfaces to respect. This situation is shown in Fig. 3 (a).

To illustrate this problem for the case of ASPECTJ, suppose that after installation of the `BonusProgram` aspect it is discovered that the log needs a customer entry (changes highlighted):

```
class Log {
  void add(Item item, int amount, Customer cus) {…}
}

class ShoppingSession {
  …
  void buy(Item item, int amount) {
    …
    log.add(item, amount, cus);
  }
}
```

This change breaks the `buying` pointcut from above, which no longer matches `Log`'s `add` method. Although this can be fixed by adapting the pointcut as in

```
pointcut buying(Item item, int amount):
  (execution(* *.add(Item, int)) && args(item, amount)) ||
  (execution(* Log.add(Item, int, Customer)) && args(item, amount,..));
```

or similar (but note that the new pointcut must not match `Invoice.add` accidentally) nothing in `Log` informs the programmer of this necessary change. The untoward effect this has on modularity (including modular reasoning) and independent development has been discussed, e.g., in [Gudmundson and Kiczales 2001; Clifton and Leavens 2003; Rajan and Sullivan 2003; Xu et al. 2004; Störzer and Graf 2005; Sullivan et al. 2005; Aldrich 2005; Dantas and Walker 2006; Griswold et al. 2006; Ongkingco et al. 2006; Steimann 2006].

Modularity problems are usually solved through the introduction of interfaces, i.e., "shared boundaries across which information is passed" [Geraci 1991]. In our example, boundaries are shared between classes `ShoppingCart` and `Log` on the one side and the aspect `BonusProgram` on the other, and the information passed consists of the parameters

`Item item` and `int amount`. However, declaration of this interface remains implicit in `BonusProgram` (it can be derived from the pointcut named `buying`), and is completely absent from the classes.

A number of attempts have been undertaken to tackle this problem by introducing more explicit interfaces.[3] So-called aspect-aware interfaces [Kiczales and Mezini 2005] annotate class members with the aspects they are advised by, and aspects with the class members they advise. However, rather than letting a designer specify these interfaces upfront, they are computed from the aspects and the classes after a system has been composed, and consequently change when the composition is changed. The support for independent development and reuse (primary purposes of modules) is therefore rather weak. This problem is avoided by the approach of Open Modules [Aldrich 2005; Ongkingco et al. 2006], by adding a new module construct to the base language whose interface declares the pointcuts the encapsulated program entities expose. Aspects then depend on the pointcuts declared in these interfaces, but nevertheless depend on concrete pointcuts, which may need to be changed when the base program changes. Coupling is therefore still strong and independent development still compromised. So-called crosscut programming interfaces (XPIs) [Sullivan et al. 2005; Griswold et al. 2006] rely on design rules to specify the requirements for classes exposing join points; however, since most design rules cannot be enforced automatically, XPIs also largely rely on pointcut expressions that classes must observe and aspects depend upon. To summarize, the decoupling achieved by any of these approaches does not go beyond what is shown in Fig. 3 (b): aspects depend on pointcuts that are defined external to them, and pointcuts depend on classes whose join points they are to specify. By contrast, our goal is to let classes and aspects both depend on a common interface between them, and to leave the interface completely independent of both (Fig. 3 (d)).

3. JOIN POINT TYPES AND POLYMORPHIC POINTCUTS

Our first step to achieve the modularity we envision is to make the shared boundary and the information passed explicit, on both sides. We do this through the introduction of join point types.

3.1 Join point types

Analogous to typed P/S [Eugster 2007] and also to JAVA's type-based exception handling, we interpret join points as typed events and introduce *join point types* as first class constructs that serve to specify the interface between classes exhibiting join points and aspects handling them (Fig. 3 (c)). In the case of our example, we define the following join point type:

```
joinpointtype Buying {
  Item item;
  int amount;
  pointcut execution(* *.add(Item, int, ..)) && args(item, amount,..);
}
```

This type gets instantiated every time a join point covered by its pointcut gets executed in the program. The resulting join point type instance is a record in memory whose fields are set to the parameters of the context in which the join point occurs, as prescribed by the pointcut. (In the given example, the first actual parameter of an executed `add` method is assigned to `item`, and the second is assigned to `amount`.) Because it characterizes the nature of its instances, we think of the pointcut as the *type predicate* (or *characteristic function*, i.e., the intensional specification) of the join point type. Since there is only one

[3] We look only at the most prominent examples here. Section 6 contains a comprehensive discussion of related work.

pointcut definition (type predicate) per join point type, the join point type's name identifies it uniquely so that it may be considered implicitly named. In Section 3.2, we will move the pointcut out of the join point type definition into the classes in which it applies; there, it will be named by the join point type to which it belongs.

Join point types like the above let us declare interfaces (boundaries and information passed) between classes and aspects. In our example, we add the following clauses to make the interfaces explicit:

```
class ShoppingCart exhibits Buying {…}
class Log exhibits Buying {…}
aspect BonusProgram advises Buying {…}
```

The `exhibits` clauses mark the *caller* side of implicit invocation, and the `advises` clause the *called*. This may appear counter to intuition, since the aspect (as the "advisor") seems to be the active, and the class (the "advised") the passive, and indeed the aspect depends on the classes it advises and not vice versa; however, such reversal of dependency is not unusual for interfaces (so-called *enabling interfaces* [Steimann and Mayer 2005]).

Definition of the join point type `Buying` as above allows us to rewrite the aspect `BonusProgram` as follows (changes highlighted):

```
aspect BonusProgram advises Buying {
  void around (Buying jp) {
    if (jp.item.category == Item.BOOK)
      jp.amount += jp.amount / 2;
    proceed(jp);
  }
}
```

The advice is now parameterized by the variable `jp` of type `Buying`, which holds the join point instance that led to the implicit invocation of the advice and which is also the (sole) argument of the `proceed` statement.[4] Its fields can be written, in which case the changed values replace those of the join point for the duration of the `proceed` statement. Following the semantics of JAVA, writing to the fields can be prevented by declaring them `final` in the join point type (cf. the discussion of spectators and assistants in Section 6).

Generally, a class can exhibit, and an aspect can advise, arbitrarily many join point types. Currently, a class cannot possess an `advises` clause, nor can an aspect possess an `exhibits` clause. This is to avoid self-reference and the resulting problems addressed in [Forster and Steimann 2006; Bodden et al. 2006], but may be changed in future work. For reasons given below (Section 3.5), join point type exhibition is not inherited by the subclasses of an exhibiting class. A class declaring to exhibit a join point type but providing no join points statically matching its pointcut (having no join point shadows) presents no error; yet, the compiler issues a warning in such cases indicating that either pointcut or class definition may be inappropriate. Note that it is impossible to require that a class always produces a join point for every join point type it exhibits, since generally, the occurrence of a join point may depend on dynamic conditions the satisfiability of which the compiler has no way of checking. However, the compiler does make sure that a pointcut definition is present for each join point type exhibited, and that it binds all fields of its join point type (see Section 4.2).

Our use of join point types improves modularity in that maintainers of a class wishing to make changes to it can consult the definitions of the join point types the class exhibits, and observe the pointcuts specified there (Fig 4). However, seemingly harmless changes such as the one performed at the end of Section 2, and even refactorings believed to not change program behaviour at all, may require adaptation of the pointcut, establishing a

[4] Note how this is reminiscent of the `catch` statement of exception handling in JAVA; this is deliberate (cf. Section 1.1).

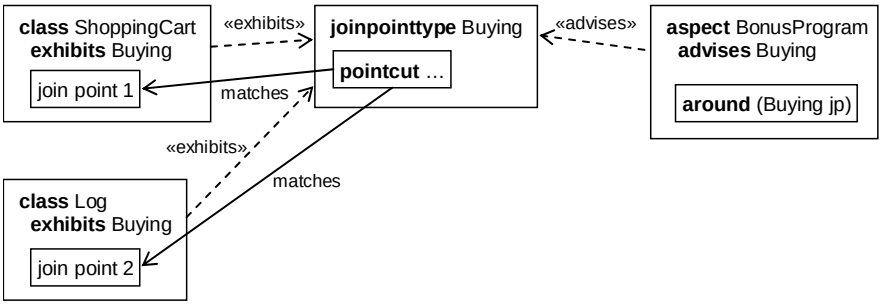

Fig 4. Classes `ShoppingCart` and `Log` both depend on the join point type `Buying`, as does aspect `BonusProgram` (dashed arrows). The pointcut defined in `Buying` matches join points in the classes, and is used to trigger the advice in `BonusProgram`. Note how this corresponds to Fig. 3 (c).

strong change dependency between a class and its exhibited join point types. Also, the surface structure (appearance) of join points of a single join point type can vary greatly from class to class, in ASPECTJ typically resulting in complex pointcuts consisting of many disjuncts (the "*quantification failure*" noted in [Sullivan et al. 2005]). Such a pointcut then mirrors the diversity of the classes it covers; it compromises independent development and presents a maintenance problem in its own right. This leads us to polymorphic pointcuts.

3.2 Polymorphic pointcuts

The object-oriented answer to diversity is polymorphism: rather than having the case analysis in a single place (e.g., a switch statement), the different cases are represented by different implementations of the same feature in different classes. Transferred to our problem, this means that each class exhibiting a certain join point type specifies its own pointcut, which matches the join points delivered by this class. In our example, the pointcut definition in `Buying` should therefore be split among the classes `ShoppingCart` and `Log` as follows:

```
class ShoppingCart exhibits Buying {
  pointcut Buying:
    execution(* add(Item, int)) && args(item, amount);
  …
}

class Log exhibits Buying {
  pointcut Buying:
    execution(* add(Item, int, ..)) && args(item, amount, ..);
  …
}
```

The resulting dependencies are shown in Fig. 5. Note that the local pointcuts lack information to which class(es) they apply; the scope of each such pointcut is implicitly constrained to the class in which its definition occurs. The disjunction of all class-local pointcuts associated with a join point type then constitutes the complete pointcut of that type. Because this is reminiscent of how different classes implementing the same interface provide for polymorphic methods in JAVA, we call such pointcuts *polymorphic*.

The question then is what remains of the pointcut in the join point type. Ideally, something like a design-by-contract language [Meyer 1997] was available that could specify the "semantics" of a join point type (i.e., the set of join points it covers) independent of its "implementation" in the classes (a *semantic pointcut language* [Lopes et al. 2003; Ostermann et al. 2005; Gybels and Brichau 2003; Masuhara and Kawauchi 2003] based on predefined tests [Sakurai and Masuhara 2008] or based on a conceptual model of the program [Kellens et al. 2006]). In absence of such a language, we have to resort to

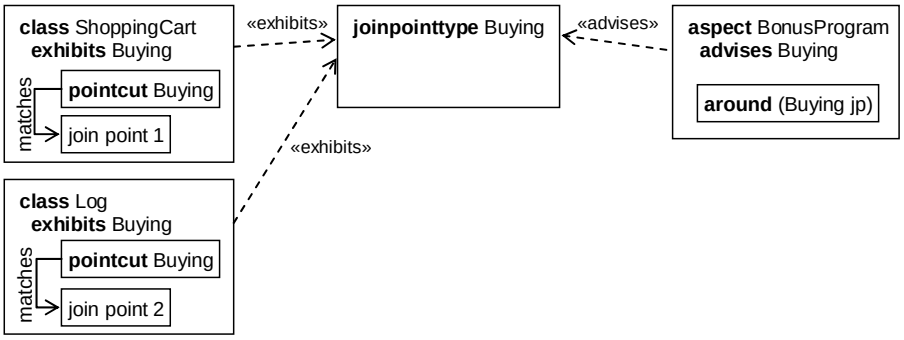

Fig. 5. Polymorphic pointcuts removing the dependency of join point types on exhibiting classes. The decoupling achieved is thus that of Fig. 3 (d). Note how polymorphic pointcuts resemble interface implementation in JAVA; here, however, (implicit) method invocations, not methods, are implemented.

an informal description of the nature of the join points (as is also done for the crosscutting interfaces described in [Sullivan et al. 2005; Griswold et al. 2006]; cf. Section 6 for a discussion). It is then the responsibility of the developer of each class exhibiting a join point type that the join points matched by the class's local (polymorphic) pointcut conform to this informal specification. Note how this mirrors the current situation with JAVA's interface types, which also leave semantics to their implementing classes.

3.3 Explicit announcement of join points

Our view of join points as instances of types opens up an interesting opportunity: it allows us to create join point instances explicitly. For instance, suppose that we want to add a counter cumulating the total number of items delivered, and that we therefore extend `ShoppingSession` and its method `buy` as follows:

```
class ShoppingSession {
  int totalAmount = 0;
  …
  void buy(Item item, int amount) {
    sc.add(item, amount);
    inv.add(item, amount, cus);
    log.add(item, amount, cus);
    totalAmount += amount;
  }
}
```

The added statement must be advised by `BonusProgram` to maintain consistency, but because this statement does not involve the variable `item` (access to which is needed by the advice), formulation of a suitable pointcut is unobvious (a combination of problems called *state-point separation* and *inaccessible join points* in [Sullivan et al. 2005] and reported to be quite common in [Murphy et al. 2001; Kästner et al. 2007]). Rather than rewriting our program so as to allow pointcut matching (the *intimacy* described in [Elrad et al. 2001; Xu et al. 2004; Sullivan et al. 2005]), we introduce the following construct that creates the join point instance with all required parameters explicitly:

```
class ShoppingSession exhibits Buying {

  …
  void buy(Item item, int amount) {
    sc.add(item, amount);
    inv.add(item, amount, cus);
    log.add(item, amount);
    exhibit new Buying(item, amount) {
      totalAmount += amount;
    };
  }
```

```
}
```

The type of this newly created join point, which does typically not fall under the type predicate (pointcut) of its declared type `Buying` (because otherwise the explicit creation would be redundant), can be thought of as an *anonymous inner subtype* (analogous to the anonymous inner classes of JAVA), i.e., as a join point subtype that comes with its own, implicit type predicate.

Viewing explicit join point creation as a special application of a more general concept of join point types (supporting implicit invocation with both implicit and explicit announcement) distinguishes our approach from other recent proposals of making implicit invocation more explicit, most notably those of [Hoffman and Eugster 2007] and [Rajan and Leavens 2008]). In particular, while Hoffman and Eugster also allow arbitrary blocks of code to be marked through explicit join points for being advised, and also to add pointcuts scoped to the hosting class to these join points, the notion of a join point type, and thus an explicit interface between the base code and the aspects, is absent from their approach. By contrast, Rajan and Leavens introduce event types akin to our join point types, but provide no means for class-locally specified implicit announcement. We support the notion of implicit announcement, but tame it by restricting its scope to that of the implementation of an explicit interface (the `exhibits` clause), and supplement it with explicit announcement where implicit announcement fails or requires awkward modification of code. Cf. Section 6 for a more detailed discussion of related work.

3.4 Join point subtypes

Interpreting explicit join point creation as a form of anonymous subtyping suggests that join point types can also have named subtypes. The notion of subtyping in turn suggests that instances of join point types are also instances of their supertypes (set inclusion semantics of subtyping) or, more concretely, that instances of join point types should be allowed to occur wherever instances of their supertypes are expected (the principle of substitutability [Liskov and Wing 1994]). This has implications for join point definition, join point creation, and join point advising.

3.4.1 Defining join point subtypes Following the definition of Section 3.2, a join point type definition consists of a set of field declarations and an informal characterization of the nature of its instances (i.e., what the occurrence of a join point conveys to the aspects observing it). Join point subtyping can then be defined by letting a join point subtype inherit the fields of its supertype, and by letting it add new fields (so-called type extension [Wirth 1988]): since the only purpose of join point types is to capture the relevant context in which implicit invocation takes place, and since an advice (as the only client of join point types) can use an instance of a join point subtype as if it were an instance of one of its supertypes, simply by dropping the added fields, substitutability is always guaranteed. That the informal characterization of the subtype does not contradict the ones of the supertypes cannot be checked — this is in the responsibility of the designer of the subtype. As much as properties of a join point type (its intension) are caught in the definition of class-local pointcuts, consistency must be maintained there; this will be discussed in Section 3.4.3.

Based on these considerations, we extend our notion of join point types by subtyping and require it to be declared using the `extends` keyword followed by the name of the join point supertype. Continuing our running example, this allows us to introduce a new, more general join point type `CheckingOut` defined as

```
joinpointtype CheckingOut {
  // going to take this item and amount from stock
  Item item;
  int amount;
}
```

and to let `Buying` subtype and inherit the fields from `CheckingOut` by writing

```
joinpointtype Buying extends CheckingOut {
  // buying this item and amount
}
```

A sibling join point subtype `Renting` could then be defined as

```
joinpointtype Renting extends CheckingOut {
  // renting this item and amount until returndate
  Date returndate;
}
```

which adds a field.

3.4.2 Advising join point subtypes Regarding instances of a join point subtype as instances of its supertypes implies that aspects advising join points of a certain type can advise join points of subtypes of this type as well, simply by ignoring the added fields. For instance, if we introduce a new aspect `BusinessRules` which provides advice for checking out items, such as

```
aspect BusinessRules advises CheckingOut {
  before (CheckingOut co) {
    if (Stock.amount(co.item) < co.amount)
      throw new OutOfStockException(co.item);
  }
}
```

this advice is invoked equally for join points (events) of type `Buying` and of type `Renting`, in the latter case simply ignoring the field `returndate`. Note that for the advice of `BusinessRules` to be invoked for `Buying` events, nothing has to be changed in the base classes. In particular, no new pointcut is needed. This will only become necessary if a join point of type `CheckingOut` that is not at the same time a join point of type `Buying` is to be created.

If an aspect offers advice for a join point type and any of its subtypes, the most specific advice for a join point instance (i.e., the one advising the most specific join point type) is to be invoked. For instance, extending the above aspect `BusinessRules` with advice for join points of type `Renting` allows us to override the advice for join point type `CheckingOut`, for instance to require renting to leave a minimum number of items in stock.

Note that accepting instances of join point subtypes where an advice expects instances of a join point supertype suffices to ensure that instances of a join point subtype are also instances of its supertypes. In particular, there is no need for making sure that a (class-local) pointcut of a join point type includes all join points of the (class-local) pointcuts of its subtypes — if this is not the case, instances of the subtypes are nevertheless treated as if they were (direct) instances of the supertype. Instead, it must be prevented that two instances are created for the same event.

3.4.3 Exhibiting join point subtypes The previous subsection answered the question, which advice should be invoked by a join point instance? Complementary to this question is, a join point instance of which type should be created in case the pointcut of a join point type and one of its subtypes match the same point of execution in a program? Clearly, creating two instances for the same event where the type of one event subsumes the type of the other does not reflect adequately what happened (and would lead to the double execution of the same advice of an aspect if it is the most specific for both instances, or to execution of advice for the type and the supertype if the aspect offers both). Since instances of a join point type are subsumed by its supertypes, but not vice versa, an

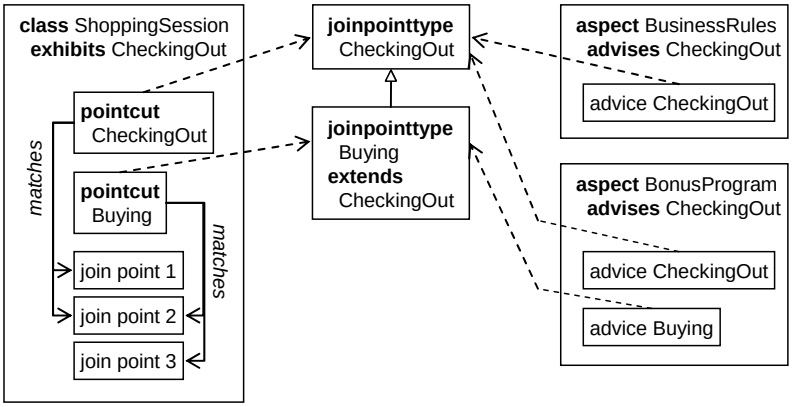

Fig. 6. Schematic view of class `ShoppingSession` exhibiting two join point types, `CheckingOut` and `Buying`, one of which is a subtype of the other, and two aspects, `BusinessRules` and `BonusProgram`, one advising both join point types, the other only the supertype. Join point `1` implicitly invokes advice bound to `CheckingOut` in both aspect `BusinessRules` and `BonusProgram`, whereas join points `2` and `3` implicitly invoke advice `CheckingOut` in `BusinessRules` and `Buying` in `BonusProgram`. Note that although it matches the pointcut of `CheckingOut`, join point `2` does not invoke advice `CheckingOut` in `BonusProgram` — it is caught by the more specific advice `Buying` (see text).

instance of the more specific type — and only this — must be created. Fig. 6 gives an example of the situation.

Fortunately, what seems like a non-trivial technical problem can be solved by a simple trick: rather than first checking whether the pointcut of a join point type and one of its subtypes overlap and then making sure that for matches of this overlap, only one instance is created, we implicitly conjoin the pointcut of the join point type with the negations of the pointcuts of its subtypes specified in the same class. [5] Effectively, adding a join point subtype does not reduce the extension of the join point type, since as noted above, the instances of the subtype that are created instead are always accepted as instances of the type. However, if pointcuts of sibling join point types overlap and a join point in the overlap gets executed, one join point instance is created for each of the overlapping types. This is intentional, since the two instances represent different events. On the other hand, it means that if the most specific advice of an aspect is one for a common supertype, this advice is executed twice.[6]

Fig. 6 illustrates the semantics of join point subtyping for class `ShoppingSession` exhibiting three join points of join point type `CheckingOut` and one of its subtypes, `Buying`, and for the aspects `BusinessRules` and `BonusProgram`. `BusinessRules` advises only the join point supertype, while `BonusProgram` advises both the join point supertype and the join point subtype. The local pointcuts defined in `ShoppingSession` for the join point types `CheckingOut` and `Buying` overlap such that `CheckingOut` matches join point `1` and join point `2`, whereas `Buying` matches join point `2` and join point `3`. Set inclusion semantics of join point subtyping requires that join point `3` belongs to the extension of join point type `CheckingOut` even though the corresponding pointcut does not match it; this is made

[5] Note that our trick is somewhat similar to the one employed by EIFFEL, which disjoins preconditions and conjoins postconditions to meet the conditions of subtyping [Meyer 1997].

[6] This situation may appear somewhat awkward. It could be avoided by adopting the abstract superclass rule for join point types: all join point types that have subtypes should be declared abstract, i.e., should not be allowed to have instances of their own. We leave this debate for future exploration.

up for by advice `CheckingOut` in aspect `BusinessRules` handling this join point (because it is an instance of a subtype of `CheckingOut`). Join point `2` is matched by both pointcut `CheckingOut` and pointcut `Buying`; since join point `2` is an instance of both join point type `CheckingOut` and `Buying`, only one instance (of type `Buying`) is created, leading to the execution of advice `CheckingOut` in aspect `BusinessRules` and advice `Buying` in aspect `BonusProgram` (and not advice `CheckingOut` in aspect `BonusProgram`).

3.5 Inheritance of join point exhibition

Orthogonal to the question of having join point subtypes is the question of whether an `exhibits` clause of a class is inherited by its subclasses, and if so, if the corresponding pointcuts are inherited with it. Since inheritance is a known problem for modularity, and since modularity is a driving force for our capture of IIIA, we decide this issue based on modularity considerations.

Letting a class declare that it exhibits join points of a certain type expresses a *statement of consent* that some of the classes' variables may be accessed by aspects advising the exhibited join points. Moreover, the local pointcut specification specifies within the class which of its variables can get accessed. This policy gives classes the opportunity to deny aspects access; in particular, and much in the spirit of information hiding, variables can only be accessed if the owning class explicitly grants access to them.

The fragile base class problem of object-oriented programming has taught that seemingly innocuous changes to a base class can break the contracts of subclasses (see [Mikhajlov and Sekerinski 1998] for a discussion of a wide range of such situations). This can directly be transferred to the inheritance of (class local) pointcuts: a change in a pointcut expression that makes perfect sense in the base class can have unintended effects in any of its subclasses (which may not even be known to the changer of the base class). Therefore, we decided that the scope of pointcut definitions of a class does not automatically extend to its subclasses, so that the designer of a subclass does not have to be aware of the pointcuts of its superclasses. Note that if the subclass inherits code from the superclass that is covered by pointcuts of the superclass (so that join point instances may be created when the code is executed), execution of this code in the context of the subclasss may nevertheless lead to join point creation; however, this code is outside the control of the subclass and if its effect worries the programmer, the superclass's specification should be consulted. The situation is analogous to calling a method of a different class whose execution leads to join point creation — this also does not give rise to a corresponding `exhibits` clause in the calling class. Besides, and as will be seen from an example in Section 5.3 (Fig. 11), not letting subclasses inherit join point exhibition preserves its crosscutting nature.

There are however situations in which a subclass should exhibit the same join point type as one of its superclasses. In these cases, the `exhibits` clause must be repeated in the subclass, and a new pointcut must be specified (whose scope is again implicitly limited to the enclosing class). If the pointcut of the subclass is syntactically identical to that of the superclass, the one of the superclass can be reused by using the `super` keyword; if it is a variation of that of the superclass, `super` can be part of a suitable logical expression.

Fig. 7 gives an example of how explicit pointcut inheritance (via the `super` keyword) works in practice. In the example, which is based on the standard drawing example from [Kiczales and Mezini 2005], a hierarchy of geometrical shapes (`Shape`, `Point`, and `Line`) has to notify a display whenever a shape is changed so that its representation on the display can be updated. The join point type representing the corresponding event is named `UpdateSignalling`; it has no fields and its informal intension is specified by a comment (recall that the pointcuts are sourced out into the exhibiting classes).

```
joinpointtype UpdateSignalling {
  // change of state that affects display
}

abstract class Shape exhibits UpdateSignalling {
  pointcut UpdateSignalling : execution(void moveBy(int, int));
  public abstract void moveBy(int dx, int dy);
}

class Point extends Shape exhibits UpdateSignalling {
  pointcut UpdateSignalling : super || execution(void set*(int));
  int x, y;
  public int getX() { return x; }
  public int getY() { return y; }
  public void setX(int x) { this.x = x; }
  public void setY(int y) { this.y = y; }
  public void moveBy(int dx, int dy) {
    x += dx;
    y += dy;
  }
}

class Line extends Shape exhibits UpdateSignalling {
  pointcut UpdateSignalling : super;
  private Point p1, p2;
  public Point getP1() { return p1; }
  public Point getP2() { return p2; }
  public void moveBy(int dx, int dy) {
    p1.x += dx; p1.y += dy;
    p2.x += dx; p2.y += dy;
  }
}

aspect Display advises UpdateSignalling {
  after (UpdateSignalling us) { update(); }
  static void update() {...}
}
```

Fig. 7. Drawing example [Kiczales and Mezini 2005] with polymorphic pointcuts and explicit pointcut inheritance (see text).

The class `Shape` serves as a common abstraction of the classes `Point` and `Line`. It specifies an abstract method `moveBy` that should be supported by all subclasses. Since moving a shape usually requires a display update, `Shape` also specifies a pointcut for `UpdateSignalling` that matches every execution of the `moveBy` method. However, since `moveBy` in `Shape` is abstract, this join point never matches; its sole purpose is to serve reuse by join points in subclasses.

Subclass `Point` makes use of this pointcut by referring to it using `super`. However, setting a coordinate of a point individually (via a corresponding setter) also requires an update, so that it makes a corresponding addition to the inherited pointcut. Subclass `Line` on the other hand offers no such possibilities, so that it can reuse its superclass's pointcut without alterations.

One might argue that defining an abstract method and a pointcut matching its execution as in the example can only mean that the pointcut should match the concrete implementations of the method. Indeed, one could go one step further and allow interfaces to define pointcuts on the methods they declare. However, for reasons of modularity (a simple change of the interface a server implements or a client relies on may exhibit data previously kept secret without either knowing) and the analogy to the fragile base class problem (the fragile pointcut problem [Störzer and Graf 2005]), and also because of the arguments made in Section 5.3, we do not consider such possibilities here.

3.6 Inheritance of join point advising

In ASPECTJ, an aspect (which is implemented as a special kind of class) can inherit from a superclass or from an abstract superaspect. The semantics of inheritance of the class facet of aspects is the same as that for JAVA (i.e., members are inherited by the subaspect, in which inherited methods can be overridden). Pointcuts — like class members — are also inherited and can be overridden in subaspects. Finally, advice, which is always unnamed, is inherited, but cannot be overridden.

Aspect inheritance is mainly used for the possibility to specify abstract pointcuts in an abstract superaspect, which are then overridden by concrete pointcuts in concrete subaspects. This allows reuse of advice defined in the superaspect (possibly relying on abstract pointcuts) by providing a different (or concrete) set of pointcuts.

Since our conception of IIIA relies on the aspects of ASPECTJ as the containers of advice, we need to consider inheritance among aspects as well. However, since in our capture of IIIA pointcuts are not defined as parts of aspects, a subaspect cannot inherit or override pointcuts. Overriding of advice on the other hand would be possible in IIIA (since every advice is associated with the join point type it advises), but this would require that aspects inherit advice from their superaspects. This is not useful, however, since every join point instance is dispatched to (and thus causes the execution of advice in) all aspects declaring to advise its type (or any of its supertypes; cf. above), so that it would lead to duplicate execution of identical advice in all cases in which the advice is not overridden in the subaspect. Therefore, a subaspect inherits neither the `advises` clause from its superaspect, nor its advice bound to join point types.

3.7 Achievements

With IIIA designed as above, we have introduced a new kind of interfaces, join point types, that specify the information passed between an implicit invoker, the exhibiting class, and a set of implicitly invoked, the advising aspects (cf. Fig. 3 (d)). By declaring which join point types it exhibits, not which aspects may advise it, the declaring class remains completely unaware of its advising aspects. At the same time, the class is in full control of the data it passes as parameters of implicit invocations, and of where or when these invocations take place. Unlike with typed P/S [Eugster 2007] and related approaches (e.g., [Rajan and Leavens 2008]), implicit invocation may be implicitly announced — however, the pointcuts specifying the implicit announcements are always class local, so that a programming tool can always mark their shadows [Hilsdale and Hugunin 2004] in the source code of the class, without depending on any other source. This is significantly different from, e.g., aspect-aware interfaces [Kiczales and Mezini 2005] and XPIs [Sullivan et al. 2005; Griswold et al. 2006].

On the other side of the interfaces, the aspects and their advice can remain unaware of the classes they advise — making reference to join point types as interfaces only, they are completely decoupled from their advised classes, and also from the pointcuts triggering the advice. This is significantly different from other approaches preserving the modularity of aspects and classes in which aspects still specify their own pointcuts [Aldrich 2005; Ongkingco et al. 2006], and also to ones in which pointcuts are external to aspects, but nevertheless directly referenced (and homomorphic) [Sullivan et al. 2005; Griswold et al. 2006], but similar to typed P/S [Eugster 2007] and PTOLEMY [Rajan and Leavens 2008].

As a consequence, our proposal makes evolution of classes completely independent from aspects: anyone wishing to make changes to the class can check locally, without resorting to any other declarations or definitions than those of the exhibited join point types, whether one's changes respect the advising aspects' interfaces. Much more: *in case one must break with a (local) pointcut definition (i.e., must change the program so that a pointcut no longer matches where it should), one can adapt it without affecting the definitions in other classes, because the scope of a local branch of a join point is always lim-*

ited to the owning class. In fact, the only thing the programmer must guarantee is that the variables declared in the join point type are correctly bound to variables in the context of the local join points (where that they are bound is checked by the compiler; see Section 4.2). If that is impossible using a (local) pointcut definition, one can still work around it with the explicit creation of a join point instance. This means that the advised classes can be changed at will, as long as the local pointcut can be adapted accordingly. The problems of state-point separation and inaccessible join points described in [Sullivan et al. 2005] and also the fragile pointcut problem [Störzer and Graf 2005] therefore no longer exist.

On the other side of the coin, it should be clear that with our form of IIIA and its mutual explicit interfaces, it is impossible to extend code that has not been written foreseeing extension, or cannot be changed to allow it. While this may be viewed a serious limitation of our approach, we counter that not allowing arbitrary unforeseen extensions, that is, not allowing one to work around existing interfaces, is the immanent price for modularity. In a way, our join point interfaces declaring implicit invocation are similar to inheritance interfaces declaring dynamic binding and open recursion: they allow the extending of base code at well-defined plug-points.

Last but not least, since our achievements in terms of modularization depend to a large part on the utilization of polymorphic, class-local pointcuts, one might wonder what the specific cost of this feature of our proposal is. In particular, one might be concerned that a single global pointcut will translate to so many, perhaps identical, class-local poincuts that our approach becomes infeasible. While this may indeed be the case for certain standard examples of pointcuts (such as those used for tracing and debugging), we will see in Section 5 that for general applications of IIIA, proliferation of pointcut definitions is small.

4. IIIA LANGUAGE SPECIFICATION AND IMPLEMENTATION OF A COMPILER

We specify our language extension by giving its syntax rules, by describing the semantic checks a compiler has to perform, and by describing the transformations of a program using our IIIA constructs into one suitable for an aspect weaver. Note that we do not provide any formal soundness or completeness proofs; however, during our practical experiments with our compiler implementing the language as described here (the results of which are presented in Section 5) we have not witnessed any type errors in the generated code.

4.1 Syntax

The syntax of our IIIA extension to JAVA and ASPECTJ is specified by the rules shown in Fig. 8. Note that join point type declarations have no access modifier; they are implicitly public. Also, the syntax requires that `exhibit` must be followed by a `new` expression (a kind of default constructor call, where the parameters are implicitly given by the join point type's fields). This makes sure that exhibits cannot be executed on a variable.

4.2 Static semantics (informal)

A IIIA program must conform to the following static semantic constraints:

- The members of join point types must be fields. Their only allowed modifier is `final`. A specified supertype must be a join point type. (These are semantic, rather than syntactic, restrictions because join point types are defined as special kinds of classes.)
- The types following the `exhibits` and the `advises` clauses must be join point types.
- For every join point type declared to be exhibited by a class, there must be a corresponding pointcut definition or explicit join point creation in the class, or an error will

```
(* --- Extending the Base-Grammar of AsepctJ --- *)
type                          = … | joinpoint_type;
type_declaration              = … | joinpointtype_decl;
class_declaration             = … | exhibiting_class_declaration;
class_member_declaration      = … | polymorphic_pc_declaration;
block                         = … | exhibit_block;
aspect_declaration            = … | advising_aspect_declaration;
advice_declaration            = … | joinpoint_advice_declaration;
basic_pointcut_expr           = … | super_pointcut_expr;

(* ---- IIIA-Syntax-Extension ---- *)
(* Join Point Type Extension *)
joinpoint_type                = { IDENTIFIER "." } IDENTIFIER;
joinpointtype_decl            = ["public"] "joinpointtype" IDENTIFIER ["extends" joinpoint_type]
                                joinpointtype_body;
joinpointtype_body            = "{" {joinpointtype_field_declaration} "}";
joinpointtype_field_declaration = ["final"] type IDENTIFIER ";";

(* Class-Side Extension *)
super_pointcut_expr           = "super";
polymorphic_pc_declaration    = "pointcut" joinpoint_type ":" pointcut_expr ";";
exhibiting_class_declaration  = [modifiers] "class" IDENTIFIER
                                ["extends" type] ["implements" interface {"," interface}]
                                "exhibits" joinpoint_type {"," joinpoint_type} "{" class_body "}";
exhibit_block                 = "exhibit" "new" joinpoint_type ( {argument} ) { {statement} } ";";

(* Aspect-Side Extension *)
advising_aspect_declaration   = [modifiers] "aspect" IDENTIFIER  "advises" joinpoint_type {"," joinpoint_type}
                                "{" aspect_body "}";
joinpoint_advice_declaration  = ("before"|"around"|"after") "(" joinpoint_type IDENTIFIER ")" "{" {statement} "}";
```

Fig. 8. Syntax of the IIIA language extension of ASPECTJ (EBNF).

be reported. For every pointcut defined in a class, there must be a corresponding `exhibits` declaration, either of its associated type or one of its supertypes. Each class-local pointcut must bind all fields of the corresponding join point type to the context of a join point using `this()`, `target()` or `args()`. Class-local pointcut definitions must have no access modifiers — they would be meaningless, since the pointcuts are never referenced from the program text (the names are necessary for solely disambiguation). (Again, this is a semantic rather than a syntactic check since class-local pointcuts are special cases of the more general ASPECTJ pointcuts.

- For every explicit join point creation (of the form `exhibit new <join point type>(…) {…}`) in a class, the exhibited join point type or one of its supertypes must be declared as being exhibited by the class. The parameters to the instance creation (listed in the parentheses) must have the types of the fields of the instantiated join point type.
- For every join point type declared to be advised by an aspect, there must be a corresponding advice defined in the aspect. Every advice must name one and only one join point type. For every advice defined in an aspect, there must be a corresponding `advises` declaration.
- A `proceed` in the body of an advice must have precisely one parameter, which must be a variable of the join point type to which the advice is bound.
- For every occurrence of `super` in a pointcut definition of a class, there must be a pointcut definition for the same join point type in one of its superclasses.

The new language constructs of IIIA, join point types, polymorphic pointcuts, and explicit join point creation, map to standard constructs of JAVA and ASPECTJ as follows:

- A join point type maps to a class with the type's fields and a constructor for creating instances and setting the fields; a join point subtype maps to a corresponding subclass.
- A class local pointcut for a join point type maps to a disjunct of the global pointcut for that type. The disjunct is restricted to the scope of the class by adding a

`within(<class name>)` expression conjoined with `!within(<inner class name>)` expressions, excluding matches in inner classes and outside the class. If the class has pointcuts for subtypes of the visited pointcut's associated join point type, their generated disjuncts are negated and conjoined with the disjunct of the visited pointcut. For internal reference, the global pointcut is named by the name of the join point type.

- An aspect maps to an aspect; an advice associated with a join point type maps to advice bound to the correspondingly named global pointcut created from the class-local branches.
- To realize substitutability (subtyping) of join point types on the aspect side, subtypes of join point types advised in an aspect, for which no specific advice exists in that aspect, are mapped to new advice whose body is identical to that advising the supertype (cf. the discourse in Section 3.4.2).
- Finally, explicit announcement of a pointcut maps to a specially tagged block (see the implementation of the compiler described below).

Note that `exhibits` and `advises` clauses are used for semantic checking only; they are compiled away.

4.3 Implementation of a compiler

We have implemented a compiler for IIIA on top of the AspectBench Compiler (abc) [Avgustinov et al. 2006]. Our implementation adds a number of compiler passes, which are roughly characterized as follows (passes performing the semantic checks omitted):

- The first pass collects all join point types and creates a new node holding the fields, a constructor setting the fields (including those inherited from supertypes), and an empty pointcut definition for each type.
- The second pass visits all classes, collects all pointcut branches specified in each class, explicitly restricts the scope of each branch to the class in which it occurred, excluding pointcuts for join point subtypes (see above), and adds it so-modified as a disjunct to the pointcut of the corresponding join point type. Explicit join point creation (as expressed by the `exhibit` statement) is handled by introducing a new node type, `ExhibitBlock`, to the AST which has a field for holding the exhibited join point type. This new node type is complemented by a new pointcut designator matching nodes that are tagged with the join point type with which a pointcut is to be associated.
- The third pass visits all aspects and binds each of its advices to the corresponding pointcut constructed in the second pass, translating the fields of the join point type to parameters of the advice. It inserts a constructor call for the join point type at the beginning of each advice, which binds the pointcut and advice parameters to the fields of the join point type advised. It also creates copies of the advice for all join point types that are subtypes of the types already advised by the aspect, and for which no specific advice is defined in the aspect. Finally, it adds the fields of the join point type to the `proceed` statement in around advice.

Our compiler thus converts a program making use of our IIIA syntax to standard constructs of JAVA and ASPECTJ, the sole additions being the `ExhibitBlock` nodes and the corresponding new pointcut designator.

The compiler together with additional material can be downloaded from www.fernuni-hagen.de/ps/prjs/IIIA/.

5. APPLICATION AND FINDINGS

Experience with the design of object-oriented programming languages has taught that subtyping and inheritance are sources of considerable (and also often unexpected) complexity. However, while formal analyses can help avoid inconsistencies and ill-definedness, they provide little help for making the right design decisions, i.e., for balancing issues such as usability and expressiveness. We believe that this can only be achieved by experimenting with a language on actual programming projects, preferably involving people other than the original designers of the language.[7]

To test the feasibility of our design decisions, we have refactored a number of small standard examples of ASPECTJ to apply our conception of IIIA, and rewrote two larger applications with it, namely BERKELEY DB (used in a prior case study of ours [Kästner et al. 2007]) and AJHOTDRAW, an ASPECTJ-based refactoring of JHOTDRAW, a widely known drawing framework.

5.1 Standard ASPECTJ examples

Applying IIIA to standard examples of AOP has led to code like that of Fig. 7 and Fig. 13, i.e., to code making good use of polymorphic pointcuts, and moderate use of join point subtyping. In particular, none of the examples required use of explicit join point creation (explicit announcement) — in fact, most improvements over standard ASPECTJ solutions came from more explicit program organization, that is, easily visible dependencies through declaration of join point types, join point instance production, and consumption (advising). On the other hand, the increased explicitness of program dependencies resulted in a scattering of pointcut definitions and `exhibits` clauses that made certain examples unconvincing. In particular, the usual debugging, profiling, and tracing (which in the examples all use broadly generic pointcuts; but see [Sullivan et al. 2005; Kästner et al. 2007] for how this fails in practice), are not reasonably expressed using IIIA. This can be ascribed to the loss of obliviousness and quantification brought by IIIA, and seems to be the necessary price for achieving modularity (cf. the discussion in Section 6).

5.2 Case study #1: BERKELEY DB

To evaluate IIIA in a larger, more realistic setting, we revisited a prior case study of ours, in which we evaluated ASPECTJ's adequacy as a target language for a feature-oriented refactoring [Kästner et al. 2007]. As the basis of this study, we had chosen BERKELEY DB, a widely used open source embedded database engine implemented in JAVA (ca. 84 KLOC). BERKELEY DB offers several more or less interacting features such as transaction safety, caching, multithreading, statistics, and debugging; in most of its installations, not all of these features are actually needed. Literature suggests that AOP, ASPECTJ in particular, is ideally suited to host such a decomposition [Lee et al. 2006], and indeed, most of the features of BERKELEY DB (34 of 39) are crosscutting in character, in that they extended up to 30 (of about 300) classes per feature.

Due to the nature of the problem, a large part of the necessary refactoring consisted of introducing so-called inter-type declarations, i.e., removing fields and dependent methods from classes and reintroducing them via aspects representing the corresponding features. This capability of ASPECTJ is foreign to IIIA, but similar to other object-oriented extension mechanisms such as mixins [Bracha and Cook 1990] or virtual classes [Madsen and Møller-Pedersen 1989] so that IIIA should not be denied its practicality based on not of-

[7] This is in fact what we did: the first two authors are the original designers of the language and its compiler, while the latter two have experimented with it on real projects and provided detailed feedback of its usability.

```
pointcut latchedMethods(IN in): (
    (execution(boolean IN.validateSubtreeBeforeDelete(int)) &&
      within(IN)) ||
    execution(void IN.verify(byte[])) ||
    execution(boolean BIN.isValidForDelete())
  ) && this(in);
```

Fig. 9. ASPECTJ pointcut with branches that are distributed to the exhibiting classes in IIIA.

fering such possibilities.[8] The remainder of the refactoring had to connect the new feature code to the "base" code and to each other, for which ASPECTJ's implicit invocation mechanism using join points and pointcuts offered itself.

Interestingly, the picture we found in this repeated case study was quite different from that obtained by applying IIIA to the standard examples of AOP as described above. First of all, polymorphic pointcut definitions could not be used as often as expected; yet this was not because they would have led to undue scattering, but rather because there were only few cases in which a single advice applied to more than one place (which questions the need for quantification in an undertaking such as ours). In fact, we found that of 214 advices, only 24 were designed to match more than one join point (shadow); of these, 5 could use pattern expressions (wildcards) to specify their multiple matches (typically overloaded methods with different parameters), while the remainder resorted to enumeration (explicit disjunctions of single matches) as exemplified in Fig. 9. Splitting such disjunctions into class-local branches is trivial (it could in fact be performed by a tool); the result is simply a shift of responsibility — each target class now specifies for itself whether and which matches it contributes.

On the other hand, explicit join point creation (which seemed mostly unneeded for the standard AOP examples) turned out to be a huge improvement over join point selection through pointcuts. Using ASPECTJ as the target language (our original refactoring described in [Kästner et al. 2007]), of the 484 needed advices, 218 applied to a single statement or a sequence of statements in the middle of a method, so that standard execution or call join points could not be used. In the original study, we worked around these cases by extracting methods (43 times), introducing calls to empty hook methods (121 times) or by matching nearby method calls or field access (54 cases; note how this hides the semantics of an aspect and is extremely sensitive to change). To be able to access temporary variables in aspects, we also had to resort to hook methods, to method objects, to code replication in aspects [Sullivan et al. 2005], and to other awkward workarounds. Literally all of these problems vanish with the availability of explicit join points, which therefore became the greatest facilitator of the refactoring.[9]

Fig. 10 exemplifies this finding on a simplified excerpt from BERKELEY DB's `Tree` class. In this example, the `traceInsert` call in the original `insert` method is part of a feature and therefore should be moved to an aspect.[10] In the ASPECTJ implementation an artificial hook method is necessary to expose a join point in the middle of the method and to expose the temporary variables. The IIIA solution uses an anonymous join point subtype (explicit join point creation) instead. Note how this is different from using annota-

[8] In fact, as has been argued in [Apel et al. 2008], mixin techniques are in many situations simpler than the static injection capabilities of ASPECTJ, so that integrating IIIA with a mixin-based approach may be worthwhile.

[9] Also, use of explicit join points eliminated the need to use ASPECTJ's more sophisticated pointcuts such as `cflow` or `cflowbelow`, which in our prior refactoring were required to fix various matching problems.

[10] The tracing feature in BERKELEY DB writes log messages at well-defined points in the execution of the program, logging several context variables. These points are too heterogeneous to be matched by generic pointcuts (quantification).

```
// Original Berkeley DB Source (excerpt, strongly simplified)
public class Tree {
   public long insert(LeafNode ln, byte[] key, …) {
     BottomNode bin = findBINForInsert(key, …);
     long position = ln.log(key, …);
     bin.updateEntry(ln, position, key);
     bin.clearKnownDeleted();
     traceInsert(Level.FINER, bin, ln, position);
     …
   }
}

// ASPECTJ Implementation
public class Tree {
   public long insert(LeafNode ln, byte[] key, …) {
     …
     bin.clearKnownDeleted();
     hook(Level.FINER, bin, ln, position);
     …
   }
   void hook(Level l, BottomNode b, LeafNode l, long p) {}
}
public aspect Logging {
   before (Level l, BottomNode bin, LeafNode ln, long pos):
     execution(void Tree.hook(..)) && args(l, bin, ln, pos) {
     traceInsert(l, bin, ln, pos)
   }
}

// IIIA Implementation
joinpointtype Trace { Level logLevel; }
joinpointtype TraceInsert extends Trace {
   BottomNode bin;
   LeafNode ln;
   long position;
}
public class Tree exhibits TraceInsert {
   public long insert(LeafNode ln, byte[] key, …) {
     …
     bin.clearKnownDeleted();
     exhibit new TraceInsert(Level.FINER, bin, ln, position) {};
     …
   }
}
public aspect Logging advises TraceInsert {
   after (TraceInsert t) {
     traceInsert(t.logLevel, t.bin, t.ln, t.position);
   }
}
```

Fig. 10. Reimplementation of a logging feature using IIIA. The explicit join point creation provides access to local variables. Note the empty block: there is no statement to be advised, only a point between two statements. `before`, `after`, and `around` all have the same effect here.

tions: an annotation cannot annotate a sequence of statements, nor can it provide access to variables.

While explicit join point creation (that is, anonymous inner join point subtyping) proved extremely useful in our feature-oriented refactoring endeavour, named join point subtyping did not. However, this could not be ascribed to a general inappropriateness of the concept, but rather to the fact that in BERKELEY DB, the sets of join points covered by each join point type were usually quite small (432 of 484 were singular and only 10 covered more than 3 points in the program [Kästner et al. 2007]), so that set inclusion of extensions (indicative of subtyping; cf. Section 3.4) did not occur. At the same time, only 28 join points where shared by different aspects, which were however unrelated, also providing no good opportunity for subtyping. On the other hand, Fig. 10 shows how an abstract tracing join point type (`Trace`) can serve to structure the domain even when there is no shared advice (i.e., advice applicable to a join point type and its subtypes).

Our findings seem to be consistent with that of others, in particular those reported on in [Sullivan et al. 2005; Hoffman and Eugster 2007]. In fact, the material presented in [Sullivan et al. 2005] suggests that our anonymous join point subtyping, or explicit join point creation, can help solve the problems of *state-point separation*, *inaccessible join points*, and *quantification failure*. At the same time, we found that the IIIA implementations of features in BERKELEY DB are easier to use and to read, but this of course lies in the eye of the beholder.

5.3 Case study #2: AJHOTDRAW

In a second case study, we migrated the aspect-oriented version of JHOTDRAW, AJHOTDRAW, to our implementation of IIIA. AJHOTDRAW was written as a demonstrator of the feasibility of using ASPECTJ in existing applications, and was used as the basis of several recent case studies [van Deursen et al. 2005; Coelho et al. 2008]. Like our own migration of BERKELEY DB to ASPECTJ [Kästner et al. 2007], AJHOTDRAW makes use of both inter-type declarations (introductions) and implicit invocation. Since IIIA is orthogonal to introductions, we refactored only the implicit invocation part.

Overall, we extracted 36 different join point types and implemented 19 aspects to advise them and 25 classes to exhibit them (for comparison: AJHOTDRAW implemented 31 aspects of which 14 contained advice, while 17 contained only inter-type declarations). Splitting the (global) pointcuts into class-local branches was straightforward, since almost all applied to a single class only, and the remaining three pointcuts used separate disjuncts to specify the pointcuts for each class anyway (note how this amounts to inverting the transformation performed by our compiler). The situation was thus quite similar to BERKELEY DB, where we also found only few pointcuts matching more than one join point shadow (cf. Section 5.2). However, other than in the case of BERKELEY DB we found a convincing application of join point subtyping.

5.3.1 Uses of join point subtyping One striking observation we made was that many (23 out of 36) of the join point types we introduced were strongly related, both in terms of the meaning they carried and in terms of the fields they defined. In fact, these join point types naturally arranged into three subtype hierarchies, structuring the domain as in the BERKELEY DB case. However, in AJHOTDRAW substitutability, i.e., the application of an advice bound to a join point type to instances of its subtypes, also proved useful.

JHOTDRAW has an `AbstractCommand` class, which is an abstract superclass of all command classes. In AJHOTDRAW, the undo facility and some sanity checks that apply to commands were extracted into aspects (`UndoableCommand`, `CommandPolicy`, and `CommandDamage`). The pointcuts of these aspects were defined as

```
pointcut commandExecuteCheckView(AbstractCommand acommand) :
      this(acommand)
      && execution(void AbstractCommand+.execute())
      && !within(*..DrawApplication.*)
      && !within(*..CTXWindowMenu.*)
      && !within(*..WindowMenu.*)
      && !within(*..JavaDrawApp.*);
```

and

```
pointcut commandExecuteNotifyView(AbstractCommand acommand) :
     commandExecuteCheckView(acommand)
     && !within(org.JHOTDRAW.util.UndoCommand)
     && !within(org.JHOTDRAW.util.RedoCommand)
            && !within(org.JHOTDRAW.standard.CopyCommand)
            && !within(org.JHOTDRAW.standard.ToggleGridCommand)
            && !within(org.JHotDraw.contrib.zoom.ZoomCommand);
```

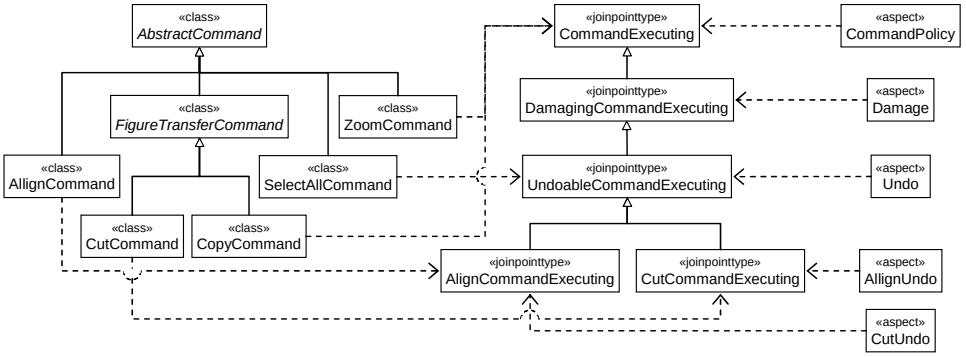

Fig. 11. Class hierarchy, the hierarchy of join point types exhibited, and the aspects advising them. Note how join point exhibition is independent from subclassing.

Note how this first defines a pointcut applying to all classes of the `AbstractCommand` hierarchy, and then removes matches in certain subclasses (using `!within` expressions). This reflects the crosscutting nature of the `commandExecuteNotifyView` pointcut, which means that if a class is to be advised, its subclasses are not automatically to be advised, too. However, this information is coded in rather unwieldy pointcut definitions.

Using our join point types and join point subtyping it was easy to refactor the code to reflect the relationships between commands and their aspects more clearly. As can be seen from Fig. 11, the classes of the `AbstractCommand` hierarchy exhibit join point types from the `CommandExecuted` hierarchy. Note that the two hierarchies are not parallel; in fact, if join point type exhibition is considered a form of classification, this classification is independent of the primary (or "dominant" [Tarr et al. 1999]) classification established by subclassing. For this to be the case, however, it is necessary that join point exhibition is not inherited by subclasses (cf. the discussion in Section 3.5). If a subclass happens to exhibit the same join point type as its supertype, it has to declare to do so, and if it wants to use the same pointcut, it has to import it (using `super`). Inheritance of join point exhibition by subclasses is not default behaviour, but an explicit act.

While explicitly marking each class with the join point types it exhibits models the crosscutting nature of the `AbstractCommand` and the `CommandExecuted` hierarchies nicely, the aspect side also makes use of subtyping and substitutability: here, all classes exhibiting a join point type from the `CommandExecuted` hierarchy enjoy the advice bound to the `CommandExecuted` join point type (the root of the join point type hierarchy) as expressed in the `CommandPolicy` aspect; all classes exhibiting a subtype of the `DamagingCommandExecuted` join point type enjoy the advice of the aspect `CommandDamage`; and so forth. The dispatching of a single event (the execution of a command) to advices of several types is fully implicit, elegantly reducing the complexity of the code.

5.3.2 Refactoring the OBSERVER pattern Beyond the refactoring of existing aspects in AJHOTDRAW, we found 16 opportunities for using IIIA in classic publish/subscribe constellations, that is, for occurrences of the OBSERVER pattern. For this, we replaced the event class by a join point type, moved the calling of the `update()` (or like) method to an aspect, replaced the registration of the observer by an `advises` clause, and replaced notification by an `exhibits` clause and a suitable pointcut in the subject class. However, due to the singleton character of aspects, and due to the fact that in the original code, individual objects, not classes, are notified, boilerplate code had to be written in order to dispatch updating to the observing object (Fig. 12 gives an example of this). This could have been avoided had a symmetrical approach to AOP been used in which objects advise objects (such as classpects [Rajan and Sullivan 2005]; cf. the discussion in Section 6).

```
public joinpointtype ToolJP {
  Tool subject;
}

public joinpointtype ToolEnabledJP extends ToolJP {
  boolean enabled;
}

public abstract class AbstractTool exhibits ToolEnabledJP … {
  …
  public void setEnabled(boolean newIsEnabled) {
    …
    exhibit new ToolEnabledJP(this, newIsEnabled) {};
    …
  }
}

public class ToolButton implements ToolListener … {…}

aspect ToolListening advises ToolEnabledJP … {
  HashMap<AbstractTool, Set<ToolListener>> listeners = new …

  void addToolListener(AbstractTool subject, ToolListener listener) {
    listeners.get(subject).add(listener);
  }

  after (ToolEnabledJP jp) {
    for (ToolListener listener : listeners.get(jp.subject))
      listener.setEnabled(jp.enabled);
  }
}
```

Fig. 12. Implementation of the OBSERVER pattern in JHOTDRAW using IIIA (strongly simplified). Boilerplate code is needed to maintain the set of listeners and to dispatch event notification.

5.4 Summary of findings

Table II summarizes our findings. Apart from the fact that join point types are generally usable, the picture is not coherent — the only conclusion we can draw is that all other features of our capture of IIIA are also useful, but not in all applications. This however was to be expected.

Apart from this general observation, we found that in our two case studies of IIIA, using explicit join point creation seemed more convenient than writing pointcuts, even when the latter posed no technical problems. Particularly considering the fact that in our case studies, the average degree of quantification (i.e., the average number of join point shadows matched by pointcuts) is close to 1, and that generally, the pointcut language of ASPECTJ often requires the rewriting of code so that it can be matched, writing generic join point specifications seems rarely worth the effort.[11] The picture is rather different for the standard (text book) examples of ASPECTJ, which usually have much higher degrees of quantification.[12]

The generally low degree of quantification has a like effect on the usefulness of polymorphic pointcuts: the number of classes exhibiting the same join point type is usually quite small, so that the scattering of join point type exhibition as well as that of pointcuts is limited. This counters fears that use of our IIIA will lead to annoyingly verbose programs, in particular ones in which implicit calling information is widely scattered, where

[11] One notable exception is the writing of execution pointcuts, which seems more convenient than an explicit join point creation. See Section 7 on future work for how this can be avoided using standardized join points.

[12] However, a recent study of 11 ASPECTJ programs has shown that the degree of quantification is generally low: only 2% of all pointcuts are homogenous, i.e., match more than one join point shadow [Apel and Batory 2008].

Table II. Frequency of use of IIIA programming constructs

IIIA PROGRAMMING CONSTRUCT	STANDARD (TEXT BOOK) EXAMPLES OF ASPECTJ	BERKELEY DB	JHOTDRAW
join point type	+	+	++
polymorphic pointcut	++	º	+
explicit join point creation	–	++	º
join point subtyping	º	–	+

legend: – = no, º = little, + = good, ++ = strong use

it would be nicer to have it all in one place. In fact, given the low degree of quantification that we found, it seemed more natural to specify join points (close to) where they occurred, rather than (close to) where they are consumed.

Another observation in this vein is that while our definition of IIIA introduces some syntactic overhead (in the form of `exhibits` and `advises` clauses), we felt that this overhead leads to better readable programs, which is basically due to more explicitly stated dependencies. The effect is largely comparable to that of the use of checked exceptions in JAVA: while throwing an exception in the body of a method may seem a sufficient expression of the fact that the method has exceptions, it is the repeating in the method signature that leads to better readability (and that allows type-checking in absence of an implementation).

Last but not least, there appears to be a correlation of the number of join point types in a program and the opportunities for join point subtyping. This could be explained by the fact that the greater the number of join point types in a program, the greater is the chance that these are conceptually related, and lend themselves to organization through subtyping.

6. RELATED WORK

Event-driven programming and publish/subscribe In EDP, registering and unregistering of subscribers usually occur at runtime, whereas in our approach to IIIA they are "woven in" using the weaving mechanism of an aspect-oriented programming language (see, e.g., [Avgustinov et al. 2005; Hilsdale and Hugunin 2004], but also [Rajan et al. 2006] for a viable alternative). Also, the announcement, or firing, of events in EDP is usually explicit, while it is by definition implicit in our approach: as can be seen from Fig. 13, there is no publish statement or explicit call of a corresponding procedure. Types have been introduced to EDP and P/S mainly as filters for subscribers [Eugster 2007]: rather than accepting every event and checking it individually for relevance, a subscriber subscribes only to certain types of events. By contrast, we use types mostly to specify interfaces on the side of the publisher, a purpose that is explicitly declined by proponents of implicit invocation [Garlan and Scott 1993; Notkin et al. 1993]. However, denying interfaces sacrifices modularity, which we want to restore.

Aspect-oriented programming According to most common definitions of AOP, what we suggest is no longer aspect-oriented. For instance, compared to ASPECTJ it does not perform well in the removal of scattered code, and therefore does not modularize crosscutting concerns in the way expected by many in the AOP community. Compared to symmetric approaches such as HYPER/J [Ossher and Tarr 2000], it performs even worse — because of its restriction to implicit invocation, it does not support the merging of classes (or aspects) delivering aspect-wise structure and behaviour (introduction of new features into classes is not supported). However, combining IIIA with mixin techniques is a promising approach to solve this problem [Apel et al. 2008].

```
joinpointtype ItemProducing {}
joinpointtype ConsumerCreation {
  Consumer consumer;
}

class Producer exhibits ItemProducing {
  pointcut ItemProducing : execution(Item produceItem());
  …
}

class Consumer exhibits ConsumerCreation {
  pointcut ConsumerCreation : execution(new(..)) && this(consumer);
  …
}

class Item {…}

aspect Dispatcher advises ItemProducing, ConsumerCreation {
  List consumer = new ArrayList();
  after (ConsumerCreation creation) returning {
    consumer.add(creation.consumer);
  }
  after (ItemProducing producing) returning (Item it) {
    // dispatch item to idle consumer
  }
}
```

Fig. 13. Event-driven consumer/producer communication based on IIIA. The `Consumer` class signals its instantiation, an event of type `ConsumerCreation` parameterized with the new consumer, to the `Dispatcher` aspect. The `Producer` class signals the production of a new item to the `Dispatcher`, which dispatches it to an idle consumer. Note how both `Consumer` and `Producer` remain completely unaware of each other, and also of the dispatcher.

It follows that implementing important standard aspects such as logging or tracing, and also certain design patterns requiring structural introductions, with our proposal is no good idea. Also, in terms of the much-cited quantification and obliviousness characterization [Filman and Friedman 2004], our proposal does not make it as a form of AOP: quantification is restricted to classes declaring to exhibit join points, and obliviousness is compromised to the extent that all classes in which join points may occur must be explicitly tagged as such. In fact, we even go as far as permitting explicit marking of individual join points through the `exhibit new <join point type>` construct, which eliminates obliviousness and quantification completely. On the other hand, since aspects and classes depend on join point types, not on their counterparts, they remain oblivious of each other.

Reduction of AOP to implicit invocation Xu et al. have shown how aspect-oriented programs can be automatically reduced to implicit invocation, so that available model checking approaches designed for implicit invocation can be used for aspect-orientated programs also [Xu et al. 2004]. However, as the authors themselves admit, the practicality of their approach is limited by the practicality of model checking in general: formulation of conditions to be checked is difficult, scalability is poor, and translation of the results (found counterexamples) back to the original input, in this case aspect-oriented programs, is nontrivial. By contrast, we have suggested an intuitive and simple to use type system that lets the compiler make certain checks.

Explicit Join Points In parallel but independent work, Hoffman and Eugster [2007] have elaborated on a notion of explicit join points that is much akin to ours. Syntactically, an explicit join point resembles a static method call qualified by an aspect, but like our join point type instantiation it can be associated with a block of statements the aspect is to advise. As an extension to explicit join points, Hoffman and Eugster allow the adding of pointcuts whose scope is restricted to where they are defined in the base code. However, their relationship of pointcuts and explicit join points is the inverse of ours: while we introduce polymorphic pointcuts as (class) local join point type predicates and add ex-

plicit join point creation as a natural extension (analogous to an anonymous subclass, specifying a pointcut with a single match), the explicit join points of Hoffman and Eugster govern over local pointcuts so that the latter are unavailable without the former. Also, since their explicit join points are typeless, there is no opportunity of subtyping; in fact, in Hoffman and Eugster's approach explicit join points are bound to specific aspects, whereas in ours they are only bound to a join point type and its supertypes, which can be advised by any aspect declaring to do so. This gives us a broadcast semantics, and with it greater decoupling.

Another approach to modular AOP making the shift towards explicit announcement is Rajan's and Leaven's language PTOLEMY [2008]. Like our own work, PTOLEMY relies on event types to decouple the implicit invokers and invokees (the sources and sinks of join points), and on fields of these types as holders of context information. A minor difference is that PTOLEMY binds the arguments of the context to the fields of an event using context variables of the same name; however, this kind of binding may cause problems in case of nested event announcement, when names, but not values of arguments are identical. Similar to Hoffman and Eugster [2007] and also to our anonymous join point subtypes, PTOLEMY's explicit announcement can wrap arbitrary blocks of code, which event handlers can choose to execute.[13] However, unlike our own work, which offers explicit announcement as a supplement to implicit announcement, the quantification property of PTOLEMY seems rather limited: in particular, by relying entirely on the tagging of events in source code using explicit event expressions, it is unclear to us in which way their quantification differs from that of typed EDP [Eugster 2007]. The disjunction of event types offered by PTOLEMY can easily be emulated in EDP and our form of IIIA by supplying separate handlers that invoke identical code.

Unlike in our work and also in ASPECTJ, in PTOLEMY event handlers are methods associated to objects, not aspects, explicitly registered as having an interest in announced events. This symmetry appears to be similar to the approach of classpects [Rajan and Sullivan 2005] (see below), although it appears that no provisions are made in PTOLEMY so as to let different subscriber instances register to different publisher instances. The filtering of events by objects having interest in events published by certain objects only is then left to the subscribing objects, likely imposing significant overhead.

Classpects and EOS-U To achieve greater conceptual integrity, the "classpects" of EOS-U [Rajan and Sullivan 2005] drop the distinction between classes and aspects and let instances advise other instances. However, this requires binding of advising to advised objects, which introduces additional dependencies. By contrast, we have introduced join point instances that are automatically created when a pointcut matches, and let advice operate on these instances as if it were a method of the corresponding join point type (advice is dynamically bound depending on the type of a join point). Our gain in conceptual integrity is therefore comparable. On the other hand, we believe that EOS-U would profit from the typing we suggested: for instance, its `addObject` method could be typed to accept only objects exhibiting the advised join point type.

Crosscutting Interfaces (XPIs) Griswold et al. [2006] suggest the introduction of crosscutting interfaces (XPIs) as interfaces "that base code designers 'implement' and that aspects may depend upon" [Sullivan et al. 2005]. For this, each XPI comes with a "syntactic part" that exposes the signature of named pointcuts, and a "hidden implementation" [Griswold et al. 2006, p. 54], the part that specifies the concrete pointcut expressions. XPIs are enhanced by informal, "semantic" specifications ("design rules") of join points that need to be observed by the maintainers of classes. Note that storing the implementa-

[13] Note that, as with many other ASPECTJ-related approaches, this can lead to the paradoxical situation that the event that triggered the handler actually did not take place.

tion, the pointcuts, in the interface is somewhat unusual (cf. the discussion of attaching pointcuts to join point types in Sections 3.1 and 3.2), but may be seen as technical tribute to ASPECTJ as the language in which XPIs are currently implemented. However, this technicality impairs independent module evolution to a certain extent, since the implementation of the interface is not part of the implementation of the module (so that decoupling reaches only stage (b) in Fig. 3). At the same time, it does not solve many of the problems noted in [Sullivan et al. 2005], which are mostly due to the inability to formulate pointcuts that readily match the intended points in a program. By letting interfaces be implemented polymorphically, that is, per implementing class as we do, XPIs would achieve full modularity (stage (d) in Fig. 3). Also, by letting join points be created explicitly, many of the problems of state-point separation, inaccessible join points, and quantification failure would be solved. From there, however, it is only a small step to our concept of join point types as interfaces.

Open Modules Following Aldrich's influential work [2005], Ongkingco et al. [2006] present an implementation of Open Modules for ASPECTJ. It introduces a module concept as an owning collection of classes that together declare a set of friend aspects (that can freely access all classes of the module) as well as specific pointcuts advertised or exposed (the difference is of little importance here) to aspects. All join points included in the module that are not exposed are invisible from the outside. In addition, a module may expose join points selectively to aspects that it names. This is somewhat comparable to, although still sufficiently different from, our approach in which join points are exposed to aspects that declare to depend on the join points' types. In sharp contrast to our work is that in Open Modules classes remain unaware of the join points they expose, and also of the pointcuts specifying those join points. In particular, in Open Modules à la Ongkingco et al. [2006] the pointcuts used by an aspect cannot be adapted and maintained on a per class basis, thereby limiting independent evolution of aspects and base classes to a certain extent. Also, the ability to declare friend aspects of a module, while allowing such things as debugging via aspects, provides for unspecified (implicit) interfaces to the module, which basically implies that friend aspects are part of the modules whose classes they advise.

Spectators and assistants Clifton and Leavens [2002] use the `accept` keyword to let classes declare that they admit advice from the aspects listed thereafter. In Fig. 3, this would correspond to (a) with bidirectional dependencies. We are taking a different route: by introducing join point types as middle men between aspects and their targets, and by introducing class-wise polymorphic pointcuts, we reach the degree of decoupling shown in Fig. 3 (d). Clifton and Leavens [2002; 2003] further distinguish between spectators (aspects that may only observe) and assistants (aspects that can actually change state). Using our approach, and would JAVA offer a modifier similar to C++'s `const`, we could allow declaring single fields of a join point type as being observable only, or as being changeable, thereby granting finer-grained access control. On the other hand, preventing direct write access to objects cannot prevent the behaviour changing interception of methods. For a more detailed discussion of how potentially interfering aspects can be separated from "harmless advice", see [Dantas and Walker 2006].

Modular aspects with ownership (MAO) In subsequent work, the distinction between spectators and assistants and the enforcement of corresponding access policies have been implemented using an ownership type and effect system [Clifton et al. 2007]. Using this system, it can be statically checked whether an aspect can make changes to heap objects it does not itself own (usually objects of the base program). Together with restrictions on aspects regarding their alteration of the advised program's control flow, the number of aspect that need to be considered when reasoning about a particular piece of code can be

reduced significantly. Compared to our introduction of join point types as bilateral interfaces between aspects and base code (with references to the interfaces in both the aspects and the base code), the interfaces of modular aspects with ownership express obligations only for the aspect, and need to be inspected by designers of base code for potential effects.

Aspect-implied interfaces In their effort to restore modularity of AOP, Kiczales and Mezini [2005] argue that "aspects cut new interfaces through the primary module structure", and that a tool can compute these interfaces once a system has been assembled. This means that a module is no longer sovereign over its own interfaces — rather, they are forced upon it by system composition. It follows immediately that modules cannot be changed independently of their use in a particular assembly, simply because it is unclear which interfaces to keep constant. This in turn hampers reuse in all cases in which a module is to be used in more than one composition. By contrast, what we have suggested here is much more conservative: we require that all interfaces of a module be made explicit at module design time, so that programmers can observe them while doing whatever they need to do, independently of each other. In our approach to IIIA, a change to an aspect can never require a change of a class; changing the fields of a join point type, the interface between a class and its advising aspects, may require changes (in both the class and its aspects), but these are enforced by the compiler.

Adding Polymorphism to ASPECTJ Ernst and Lorenz [2003] noted that the polymorphism present in ASPECTJ is basically ad-hoc; all available inclusion polymorphism is that of the base language (JAVA). In order to introduce late binding of advice, the authors require some kind of advice grouping, so that a binding algorithm can "choose exactly one most specific advice and invoke it, ignoring all the others in the group (they are being overridden)." [Ernst and Lorenz 2003]. By our introduction of join point types and subtypes, and by linking advice to join point types (providing some kind of "advice signatures" [Ernst and Lorenz 2003]), we have installed such groupings. However, with the language and its translation to ASPECTJ as defined above, advice is still bound at compile time; in particular, we have not yet explored whether and how our approach could open the door for separate compilation.

In a different line of work, Apel et al. proposed the notion of *aspect refinement* to introduce polymorphism to ASPECTJ [2007]. Using aspect refinement, advice can be named and thus overridden in specialized aspects, where it can also be bound to specialized pointcuts. A specialized pointcut may refer to its base using the `super` keyword, which is similar to our specialization of pointcuts as type predicates of join point subtypes. Even though in our present work we did not consider specialization of aspects or advice, letting join point handlers (advice) of a specialized join point type refer to handlers of the more general type seems like a natural extension.

Type-theoretic interpretation of pointcuts and advice In [Ligatti et al. 2006], Ligatti, Walker, and Zdancewic present formal semantics for an idealized AOPL. For this, they extend the simply-typed lambda calculus with two new abstractions covering join points, pointcuts, and advice, and prove type safety for this calculus. They present a small functional language, MINIAML, and show how this maps to the core calculus. MINIAML has some similarities with our language, most prominently that it allows scoping of advice: functions can be hidden from advice, thereby allowing "programmers to retain some control over basic information hiding and modularity principles in the presence of aspects." [Ligatti et al. 2006] The mapping of MINIAML to the core calculus is non-trivial; we expect a corresponding mapping of our own language, although certainly desirable to prove the soundness of our type system, to be no easier. On the other hand, what we have deliv-

ered can be immediately tried out in practical settings, allowing the community to test and improve it until it is maximally useful.

Stratified aspects In our own previous work, we proposed [Forster and Steimann 2006] and implemented [Bodden et al. 2006] an extension of ASPECTJ that adds type levels to its join points and aspects. In the resulting language, type information in a program is partly implicit, and for the rest consists of `meta` modifiers attached to aspects and pointcuts. According to this type system, all join points contained in classes are of type level 0, all in aspects of type level 1, all in aspects declared with a single meta modifier of type level 2 and so forth. Pointcuts to range over join points of type level 0 remain unmodified, while those to range over type level 1 and higher have to be modified with a corresponding number of `meta` modifiers. This allows us to build towers of aspects as advertised in [Rajan and Sullivan 2005], albeit on the class rather than the instance level (cf. below). As can easily be seen, our current type system can emulate our previous one, simply by dividing the set of join point types into disjoint subsets each associated with a type level, and requiring that aspects advise only join point types from levels lower than the join points they themselves exhibit (if that is what they do; aspects exhibiting join points are not discussed in this paper). In fact, it should even be possible to automatically construct the type strata from the `exhibits`/`advises` relationships found in a program, and to report a typing error (or warning) should the relationship contain circles (potentially leading to self-application and recursion).

Fine-grained generic aspects Rho et al. [2006] propose a mechanism for capturing arbitrary join points in the execution and structure of a program. Fine-grained pointcuts use a combination of metaprogramming and logic programming to allow the programmer to precisely select join points. While the fine granularity increases the expressive power of advice compared to present aspect-oriented languages, it decreases modularity further since more details of a module's implementation are exposed.

Test-based pointcuts for robust and fine-grained join point specification Sakurai and Masuhara [2008] present a pointcut mechanism for selecting join points using the unit test cases that are associated with the base program. Test-based pointcuts can be used to distinguish between different execution histories of a method and, thus, are more powerful for the specification of dynamic join points. Each test-based pointcut refers to one or more unit test methods of a unit test class. Hence, the unit test class is like an interface between the base program's execution and the advices. Typing and subtyping are not addressed in this work. Type checking is done like in conventional ASPECTJ-like languages and join points do not have types that can be checked against advice.

Managing the evolution of aspect-oriented software with model-based pointcuts Kellens et al. [2006] introduce model-based pointcuts in order to decouple the implementation of a base program from its aspects. Instead of referring to the identifiers of the elements of a base program, model-based pointcuts refer to a conceptual model of the base program. The advantage is that the base program's implementation may change without breaking the pointcuts, as long as the model remains unchanged and consistent with the intention of the base program. This burdens the programmer to make sure that changes do not invalidate the model. The conceptual model is a kind of interface between program and advice. However, type checking is not possible since the conceptual model is at a different level of abstraction, usually written in another language with different type structure.

Event patterns Douence, Motelet, and Südholt [2001] have suggested looking at sequences (or temporal patterns) of events for identifying the join points aspects are to advise. Like ourselves, they equate points of interest with events; however, they extend a pointcut language such as ASPECTJ's to allow the specification of patterns to be matched

(also referred to as tracematches [Allan et al. 2005]). For this purpose, they introduce a single event type, `Event`, whose instances consist of a name tag and a time stamp. Using this event type in pattern specifications increases the expressiveness of the pointcut language (by adding a temporal dimension), but it does not alleviate problems of modularity in any way.

7. FUTURE WORK

During our experiments with IIIA and our compiler, a number of directions for future work became apparent. The following seem most important to us.

- It may make sense to combine interface implementation with join point exhibition: a class offering a method published in an interface it declares to implement may at the same time declare to announce whenever this method is being executed. Rather than specify a corresponding execution pointcut in every class implementing this interface (as is currently required), it might be more convenient to specify this pointcut in the interface and have it inherited by the implementing classes. To avoid untoward change dependencies (as discussed at the end of Section 3.5), the pointcut could be standardized as always binding the parameters of the method to the fields of a corresponding join point type.
- Another issue that should be investigated is whether, rather than letting the advice choose its kind (i.e., before, after, etc.), the kind of advice should be associated to the join point type (so that both advised class and advising aspect must agree on the kind). In fact, one could argue that a class should not only have control over which join points it exhibits, but also whether it admits advice before or after a join point is executed, and whether the advice is allowed to change the context in which it executes (around advice). Attaching the kind to the join point type poses new questions, however, in particular with respect to subtyping: how can subtypes change (extend, restrict) the kind of a join point type without breaking contracts of their supertypes?
- A rather minor open issue is the problem of join point exhibition in anonymous inner classes: because the JAVA syntax of these classes leaves no room for an `exhibits` clause, and since join point exhibition is not inherited by subclasses, there is no consistent way for making these classes announce join points. We currently work around this problem by assuming the `exhibits` clause of the superclass (making an exception to the rule that these clauses are not inherited), but then this always requires the specification of join points within the anonymous class, either through pointcut or through an explicit announcement. An alternative would be to not require an `exhibits` clause for anonymous classes, but this is not very satisfactory, either.
- Last but not least, it may make sense to investigate the combination of join point types with traits [Ducasse et al. 2006]. In fact, since join point types can be viewed (and are currently implemented in our compiler) as classes with state, but no behaviour, attaching behaviour through traits, rather than aspects, may be a viable alternative. Together with the fact that mixin-like constructs can replace for the introductions offered by ASPECTJ [Apel et al. 2008], and that implicit invocation can be standardized as sketched above, our IIIA could drop much of the dependence on AOP and its language constructs.

8. CONCLUSION

Due to the lack of explicit interfaces, implicit invocation with implicit event announcement mechanisms such as those offered by aspect-oriented programming languages suffer from serious modularity problems. Inspired by how typed exceptions are declared in JAVA, and how its interfaces-as-types allow for polymorphic implementations while at the same time decoupling callers from the called, we have introduced the notion of *join*

point types as interfaces between producers and consumers of events. Borrowing the pointcut language from ASPECTJ, the type predicates of our join point types are defined as class-local, polymorphic pointcuts. Join point types extend naturally to subtyping and to explicit join points as anonymous subtypes. Applications of the so extended, fully modular language are the same as that for other implicit invocation mechanisms with implicit or explicit event announcement, such as (database) triggers or occurrences of the EVENT NOTIFICATION [Riehle 1996] and OBSERVER [Gamma et al. 1995] patterns. Its limitations are clearly cases in which the publisher should remain unaware of the fact that it publishes. This includes, for practical reasons, some of the most prominent applications of aspect-oriented programming, in particular all extensively crosscutting concerns such as logging or tracing.

ACKNOWLEDGEMENTS

The authors are indebted to Eric Bodden for his many comments on an earlier version of this paper, as well as to the anonymous reviewers for their critical questions and helpful suggestions.

The work of Sven Apel is being supported in part by the German Research Foundation (DFG), project number AP 206/2-1.

REFERENCES

ALDRICH, J. 2005. Open modules: Modular reasoning about advice. In *Proceedings of the 19th European Conference on Object-Oriented Programming* (Glasgow, UK). 144–168.

ALLAN, C., AVGUSTINOV, P., CHRISTENSEN, A. S., HENDREN, L., KUZINS, S., LHOTÁK, O., DE MOOR, O., SERENI, D., SITTAMPALAM, G., AND TIBBLE, J. 2005. Adding trace matching with free variables to AspectJ. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Diego, CA, USA, October 16–20, 2005).

APEL, S., KÄSTNER, C., LEICH, T., AND SAAKE, G. 2007. Aspect Refinement — Unifying AOP and stepwise refinement. *Journal of Object Technology* 6, 9, 13–33.

APEL, S., LEICH, T., AND SAAKE, G. 2008. Aspectual Feature Modules. *IEEE Trans. Softw. Eng.* 34, 2 (Mar. 2008), 162–180.

APEL, S. 2010. *How AspectJ is Used: An Analysis of Eleven AspectJ Programs*. Journal of Object Technology 9, 1.

AVGUSTINOV, P., CHRISTENSEN, A. S., HENDREN, L., KUZINS, S., LHOTÁK, J., LHOTÁK, O., DE MOOR, O., SERENI, D., SITTAMPALAM, G., AND TIBBLE, J. 2005. Optimising AspectJ. *SIGPLAN Not.* 40, 6 (Jun. 2005), 117–128.

AVGUSTINOV, P., CHRISTENSEN, A. S., HENDREN, L. J., KUZINS, S., LHOTÁK, J., LHOTÁK, O., DE MOOR, O., SERENI, D., SITTAMPALAM, G., AND TIBBLE, J. 2006. abc: an extensible AspectJ compiler. *Transactions on Aspect-Oriented Software Development* 1, 293–334.

BRACHA, G. AND COOK, W. 1990. Mixin-based inheritance. In *Proceedings of the European Conference on Object-Oriented Programming/Proceedings on Object-Oriented Programming Systems, Languages, and Applications* (Ottawa, Canada). OOPSLA/ECOOP '90. ACM, New York, NY, 303–311.

BOBROW, D. G., DEMICHIEL, L. G., GABRIEL, R. P., KEENE, S. E., KICZALES, G., AND MOON, D. A. 1988. Common Lisp Object System specification. *SIGPLAN Not.* 23, SI (Sep. 1988), 1–142.

BODDEN, E., FORSTER, F., AND STEIMANN, F. 2006. Avoiding infinite recursion with stratified aspects. In *Proceedings of NODe/GSEM* (Erfurt, Germany) NODe/GSEM '06. GI-Edition Lecture Notes in Informatics P-88, 49–64.

CLIFTON, C. AND LEAVENS, G. 2002. Observers and assistants: A proposal for modular aspect-oriented reasoning. In *Workshop on Foundations of Aspect-Oriented Languages*. FOAL 2002. 33–44.

CLIFTON, C. AND LEAVENS, G. 2003. Obliviousness, modular reasoning, and the behavioral subtyping analogy. In *Software Engineering Properties of Languages for Aspect Technologies*. SPLAT. Workshop at AOSD 2003.

CLIFTON, C., LEAVENS, G.T., AND NOBLE, J. 2007. MAO: Ownership and effects for more effective reasoning about aspects. In *Proceedings of the European Conference on Object-Oriented Programming*. ECOOP 2007. 451–475.

COELHO, R., RASHID, A., GARCIA, A., FERRARI, F., CACHO, N., KULESZA, U., STAA, A., AND LUCENA, C. 2008. Assessing the impact of aspects on exception flows: An exploratory study. In *Proceedings of the 22nd European Conference on Object-Oriented Programming* (Paphos, Cypress, July 07–11, 2008). J. VITEK, Ed. Lecture Notes In Computer Science, vol. 5142. Springer-Verlag, Berlin, Heidelberg, 207–234.

DANTAS, D. S. AND WALKER, D. 2006. Harmless advice. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Charleston, South Carolina, USA, January 11–13, 2006). POPL '06. ACM, New York, NY, 383–396.

DOUENCE, R., MOTELET, O., AND SÜDHOLT, M. 2001. A formal definition of crosscuts. In *Proceedings of the Third international Conference on Metalevel Architectures and Separation of Crosscutting Concerns* (September 25–28, 2001). A. YONEZAWA AND S. MATSUOKA, Eds. Lecture Notes In Computer Science, vol. 2192. Springer-Verlag, London, 170–186.

DUCASSE, S., NIERSTRASZ, O., SCHÄRLI, N., WUYTS, R., AND BLACK, A. P. 2006. Traits: A mechanism for fine-grained reuse. *ACM Trans. Program. Lang. Syst.* 28, 2 (Mar. 2006), 331–388.

EICHBERG, M., MEZINI, M,. AND OSTERMANN, K. 2004. Pointcuts as functional queries. In WEI-NGAN CHIN, editor, *Programming Languages and Systems: Second Asian Symposium*, APLAS 2004, Taipei, Taiwan, November 2004. Lecture Notes in Computer Science, 366–381.

ELRAD, T., FILMAN, R. E., AND BADER, A. 2001. Aspect-oriented programming: Introduction. *Commun. ACM* 44, 10 (Oct. 2001), 29–32.

ERNST, E. AND LORENZ, D. H. 2003. Aspects and polymorphism in AspectJ. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development* (Boston, Massachusetts, March 17–21, 2003). AOSD '03. ACM, New York, NY, 150–157.

ESWARAN, K. P. 1976. *Specifications, Implementations and Interactions of a Trigger Subsystem in an Integrated Database System*, IBM Research Report RJ1820 (Nov. 1976).

EUGSTER, P. T., FELBER, P. A., GUERRAOUI, R., AND KERMARREC, A. 2003. The many faces of publish/subscribe. *ACM Comput. Surv.* 35, 2 (Jun. 2003), 114–131.

EUGSTER, P. 2007. Type-based publish/subscribe: Concepts and experiences. *ACM Trans. Program. Lang. Syst.* 29, 1 (Jan. 2007), 6.

FILMAN, R. E. AND FRIEDMAN, D. P. 2004. Aspect-oriented programming is quantification and obliviousness In *Aspect-Oriented Software Development*, R. E. FILMAN, T. ELRAD, S. CLARKE, AND M. ASKIT, Eds. Addison-Wesley Longman, Amsterdam.

FORSTER, F. AND STEIMANN, F.: AOP and the antinomy of the liar. In *Workshop on the Foundations of Aspect-Oriented Languages*. FOAL 2006. 47–56.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

GARLAN, D. AND SCOTT, C. 1993. Adding implicit invocation to traditional programming languages. In *Proceedings of the 15th international Conference on Software Engineering* (Baltimore, Maryland, United States, May 17–21, 1993). International Conference on Software Engineering. IEEE Computer Society Press, Los Alamitos, CA, 447–455.

GARLAN, D. AND SHAW, M. 1994. *An Introduction to Software Architecture*. Technical Report. UMI Order Number: CS-94-166., Carnegie Mellon University.

GERACI, A. 1991 *IEEE Standard Computer Dictionary: Compilation of IEEE Standard Computer Glossaries*. Institute of Electrical and Electronics Engineers Inc., The.

GRISWOLD, W. G., SULLIVAN, K., SONG, Y., SHONLE, M., TEWARI, N., CAI, Y., AND RAJAN, H. 2006. Modular software design with crosscutting interfaces. *IEEE Softw.* 23, 1 (Jan. 2006), 51–60.

GUDMUNDSON, S. AND KICZALES, G. 2001. Addressing practical software development issues in AspectJ with a pointcut interface. In *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 international Conference on Aspect-Oriented Software Development* (Boston, Massachusetts, March 17–21, 2003). AOSD '03. ACM, New York, NY, 60–69.

HILSDALE, E. AND HUGUNIN, J. 2004. Advice weaving in AspectJ. In *Proceedings of the 3rd international Conference on Aspect-Oriented Software Development* (Lancaster, UK, March 22–24, 2004). AOSD '04. ACM, New York, NY, 26–35.

HOFFMAN, K. AND EUGSTER, P. 2007. Bridging Java and AspectJ through explicit join points. In *Proceedings of the 5th international Symposium on Principles and Practice of Programming in Java* (Lisboa, Portugal, September 05–07, 2007). PPPJ '07, vol. 272. ACM, New York, NY, 63–72.

KÄSTNER, C., APEL, S., AND BATORY, D. 2007. A Case Study Implementing Features Using AspectJ. In *Proceedings of the 11th international Software Product Line Conference* (September 10–14, 2007). International Conference on Software Product Line. IEEE Computer Society, Washington, DC, 223–232.

KELLENS, A., MENS, K., BRICHAU, J., AND GYBELS, K. 2006. Managing the evolution of aspect-oriented software with model-based pointcuts. In *Proceedings of the European Conference on Object-Oriented Programming* (ECOOP 2006), Lecture Notes in Computer Science, Vol. 4067. Springer, Berlin. 501–525.

KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C., LOINGTIER, J., AND IRWIN, J. 1997. Aspect-oriented programming. In *European Conference on Object-Oriented Programming* (ECOOP 1997). 220–242.

KICZALES, G. AND MEZINI, M. 2005. Aspect-oriented programming and modular reasoning. In *Proceedings of the 27th international Conference on Software Engineering* (St. Louis, MO, USA, May 15–21, 2005). ICSE '05. ACM, New York, NY, 49–58.

LEE, K., KANG, K. C., KIM, M., AND PARK, S. 2006. Combining Feature-Oriented Analysis and Aspect-Oriented Programming for Product Line Asset Development. In *Proceedings of the 10th international on*

Software Product Line Conference (August 21–24, 2006). International Conference on Software Product Line. IEEE Computer Society, Washington, DC, 103–112.

LIGATTI, J., WALKER, D., AND ZDANCEWIC, S. 2006. A type-theoretic interpretation of pointcuts and advice. *Sci. Comput. Program.* 63, 3 (Dec. 2006), 240–266.

LISKOV, B. H. AND WING, J. M. 1994. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.* 16, 6 (Nov. 1994), 1811–1841.

LOPES, C. V., DOURISH, P., LORENZ, D. H., AND LIEBERHERR, K. 2003. Beyond AOP: toward naturalistic programming. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Anaheim, CA, USA, October 26–30, 2003). OOPSLA '03. ACM, New York, NY, 198–207.

MADSEN, O. L. AND MØLLER-PEDERSEN, B. 1989. Virtual classes: a powerful mechanism in object-oriented programming. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications* (New Orleans, Louisiana, United States, October 02–06, 1989). OOPSLA '89. ACM, New York, NY, 397–406.

MASUHARA, H. AND KAWAUCHI, K. 2003. Dataflow pointcut in aspect-oriented programming. In *Proceedings of The First Asian Symposium on Programming Languages and Systems* (APLAS'03) Lecture Notes in Computer Science 2895, 105–121.

MEYER, B. 1997 *Object-Oriented Software Construction (2nd Ed.)*. Prentice-Hall, Inc.

MIKHAJLOV, L. AND SEKERINSKI, E. 1998. A study of the fragile base class problem. In *Proceedings of the 12th European Conference on Object-Oriented Programming* (July 20–24, 1998). E. JUL, Ed. Lecture Notes In Computer Science, vol. 1445. Springer-Verlag, London, 355–382.

MURPHY, G. C., LAI, A., WALKER, R. J., AND ROBILLARD, M. P. 2001. Separating features in source code: an exploratory study. In *Proceedings of the 23rd international Conference on Software Engineering* (Toronto, Ontario, Canada, May 12–19, 2001). International Conference on Software Engineering. IEEE Computer Society, Washington, DC, 275–284.

NOTKIN, D., GARLAN, D., GRISWOLD, W. G., AND SULLIVAN, K. J. 1993. Adding implicit invocation to languages: Three approaches. In *Proceedings of the First JSSST International Symposium on Object Technologies For Advanced Software* (November 04–06, 1993). S. Nishio and A. Yonezawa, Eds. Lecture Notes In Computer Science, vol. 742. Springer-Verlag, London, 489–510.

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 international Conference on Aspect-Oriented Software Development* (Bonn, Germany, March 20–24, 2006). AOSD '06. ACM, New York, NY, 39–50.

OSSHER, H. AND TARR, P. 2000. Hyper/J: multi-dimensional separation of concerns for Java. In *Proceedings of the 22nd international Conference on Software Engineering* (Limerick, Ireland, June 04–11, 2000). ICSE '00. ACM, New York, NY, 734–737.

OSTERMANN, K., MEZINI, M., AND BOCKISCH, C. 2005. Expressive pointcuts for increased modularity. In *Proceedings of the 19th European Conference on Object-Oriented Programming.* ECOOP 2005. LNCS 3586, Springer-Verlag, 214–240.

RAJAN, H. AND SULLIVAN, K. 2003. Eos: instance-level aspects for integrated system design. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT international Symposium on Foundations of Software Engineering* (Helsinki, Finland, September 01–05, 2003). ESEC/FSE-11. ACM, New York, NY, 297–306.

RAJAN, H. AND SULLIVAN, K. J. 2005. Classpects: unifying aspect- and object-oriented language design. In *Proceedings of the 27th international Conference on Software Engineering* (St. Louis, MO, USA, May 15–21, 2005). ICSE '05. ACM, New York, NY, 59–68.

RAJAN, H., DYER, R., HANNA, Y., AND NARAYANAPPA, H. 2006. Preserving separation of concerns through compilation. In *Software Engineering Properties of Languages and Aspect Technologies* (March 2006) SPLAT '06. L. BERGMANS, J. BRICHAU, AND E. ERNST, Eds., workshop affiliated with AOSD 2006.

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* (Paphos, Cypress, July 07–11, 2008). J. VITEK, Ed. Lecture Notes In Computer Science, vol. 5142. Springer-Verlag, Berlin, Heidelberg, 155–179.

REISS, S. P. 1990. Interacting with the FIELD environment. *Softw. Pract. Exper.* 20, S1 (Jun. 1990), 89–115.

RIEHLE, D. 1996. The event notification pattern—integrating implicit invocation with object-orientation. *Theor. Pract. Object Syst.* 2, 1 (Nov. 1996), 43–52.

RHO, T., KNIESEL, G. AND APPELTAUER, M. 2006. Fine-grained generic aspects. In *Workshop on Foundations of Aspect-Oriented Languages*. FOAL'06. G. LEAVENS, C. CLIFTON, R. LÄMMEL, AND M. MEZINI, Eds., workshop affiliated with AOSD 2006.

SAKURAI, K. AND MASUHARA, H. 2008. Test-based pointcuts for robust and fine-grained join point specification. In *Proceedings of the 7th international Conference on Aspect-Oriented Software Development* (Brussels, Belgium, March 31–April 04, 2008). AOSD '08. ACM, New York, NY, 96–107.

STEIMANN, F. AND MAYER, P. 2005. Patterns of interface-based programming. *Journal of Object Technology* 4, 5, 75–94.

STEIMANN, F. 2006. The paradoxical success of aspect-oriented programming. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA, October 22–26, 2006). OOPSLA '06. ACM, New York, NY, 481–497.

STÖRZER, 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* (September 25–30, 2005). ICSM. IEEE Computer Society, Washington, DC, 653–656.

SULLIVAN, K. AND NOTKIN, D. 1990. Reconciling environment integration and component independence. *SIGSOFT Softw. Eng. Notes* 15, 6 (Dec. 1990), 22–33.

SULLIVAN, K. J. AND NOTKIN, D. 1992. Reconciling environment integration and software evolution. *ACM Trans. Softw. Eng. Methodol.* 1, 3 (Jul. 1992), 229–268.

SULLIVAN, K., GRISWOLD, W. G., SONG, Y., CAI, Y., SHONLE, M., TEWARI, N., AND RAJAN, H. 2005. Information hiding interfaces for aspect-oriented design. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT international Symposium on Foundations of Software Engineering* (Lisbon, Portugal, September 05–09, 2005). ESEC/FSE-13. ACM, New York, NY, 166–175.

TARR, P., OSSHER, H., HARRISON, W., AND SUTTON, S. M. 1999. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 21st international Conference on Software Engineering* (Los Angeles, California, United States, May 16–22, 1999). ICSE '99. ACM, New York, NY, 107–119.

VAN DEURSEN, A., MARIN, M., AND MOONEN, L. 2005. AJHotDraw: A showcase for refactoring to aspects. In *Proceedings of the AOSD Workshop on Linking Aspects and Evolution.* LATE '05. CWI, Amsterdam, The Netherlands.

WIRTH, N. 1988. Type extensions. *ACM Trans. Program. Lang. Syst.* 10, 2 (Apr. 1988), 204–214.

XU, J., RAJAN, H. AND SULLIVAN, K. 2004. Understanding aspects via implicit invocation. In *Proceedings of the 19th IEEE international Conference on Automated Software Engineering* (September 20–24, 2004). Automated Software Engineering. IEEE Computer Society, Washington, DC, 332–335.