Adding Open Modules to AspectJ

Neil Ongkingco[1], Pavel Avgustinov[1], Julian Tibble[1],
Laurie Hendren[2], Oege de Moor[1], Ganesh Sittampalam[1]

[1] Programming Tools Group
University of Oxford
United Kingdom

[2] Sable Research Group
McGill University
Montreal, Canada

ABSTRACT

AspectJ does not provide a mechanism to hide implementation details from advice. As a result, aspects are tightly coupled to the implementation of the code they advise, while the behaviour of the base code is impossible to determine without analysing all advice that could modify its behaviour.

The concept of *open modules* is proposed by Aldrich to solve the problems that arise from unrestricted advice. Defined for a small functional language, it provides an encapsulation construct that allows an implementation to limit the set of points to which external advice may apply.

We present an adaptation of open modules for AspectJ. We expand open modules to encompass the full set of pointcut primitives for AspectJ, extend its method of module composition to include the ability to open up a module, and describe the implementation of the design as an extension of the AspectBench compiler. We also provide an example of the use of open modules on a substantial AspectJ program to show how it would fit into existing AspectJ projects.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Languages, Design, Theory

Keywords

Encapsulation, modularity, aspect-oriented programming

1. INTRODUCTION

In AspectJ, aspects observe the execution of a base Java program. When events of interest happen, the aspect executes some extra code of its own. The intercepted events are specified via patterns called *pointcuts*; the extra code is called *advice*; and the events are named *joinpoints* [10].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
AOSD 06, March 20–24, 2006, Bonn, Germany
Copyright 2006 ACM 1-59593-300-X/06/03 ...$5.00.

The interception mechanism of AspectJ provides no explicit means for hiding implementation detail. As a consequence, the use of aspects can be quite brittle: a change in the advised code can easily lead to unwanted effects, both because a joinpoint no longer matches, or because the advice now intercepts too many joinpoints [16]. The problem is thus one of responsibility: must the maintainer of the base program preserve joinpoint behaviour, or is it the task of the aspect author to adapt his aspect whenever the basecode changes?

A very promising solution was put forward by Jonathan Aldrich: he suggested that any exposed joinpoints be declared in the interface of a module (a module being a collection of classes) [3]. Their exposition in the interface means that the module maintainer undertakes to preserve the joinpoint behaviour relative to aspects which are not part of the module itself. Aldrich's design is for a small, purely functional language and a module system akin to that of ML, augmented with around advice on function calls.

In this paper, we adapt and extend Aldrich's design to the full AspectJ language, to allow ourselves and others to experiment with its use in practice. The contributions of this work are these:

Extend the notion of open modules to full AspectJ: We have pushed Aldrich's concept of open modules in three directions. First, while Aldrich handled only *call* pointcuts in his example language, we generalise to arbitrary AspectJ pointcuts. Second, when introducing open modules into AspectJ we needed to make sure that they can be introduced into an AspectJ project without changing the existing code. In particular, we prove that modules never lead to precedence conflicts between aspects. Indeed, we show that open modules define a total order on aspect precedence. Third, we demonstrate how open modules reduce to a very simple hiding construct associated with a class, thus providing an 'intermediate representation' for information hiding constructs in AspectJ.

Define appropriate notions of module composition: Aldrich's approach to module composition focused on the notion that one wants a construct to further *restrict* an existing module, and we also support this notion. However, we feel that it is also useful to have a construct that *opens* an existing module by exposing more joinpoints. This opening construct allows an aspect author to seize responsibility for the maintenance of joinpoint behaviour in other parts of the system; a

typical use would be a debugging aspect. We demonstrate the utility of module opening through a number of usage scenarios.

Another important aspect of our approach is that we demonstrate that these two forms of composition, restriction and opening, can both be reduced to the same simple intermediate representation.

Full implementation for AspectJ: We have fully implemented our design using *abc*, the extensible research compiler for AspectJ [2]. To our knowledge this is the first implementation of open modules for a full-fledged aspect-oriented language. We feel that having such an implementation will allow us, and other researchers, to experiment with open modules in order to determine the strengths and weaknesses of the approach.

The structure of this paper is as follows. We first discuss the rationale for open modules, and the way we have added them to AspectJ, through a number of small but representative examples in Section 2. In Section 3 we give a more precise account of our design for adding open modules to AspectJ. In particular, we start with a very simple hiding construct, augment it to a feature more similar to the one originally proposed by Aldrich, and then go on to discuss the different forms of composition. At each step of this development, we show how more complex forms can be expressed in more primitive terms, thus proving our 'intermediate representation' result. In Section 4 we illustrate our design for open modules with reference to a more substantial example, that involves eight different aspects. We then turn to some formal properties of our design, in particular regarding the nettly problem of aspect precedence in Section 5. Here we prove that subject to some very natural conditions, our new constructs do not introduce precedence conflicts — an important property when composing large systems with many aspects. This paper is primarily a proposal for features to help tame the power of AspectJ, but it is also a substantial exercise in extending an AspectJ compiler. In Section 6 we discuss the challenges we encountered with implementing open modules in the AspectBench Compiler, and how these obstacles were overcome. There is a wealth of alternative proposals for similar features (although none seems to be implemented for the full AspectJ language), and we discuss these in Section 7. We conclude in Section 8.

2. RATIONALE

While aspects provide a way to encapsulate cross-cutting features in a single construct, they are so tightly coupled with the implementation of the classes that they violate another facet of modularity: that of hiding the implementation behind a well defined interface. In the current state of AspectJ, a piece of advice declared in any aspect may be applied to any joinpoint in the entire program, effectively bypassing class interfaces.

The lack of interfaces makes aspects vulnerable to any changes in the classes to which they apply. We use a short example to demonstrate this vulnerability. Figure 1 shows a simple Figure class, which is just a collection of points and a translate method that moves the points by a specified displacement.

Suppose a replay feature is implemented using an aspect, as in Figure 2. The aspect would intercept all calls to Figure.translate and store the translations in a list for replaying. Note that the advice is very tightly coupled to the call to translate, and any change to the implementation of Figure that changes the pattern of calls to translate will break the aspect. For example, if the implementation of Figure was changed such that the list can contain other figures as well as points (as in Figure 3), the behaviour of the replay aspect would change drastically. The advice in ReplayAspect would match both the external call to translate as well as the call to translate inside Figure. This leads to duplicate entries in the replay list.

```
public class Figure {
    List elements;
    public Figure translate(int dx, int dy) {
        for (Iterator iter = elements.iterator();
              iter.hasNext(); ) {
            Point elem = (Point)(iter.next());
            elem.translate(dx,dy);
        }
        return this;
    }
}
```

Figure 1: Simple Figure Class

```
aspect ReplayAspect {
    pointcut translate(int dx, int dy):
        call(* Figure.translate(int, int)) && args(dx,dy);

    LinkedList moves = new LinkedList();

    before(int x, int y, Figure fig) :
        translate(x,y) && target(fig){
        //Store fig, x and y in the moves list
    }
}
```

Figure 2: Replay Aspect

That such a seemingly innocuous change to Figure could change the behaviour of the program in an unexpected manner seems to violate the encapsulation that the class is expected to provide. As there is no well-defined interface between Figure and its client aspects, all aspects that apply to it would need to be checked before any modifications are made. This makes the evolution of base code difficult. The problem is made worse if Figure were part of a third-party library where the source code is unavailable. In such a case, it would be very difficult to diagnose the problem as the implementation details of Figure would be hidden.

This particular problem can be solved by specifying an interface that enforces the condition that aspects only apply to external calls to translate. Doing so will cause the advice in ReplayAspect to fail when matching with the call to Figure.translate inside Figure.

Figure 4 shows an interface between Figure and its client aspects. The *module* contains the class Figure, as well as an **advertise** declaration that specifies which joinpoints in Figure are available to advice. The **advertise** declaration is used to expose *external* calls to Figure.translate, that is, calls that match the pointcut

```
call(Figure Figure.translate(int, int)) &&
    !within(Figure)
```

This solves the problem of duplicate entries in ReplayAspect.

```
public class Figure {
    public Figure translate(int x, int y) {
        for (Iterator iter = elements.iterator();
             iter.hasNext(); ) {
            Object elem = iter.next();
            if (elem instanceof Point) {
                ((Point)elem).translate(x,y);
            }
            else if (elem instanceof Figure) {
                ((Figure)elem).translate(x,y);
            }
        }
        return this;
    }
}
```

Figure 3: Modified Figure class

```
module FigureModule {
    class Figure || Point;
    friend DebuggingAspect;
    advertise : call(Figure Figure.translate(int, int));
    expose : call(* Point.translate(int, int)) &&
        within(Figure);
}
```

Figure 4: Figure Module

It may become necessary to expose more than joinpoints that are outside Figure. If, for example, there were an aspect that computes the total cost of a figure translation by counting the number of individual Point.translate calls in Figure, such as Figure 5 , it would need to be able to apply advice to joinpoints internal to Figure.

Some aspects, such as debugging aspects, are by their nature very intrusive and would require access to all of the joinpoints of a module. As such, these aspects might be severely hindered by the visibility constraints that have been outlined above. To allow for these types of aspects, we define the **friend** keyword, which declares that an aspect is to be allowed full access to all the joinpoints that belong to the members of a module. In FigureModule, the aspect DebuggingAspect is declared to be a friend of the module, and thus has full access to the joinpoints in the module.

```
aspect TranslateCost {
    int total = 0;

    pointcut translate():
        call(* Point.translate(int, int)) && within(Figure);
    before() : translate() {
        total++;
    }
}
```

Figure 5: Figure Translate Cost Aspect

The interface must be able to expose important internal events to aspects. The **expose** declaration in Figure 4 exposes any joinpoints matched by its pointcut, not just those that are external to Figure. This allows TranslateCost to apply advice to the Point.translate calls inside Figure. The distinction between **advertise** and **expose** is that **expose** affects all joinpoints, whereas **advertise** only affects joinpoints for which the shadow occurs outside the module. In fact, **advertise** is implemented by translation into **expose**, which we explain in section 3.5.

The alternative solution to this problem would have been to modify the pointcut in ReplayAspect to

```
call(* Figure.translate(int, int)) &&
    !cflowbelow(call(* Figure.translate(int, int)))
```

once the modifications to Figure had been made. This, however, assumes that the programmer who made the modifications would have access to all the client aspects of Figure, which is not always the case (e.g. if Figure were distributed as part of a library).

It is conceivable for a software project to be partitioned into two groups of code: one that allows the use of aspects, and another that does not allow the use of any aspects. As an example, take a project that has common core components stored under the package core, and a set of applications that use those components under the package apps. Further suppose that because the core components are used by many different applications, a policy that aspects may not advise any of the core components is instituted over the project, to prevent aspects from one application from inadvertently affecting the behavior of a core component used by another application. This policy is expressible in open modules, by specifying a module that contains core, and not advertising or exposing any joinpoints as in the module Core in Figure 6.

```
module Core {
    class core ..*;
}

module AppFTP {
    class apps.ftp ..*;
    expose to apps.ftp..* : call(* *(..));
}
```

Figure 6: Core and Application Example

Similarly, it would be desirable to ensure that the classes of an application are advised only by the aspects that belong to that application. For this, we define a different form of the **expose** and **advertise** declarations that includes a **to** clause, which specifies the aspects to which the pointcut is exposed. Module AppFTP in Figure 6 shows a module that exposes all calls to functions in the classes under apps.ftp, but only to aspects that are also under apps.ftp.

The interfaces defined above encapsulate the implementation of the classes by defining a set of pointcuts which are visible to aspects, and hiding the rest. This would require a change to the matching behaviour of AspectJ, to ensure that any advice that would normally be matched against all joinpoints in the classes now respects the joinpoints exposed by the pointcuts used in the interface.

3. LANGUAGE DESCRIPTION

Now that we have conveyed an informal understanding of modules through a few examples in the preceding section, we start again from the beginning, and develop the definition of modules in a rigorous fashion. The structure of this development is as follows. First we review the official semantics of AspectJ. Next we introduce a simple hiding operator, and a new pointcut. This machinery then allows us to define the meaning of open modules, as well as the composition of such modules, via a normal form. The simple hiding construct is thus intended as an intermediate form, a device for understanding, but not for direct use by an AspectJ programmer.

3.1 AspectJ joinpoint matching

In AspectJ, joinpoint matching is defined as an operation at runtime, not (as is commonly but mistakenly believed) as a program transformation. It is important, therefore, to define the effect of open modules with respect to that runtime definition. We briefly review the semantics of joinpoint matching before discussing how it should be changed to accommodate appropriate forms of information hiding.

A *joinpoint* is an event of interest at runtime, such as a method call: a method call joinpoint starts upon each method invocation, and it completes when the call returns, be it normally or via an exception. Joinpoints are always properly nested: two joinpoints are either disjoint or one is included in the other. One can thus think of a program execution trace as a sequence of joinpoint enter and exit events, where enter and exits are properly bracketed. Besides method call, AspectJ has 10 other different kinds of joinpoint, for setting and getting a field, for executing a method body, and so on.

Advice in AspectJ consists of a kind (**before**, **after** or **around**), a pointcut and a piece of code. Both **before** and **around** advice are matched against joinpoint enter events, while **after** advice is matched against joinpoint exit events. The matching consists of taking the pointcut, and checking whether it matches one of the *signatures* of a joinpoint. Upon a successful match, the virtual AspectJ machine executes the corresponding advice: in the case of **before** and **after** advice, the extra code is *inserted*, but in the case of **around** it *replaces* the computation of the original joinpoint.

The set of signatures of a joinpoint is defined separately for each type of joinpoint. As an example (taken from [14]), consider the code fragment

```
T t = new T();
t.m("hello");
```

as well as the type hierarchy displayed in Figure 7. A call joinpoint occurs when we execute the statement t.m("hello"). This joinpoint has four signatures:

```
R2 T.m(String)
R2 S.m(String)
R1 P.m(String)
R1 Q.m(String)
```

```
interface Q {
 R1 m(String s);
}
class P implements Q {
 R1 m(String s) {...}
}
class S extends P {
 R2 m(String s) {...}
}
class T extends S {}
```

Figure 7: Example type hierarchy.

More generally, for each super type A of T, if m(param_types) is defined for that super type, then R(A) A.m(param_types) is a signature of the call join point, where R(A) is the return type of m(param_types) as defined in A, or as inherited by A if A itself does not provide a definition of m(param_types). Note that every signature has a unique declaring type, but the same identifier and parameter types as the other signatures.

The details of the matching process are spelled out in the AspectJ developer's notebook [14].

3.2 Visibility and hiding

If we wish to hide implementation detail of classes by hiding joinpoints, we have to modify the matching process. The simplest solution, from an implementation point of view, is to annotate each class with a *visibility pointcut* that exposes the joinpoints which aspects are allowed to observe. We may also annotate a class with its *friends*, that is, an ordered list of aspects that are permitted to intercept any joinpoint originating from the class.

For every joinpoint signature, we define the *owning class* to be the declaring class (which is defined to be part of the signature), except for handler joinpoints, where we define it to be the exception type.

Now joinpoint matching is modified as shown in the pseudo code of Figure 8. Consider a signature whose owning class is C. If we are processing a piece of advice with pointcut *pc* that is declared in a friend of C, we match as normal. However, if this is advice that is not from a friend, we add a new conjunct to the pointcut *pc*, namely the visibility of the owning class C. This has the effect of hiding any joinpoints that do not satisfy the visibility pointcut.

```
for each signature sig of jp {
    for each piece of advice pc {
        Aspect a = pc.declaringAspect;
        Class c = sig.owningClass;
        if (c.friends.contains(a))
            sig.match(pc);
        else {
            Pointcut npc = c.vis && pc;
            sig.match(npc);
        }
    }
}
```

Figure 8: Pointcut matching with hiding

The order of the list of friendly aspects is important: we take it as implicitly defining a series of precedences, where the last aspect has highest precedence. To illustrate, consider the example class in Figure 9. Now consider a call to f (from somewhere outside the Example class). It will be advised by Aspect1 because Aspect1 is a friend, and by Aspect2 because calls to f are declared to be visible. The internal call to g (from within f) will however only be advised by Aspect1.

A class is completely unaffected by advice if we give it a **false** visibility pointcut, and an empty list of friends. The default (current AspectJ) behaviour is that the visibility pointcut is **true** and the list of friends is empty.

Any pointcut can be exposed, with the exception of pointcuts that have variables. Pointcuts with variables may be implemented using *local pointcut variables* [5], but these are not yet part of standard AspectJ.

3.3 Selective hiding

As it stands, join point exposure does not discriminate between different aspects. This seems undesirable: allowing a tracing aspect to observe certain behaviour is less controversial than an aspect that overrides existing implementations via **around**.

```
aspect Aspect1 {
  before() : call(* f (..))  {}
  before() : call(* g (..))  {}
}

aspect Aspect2 {
  before() : call(* f (..))  {}
  before() : call(* g (..))  {}
}

class Example expose : call (* f(..))
              friend Aspect1
{
   void f() {
     g();
   }

   void g() {}
}
```

Figure 9: Example of hiding

For that reason, we introduce a slightly restricted form of exposure, namely

expose to abc.lib.tracing..* : **call**(* f (..))

Now we only allow aspects in a subpackage of abc.lib.tracing to intercept calls to f. More generally, any class name pattern expression may be indicated as the target of an **expose** clause. When no target is specified with the **to** syntax, we assume the pattern is *, so the exposure is universal as it was above.

We find it convenient to reduce this new feature to the ones we already have, by introducing a new form of pointcut. The pointcut

thisAspect(<classname−pattern−expression>)

acts the same as the existing **this** pointcut of AspectJ, but it matches on the aspect instance rather than on the instance of the advised object. That is to say, **thisAspect**(A) will match if the current advice is declared in an aspect whose type is a subtype of A.

Given this new pointcut, the form

expose to <pat> : <pc>

is equivalent to

expose : <pc> && **thisAspect**(<pat>)

As we shall see below, this transformation is crucial in obtaining a clean semantics via a normal form for composition of open modules.

3.4 Open modules

The annotation of individual classes with a visibility pointcut and a list of friendly aspects is *not* intended as a language construct for use by application programmers: it is intended only as an intermediate form. As set out in Section 2, our proposal is to use a generalised form of Jonathan Aldrich's *Open Modules* instead. We now show how these can be formally defined via the intermediate form discussed above.

The syntax is illustrated in Figure 10. An open module is introduced by the keyword **module**. The classes that are contained in the module are specified via one or more **class** declarations: these may be references to specific classes, or more generally they can be a class name pattern expression. A list of friendly aspects is introduced by the keyword **friend**. The aspects must be individually named, and the order in which they appear is important, again because we wish to pin down the order of precedence. Friendly aspects are not implicitly included in the classes: to control the exposure of joinpoints owned by an aspect, that aspect must be explicitly listed as a class.

```
module Example {
 class C1;
 friend A1, A2;
 class pack..Pat*
 advertise : call(* f (..));
 friend A3;
 expose : call(* g (..));
}
```

Figure 10: A sample module.

A module can also contain a number of *advertised* pointcuts. Joinpoint signatures whose owner is external to the module can be matched by these. Typically these are calls to public functions which can be logged. For example, consider a class that provides the factorial function. It could be implemented via a simple recursion, or by a loop. By advertising the pointcut **call**(* factorial (..)) , external aspects cannot distinguish between a recursive and a non-recursive implementation, because internal calls are not advisable, whereas all calls that happen outside the module can be intercepted.

Finally, a module may chose to reveal some of its internal details, by *exposing* a pointcut. Joinpoint signatures can always match an exposed pointcut, whether the advice originates from a friendly aspect or not. A joinpoint whose owning class is in the module, which matches both an advertised pointcut (which on its own would hide that joinpoint) and an exposed pointcut, is still exposed.

If a module definition does not contain any **advertise** or **expose** statements then it does not expose any joinpoints to external aspects. This is equivalent to having the visibility pointcut

expose: **false**

Each class and each aspect may occur in only one module, in the same way that in pure Java they must belong to only one package.

3.5 Normal form

There is a normal form for modules that will prove to be very convenient, both to pin down the semantics of modules and in the implementation.

First, note that it is not necessary to have more than one class declaration: we can always combine multiple class declarations by writing a class name pattern expression that consists of multiple disjuncts. Similarly, we can collect all aspects into a single declaration, where all the aspects are listed in the order they occur in the module. We can turn each advertised pointcut into an exposed pointcut by adding the requirement that they do not match within any of the classes that are contained in the module. Finally, all exposed pointcuts can be combined with ||. The result is a module that consists of one class declaration, one list of aspects, and one exposed pointcut. Figure 11 illustrates this process on the example of Figure 10.

```
module Example {
 class C1 || pack..Pat*;
 friend A1, A2, A3;
 expose : ((call(* f (..))
            && !within(C1 || pack..Pat*)))
        || (call(* g (..));
}
```

Figure 11: Normalised sample module.

The reader may now wonder why we allowed the more liberal syntax, since a single **class**, **friend** and **expose** declaration suffice. The answer is one of notational convenience: writing long formulae will make substantial module specifications hard to read. It is also worthwhile to separate the advertised and exposed pointcuts, since conceptually their purpose is different. Finally, separating the **friend** lists is actually useful when they are interspersed with module compositions, discussed further below.

It should now be clear that such a normalised module may be transformed into our earlier hiding construct. Each class that matches the class name pattern expression is annotated with the list of friendly aspects and the exposed pointcut as its visibility pointcut.

3.6 Module composition

It would not be very satisfactory if modules were flat entities, and could not be combined to form larger systems. We therefore define two notions of module composition.

The first and most obvious type of composition is one that constrains the visibility of module members further. This is indicated by the keyword **constrain**, and an example is displayed in Figure 12. We simply further restrict visibility, as shown in Figure 13: the descendant (included) module's visibility is constrained to that of the parent (including) module. Note that there is no change in the list of friendly aspects of either the parent or the child module. It is however the case that the friends of the parent enjoy the original visibility in the child: the deciding factor for introducing **expose to** <pat> : <pc> and the **thisAspect** pointcut was to give a clean meaning to **constrain**. Constrained composition gives strong guarantees for modular reasoning: any assumptions we made about the absence of interference in the child module remain true after applying the composition.

```
module M1 {
  class C1, C2;
  friend A1, A2;
  expose : C1.pointcut1();
}
module M2 {
  class C3;
  friend A3;
  constrain M1;
  friend A4;
  expose : A4.pointcut2();
}
```

Figure 12: Constrained module composition

The second type of module composition is dual to the first: here the parent gains unrestricted access to the children. That is, the parent's friends can advise joinpoints that arise in the children regardless of the visibility pointcuts. We therefore introduce such keywords by the keyword **open**.

```
module M1 {
  class C1, C2;
  friend A1, A2;
  expose : (C1.pointcut1() && A4.pointcut2());
           || (C1.pointcut1() && thisAspect(A3 || A4));
}
module M2 {
  class C3;
  friend A3,A4;
  expose : A4.pointcut2();
}
```

Figure 13: Normalised constrained composition

Again we define the semantics by a normalisation process, as illustrated in Figures 14 and 15. Note that we add the parent's visibility pointcut as a *disjunct* to the child's; this contrasts with constrained composition, where it becomes a conjunct. Furthermore, the child has gained the parent's friends as friends also, and the order follows that in the parent: A3, the included friends A1 and A2, and A4.

```
module M1 {
  class C1, C2;
  friend A1, A2;
  expose : C1.pointcut1();
}
module M2 {
  class C3;
  friend A3;
  open M1;
  friend A4;
  expose : A4.pointcut2();
}
```

Figure 14: Open module composition

```
module M1 {
  class C1, C2;
  friend A3,A1,A2,A4;
  expose : C1.pointcut1() || A4.pointcut2();
}
module M2 {
  class C3;
  friend A3,A4;
  expose : A4.pointcut2();
}
```

Figure 15: Normalised open composition

It should be noted that while a parent's friend aspects are not added to the child when using constrained composition, the precedence order of the aspects would be the same as if the child was included using open composition. Thus in Figure 13, A3 comes before A1 while A4 comes after A2, even though A3 and A4 do not appear as friend aspects of M1.

The module hierarchy is required to be free of cycles. Modules may only be included in at most one other module: this prevents other modules from indiscriminately exposing additional pointcuts using open composition. As the module hierarchy is envisioned to closely follow the package hierarchy, this limitation should not produce too many problems in practice.

The purpose of normalisation is to reduce all modules to the simple hiding construct of Section 2.1. It is possible that intermediate modules along the way to a normal form violate the constraint that each aspect occurs in only one

module, because we distribute the parent's friends among multiple children. Fortunately, one can prove that a unique total order is defined by the original modules, and all the resulting lists of friendly aspects are compatible with that unique total order. This will be discussed in detail in Section 5.

Normalisation does preserve the property that each class is part of only one module: this is very important, for otherwise the resulting visibility pointcut of that class would not be well defined. It is for this reason that we have chosen *not* to make friendly aspects implicit classes declared in a module.

3.7 Private visibility modifier

It may sometimes be desirable to expose a visibility pointcut that applies only to the immediate members of a module, without affecting the visibility of its included modules. This allows a programmer to expose arbitrary pointcuts that only apply to the immediate members of a module, without worrying about its effects on any included modules. To do this, an **advertise** or **expose** declaration is modified by the keyword **private**. Private visibility pointcuts only apply to the immediate class members and friend aspects of the module. Figure 16 shows an example that uses private visibility declarations, and Figure 17 shows its normal form.

```
module M1 {
    class C1, C2;
    friend A1;
    open M2;
    expose: A1.pointcut1();
    private expose : A1.pointcut2();
}

module M2 {
    class C3, C4;
    friend A2;
    expose : A2.pointcut3();
}
```

Figure 16: Private visibility modifier

```
module M1 {
    class C1, C2;
    friend A1;
    expose: A1.pointcut1() || A1.pointcut2();
}
module M2 {
    class C3, C4;
    friend A1, A2;
    expose : A1.pointcut1() || A2.pointcut3();
}
```

Figure 17: Normalised private visibility

3.8 Root module modifier

In certain circumstances, it may also be necessary that a module be unavailable for composition, to ensure that it defines the final visibility pointcuts for all the classes it contains. As an example, one may wish to define a *master* module that includes all other modules and is used to enforce a constraint on all those modules. To ensure that the constraints are not overriden by a new module by including the master module, the *master* module must be unavailable for inclusion in any other module. Our design for open modules introduces the **root** visibility modifier to specify that a module cannot be included by any other module in the compilation. Figure 18 shows an example of a root module that does not allow any advice to apply to constructors in the modules M1 to M3.

```
root module MasterModule {
    constrain M1, M2, M3;
    expose: !call(* new.(..));
}
```

Figure 18: Root module modifier

4. EXTENDED EXAMPLE

We now provide an example of the usage of open modules on a substantial program written in AspectJ. *Ants* is an implementation of a simulator for the problem given in the 2004 ICFP Programming Contest [1]. The problem involves two teams of ants on a hexagonal grid containing food and obstacles. Each can perform a specific set of actions; among these actions is the ability to test their immediate vicinity for certain conditions. The problem specifies rules for movement and combat between ants, which occurs when ants from opposite teams happen to be on adjacent hexes. The simulation proceeds in discrete rounds, and an ant can perform at most one action per round. The sequence of actions of an ant in a team is defined by a finite state machine, and the goal of each team is to collect more food that its opponent.

4.1 System Description

The simulator loads a world specification, and the respective state machines of each of the ant teams. It then simulates the movement of the ants following the rules specified in the problem, displaying the results on a graphical user interface.

Figure 19 shows an abbreviated UML diagram of the Ants application. Aspects are distinguished from ordinary classes by having a * before their name in the class specification. The dependency lines in the diagram also include advice application caused by aspects.

The Ants application is composed of seven major packages: automaton, command, model, parser, viewer, debug, and profile. The first four packages form the core of the simulation. The viewer package contains the GUI implementation. The debug package contain aspects that verify certain conditions on the whole of the program, while the aspects in profile check for any memory allocations that occur in the simulation's inner loop.

The automaton package contains the representation of the state machine used by the ant teams. The command package contains classes that represent the various actions that an ant can perform. It also contains an aspect Comment that adds the ability to parse comments in the ant state machine specification file. The model package contains classes that represent the different entities that form part of the simulation. It also contains two aspects, Combat and Resting that implement combat between ants and resting behaviour respectively. The parser package contains the parser for the ant specification files.

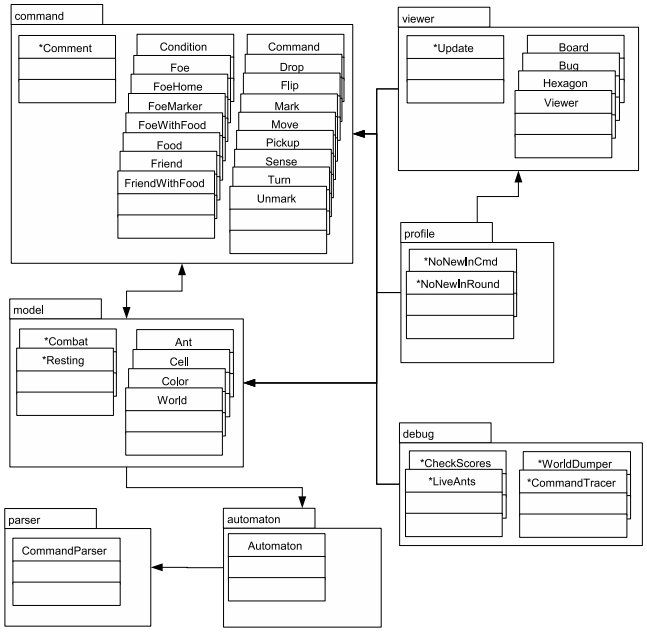

Figure 19: Ants

The viewer package contains the GUI implementation. This includes the Update aspect, which intercepts any events which should trigger a screen update.

The debug package contains aspects that verify certain conditions over the entire application, namely CheckScores, CommandTracer, WorldDumper and LiveAnts. CheckScores verifies the score kept by the World class by checking it against the score computed by directly counting number of food hexes in the map. CommandTracer intercepts every ant action and displays it on standard output. WorldDumper is triggered at the end of each round, when it scans the state of the hex grid and dumps it to a file. LiveAnts performs a sanity check on the set of ants in the simulator's list. It verifies that the list contains all of the ants that are still alive, and that it contains no dead ants.

The profile package contains the aspects NoNewInRound and NoNewInCmd. These aspects check allocations in inner loop of the simulation, as these may cause performance problems. NoNewInCmd checks for any allocations in the execution of an ant action, while NoNewInRound checks for any allocations that may occur while a round is being performed.

4.2 Module Specification

We now show how open modules can be used to specify the class-aspect interfaces in the application. Figure 20 shows a set of module declarations that may be used for the Ants application.

The modules closely follow the package structure of the application, with some modules containing more than one package. Aspects that belong to a package are declared to be a friend of the module that contains the package.

We believe that **expose** should be used with caution, as exposing joinpoints internal to a module allows external aspects to weave **around** advice, making it difficult to reason about the code internal to the module. We use the rule of thumb that if a joinpoint is to be made visible, the most preferred option is to use **advertise**, followed by **expose to**,

```
module Model {
    class model.*;
    class automaton.*;
    friend model.Combat, model.Resting;
    advertise : call(* model.World.round());
    expose : call(* model.Ant.kill());
}
module Command {
    class command.*;
    class parser.*;
    friend command.Comment;
    advertise : call(* command.Command.step(..));
}
module DebugAndProfile {
    class profile.*;
    class debug.*;
    friend profile.NoNewInCmd, profile.NoNewInRound;
    friend debug.WorldDumper, debug.LiveAnts,
        debug.CommandTracer, debug.CheckScores;
    open Model, Command;
}
module AntSystem {
    class viewer.*;
    friend viewer.Update;
    constrain DebugAndProfile;
    private expose to profile.*: call(*.new(..));
}
module JavaLang {
    class java.lang.*;
    advertise : !call(java.lang.StringBuffer.new(..));
}
```

Figure 20: Ants Module Specification

and if there is no other choice, by **expose**.

The Model module contains the packages that represent the entities in the simulation, namely model and automaton. It advertises external calls to World.round, which performs a single round in the simulation. It also exposes the internal event Ant.kill. It was necessary to use **expose** on the calls to Ant.kill as most of these calls occur inside classes in Module. Using **advertise** would have hidden too many of these events.

An alternative solution would have been to place Ant and its related aspects into a separate module, perhaps even a separate package, and then advertise the calls to Ant.kill from that module. This illustrates how the use of open modules may uncover opportunities for refactoring which would have otherwise gone unnoticed.

The Command module contains the packages command and parser. It advertises the external calls to Command.step, which executes the action specified by the command. The package parser was also included in this module as the purpose of its only member, CommandParser, is to parse ant specification files and generate the corresponding Command objects.

The DebugAndProfile module shows how open composition can be used to integrate intrusive debugging and profiling aspects into an AspectJ program. DebugAndProfile opens up the modules Model and Command, and has the aspects in debug and profile as friends.

The aspects in debug and profile apply advice to joinpoints that span multiple packages, thus it would have been messy if they were declared friends of each of those modules, or if those modules had declared a visibility pointcut to allow access from the aspects. The solution is to declare the aspects as friends of a module, and then open up the mod-

ules to which the aspects apply. This gives the debugging and profiling aspects unrestricted access to all the members of the included modules.

It should be emphasised that opening up a module using open composition places the responsibility of modularity on the owner of the including module. As open composition allows aspects unrestricted access to a module and may also expand its visibility, the guarantees about modularity made by the included module is effectively overridden by the including module. Thus the owner of the including module must be prepared to deal with any problems caused by a change in implementation of any of the members of the included module. These effects may be minimised by not exposing additional signatures when using open composition. This way, only the friend aspects of the including module are vulnerable to changes in the included module.

The module AntSystem shows how to restrict the visibility of the entire application. It contains the package viewer, and has the aspect Update as a friend. It uses constrained composition to include DebugAndProfile, and does not have a non-private signature, effectively conjoining **false** to the exposed pointcuts of DebugAndProfile and causing all external aspects to fail in matching joinpoints that belong to the application.

The module AntSystem also exposes a private visibility pointcut to the profiling aspects. As it is private, it has no effect on DebugAndProfile and its children, and only applies to viewer.∗ and viewer.Update. This allows the aspects NoNewInCmd and NoNewInRound to inspect constructor calls in viewer.

Finally, the module JavaLang hides calls to the constructor of java.lang.StringBuffer. This prevents the profiling aspects from generating false positives caused by StringBuffer allocations due to string literals.

Although this example uses pointcut primitives to define visibility pointcuts in the interests of clarity, it is recommended that named pointcuts be used when defining **advertise** or **expose** declarations, as in Figure 21. This encourages external aspects to use the named pointcut, thus insulating them from any changes to the visibility pointcut [9].

```
module ModuleA {
    expose : AspectA.fieldGets();
    class classes .∗;
}
aspect AspectA {
    pointcut fieldGets() : get(∗ ∗.∗) && within(classes.∗);
}
```

Figure 21: Using named pointcuts

4.3 Class visibility

To illustrate the effect of normalisation on the visibility pointcuts and the friend aspects of a class, we pick the class model.Ant and derive its visibility. Model.Ant is a member of the Model module, which has the visibility specification

```
    advertise : call(∗ model.World.round());
    expose : call(∗ model.Ant.kill ());
```

and the friend aspects model.Combat and model.Resting. Hence the initial visibility specification for model.Ant is

```
class Ant advertise : call(∗ model.World.round())
        expose: call(∗ model.Ant.kill())
        friend model.Combat, model.Resting
```

Note that

```
    advertise : call(∗ model.World.round());
```

in the module Model is equivalent to

```
    expose : call(∗ model.World.round()) &&
        !within(model.∗ || automaton.∗);
```

Thus the annotated Ant class is equivalent to

```
class Ant expose :
        (call(∗ model.World.round()) &&
            !within(model.∗ || automaton.∗)
        || call(∗ model.Ant.kill ())
        friend model.Combat, model.Resting
```

The module Model is included in DebugAndProfile using open composition. Since DebugAndProfile does not expose any pointcuts, its only effect on model.Ants is to add the debugging and profiling aspects to its list of friend aspects. Thus the annotated Ant class now becomes

```
class Ant expose :
        (call(∗ model.World.round()) &&
            !within(model.∗ || automaton.∗))
        || call(∗ model.Ant.kill ())
        friend profile .NoNewInCmd, profile.NoNewInRound,
            debug.WorldDumper, debug.LiveAnts,
            debug.CommandTracer, debug.CheckScores,
            model.Combat, model.Resting
```

Finally, DebugAndProfile is included in AntSystem using constrained composition. We note that constrained composition conjoins *non-private* pointcuts with the pointcuts of the included module, and exposes the existing visibility pointcuts to the friend aspects of the including module. As AntSystem contains no non-private visibility pointcuts, it conjoins **false** to the existing visibility pointcut, thus making the annotated Ant class

```
class Ant expose :
        (false &&
            ((call(∗ model.World.round()) &&
                !within(model.∗ || automaton.∗))
            || call(∗ model.Ant.kill ()))
        )
        ||
        (thisAspect(viewer.Update) &&
            (call(∗ model.World.round()) &&
                !within(model.∗ || automaton.∗))
            || call(∗ model.Ant.kill ())
        )
        friend profile .NoNewInCmd, profile.NoNewInRound,
            debug.WorldDumper, debug.LiveAnts,
            debug.CommandTracer, debug.CheckScores,
            model.Combat, model.Resting
```

which simplifies to

```
class Ant expose :
        (thisAspect(viewer.Update) &&
            (call(∗ model.World.round()) &&
                !within(model.∗ || automaton.∗))
            || call(∗ model.Ant.kill ())
        )
        friend profile .NoNewInCmd, profile.NoNewInRound,
            debug.WorldDumper, debug.LiveAnts,
            debug.CommandTracer, debug.CheckScores,
            model.Combat, model.Resting
```

Note that the private visibility pointcut of AntSystem has no effect on model.Ant.

The derivation of the annotated class highlights several decisions made in the design of open modules for AspectJ. The derivation is a straightforward process of following the chain of compositions starting from the module in which the class is an immediate member. As a class may only occur in one module and a module may be included in at most one module, a class' visibility annotation is completely determined by a simple path starting from the module in which it is an immediate member and ending with the topmost ancestor of that module in the composition tree. This is similar to following the inheritance hierarchy of classes in Java when determining the behaviour of a derived class.

Had the design allowed a class to occur in multiple modules, or a module to be included in multiple modules, one would have had to follow all the paths starting from the class to determine its visibility. This would have potentially coupled unrelated modules merely because they contribute to the visibility pointcut of a common class, thus making the modules themselves less modular.

It should also be noted that the introduction of open modules into the Ants application did not require any change in the application's code. The module definitions are in a file that is separate from Java and AspectJ code. The module namespace is also separate from that of Java and AspectJ, thus classes and aspects may not refer to the modules themselves. While this prevents the implementation of potentially useful pointcuts such as **within**(Module), it does mean that open modules may be added to or removed from a compilation without requiring any changes to the code.

The current implementation of open modules in *abc* generates warnings when advice that normally applies to a joinpoint is prevented from doing so because it does not comply with the visibility pointcuts defined for that joinpoint. This provides a compile-time hint that a certain aspect may be too imprecisely defined.

5. PRECEDENCE PROPERTIES

Apart from specifying the interface between classes and their client aspects, open modules may also be used to specify the ordering of the aspects included in modules. Indeed, it can be shown that if the set of module definitions follow certain constraints, a total order can be imposed on all the aspects in a tree rooted at a particular module.

We abstract a module m to be a sequence composed of aspects a and other modules m' to represent the sequence of friend aspects and module compositions defined in a module.

Definition (Valid Open Module Set)

We say a set S_m of modules is a *valid open module set* if it satisfies the following properties:

1. An aspect is included in at most one module.
2. If a module m includes a module m', then m' must also be in the set.
3. A module can be included in at most one module, and there are no cyclical module inclusions.

A *top-level module* is a module that is not included in any other modules in a valid open module set.

We can now state our formal result and prove it.

Theorem

Given a valid open module set S_m, let S'_m be any valid subset of S_m that has a single top-level module (i.e. S'_m is a tree in the module hierarchy). Then we can define a unique total order on the precedence of the friend aspects of modules in S'_m.

Proof

The proof is by construction: We exhibit a procedure to obtain the total order of precedence from the root module m of S'_m. Recall that we think of a module as an ordered list of friend aspects and included modules.

Definition (Induced Aspect Order)

Let m be the only top-level module of a valid open module set S_m. The aspect order $aspectorder(m)$ induced by m is

$$\begin{aligned} aspectorder(\langle\rangle) &= \langle\rangle \\ aspectorder(\langle a\rangle \frown s) &= \langle a\rangle \frown aspectorder(s) \\ aspectorder(\langle m'\rangle \frown s) &= aspectorder(m') \frown aspectorder(s) \end{aligned}$$

In the above, we use $\frown$ to denote list concatenation. As an example, given three modules m_1, m_2, m_3

$$\begin{aligned} m_1 &= \langle a_1, m_2, a_2, m_3\rangle \\ m_2 &= \langle a_3, a_4\rangle \\ m_3 &= \langle a_5, a_6\rangle \end{aligned}$$

with m_1 being the top-level module, then

$$aspectorder(m_1) = \langle a_1, a_3, a_4, a_2, a_5, a_6\rangle$$

Claim: $aspectorder(m)$ defines the required total order on the module hierarchy tree S'_m rooted at m.

Since m is the only top level module of S'_m, every other module in S'_m must be a descendant of m, as cyclical inclusions are not allowed in S'_m.

As $aspectorder(m)$ is already a sequence, it only needs to be shown that it contains no duplicate elements, and that it contains all the aspects that are friends of modules in S_m. The first condition is proved by induction, using the constraints on modules of a valid open module set. The second is satisfied by observing that *aspectorder* is a traversal, hence $aspectorder(m)$ must contain the aspects of m and its descendants. □

Note that the theorem only allows us to totally order open module sets that have only one top level module. In general, an open module set may contain several disjoint trees. To define a total order on these trees, one must include all the top-level modules in a single module.

6. IMPLEMENTATION

Open modules for AspectJ were implemented as an extension in the AspectBench compiler (*abc*) [2, 4–6]. *abc* is built on the Polyglot extensible compiler framework [12] and the Soot optimisation framework [15].

Adding an extension to *abc* involves the extension of the parser to process any new syntax introduced by the extension, the addition of abstract syntax tree (AST) nodes to represent the new constructs and the modification of the matching and weaving behaviour of the compiler. *abc* is designed to be extensible, and allows these modifications through Polyglot's extensible parser generator, and by using factories and interfaces to allow for changes to the behaviour of the existing AST, matching and weaving classes.

The open module extension required an extension of the AspectJ syntax to include module definitions, an internal

representation of the modules and a modification to the matcher to implement the effect of visibility pointcuts. The syntax extensions and AST nodes were implemented by the prescribed method of extending the parser specification and subclassing the existing Polyglot AST nodes.

The implementation of visibility pointcuts required an extension of *abc*'s matcher. *abc* provides a way to add new pointcuts as well as to modify the matching behaviour of an existing pointcut, but did not allow for an extension of the matcher itself. Visibility pointcuts add a new condition to the matching process: a piece of advice must satisfy the visibility criteria of a class before it is woven. As such it is best implemented as an extension to *abc*'s matcher. This required a refactoring of the existing *abc* matcher to allow for an extension to override the matcher's behaviour, as well as a generalisation of the way data is passed to the matcher.

Except for the matcher, *abc*'s extension mechanisms worked remarkably well during the implementation of the open module extension. The changes to the matcher have remedied an oversight in *abc*'s extensibility, making it easier to implement extensions similar to open modules.

7. RELATED WORK

Aldrich. Open modules were proposed by Aldrich [3] as a way to allow for modular reasoning while using advice. His design uses a small functional language called *TinyAspect*. Each module contains a set of functions and advice, as well as a signature, which determines which points in the code are available to advice outside the module. This signature forms a contract between a module and external advice: external advice must comply with the signature, and in return any change in the implementation of the module must maintain the semantics of the signature, thus insulating external advice from changes in the module. It also allows for module composition (our **constrain** construct), and defines the effect that composition has on the signatures. The proposal defines a set of equivalence rules which may be used to determine if a change in the implementation of a module violates the contract implied by the module's signature.

Aldrich's paper was the starting point of the investigations reported here. Relative to his work, our main contribution is to extend the design to full AspectJ: this involved dealing with non-call joinpoints, the result about consistent aspect precedence, and the reduction to the simple **expose** annotation on classes. Furthermore we introduced the notion of *opening* a module, which again can be reduced to the normal form.

One might argue that the opening construct defined in this paper destroys Aldrich's formal result about modular reasoning. Our response is that modular reasoning *is* possible, provided one knows the whole module specification, as then it is possible to calculate exactly what joinpoints are exposed by each class. We furthermore feel that such a minor complication is amply justified by the pragmatic advantages of module opening: in aspect-oriented programming, it is essential that the programmer can assert responsibility for joinpoints that occur in parts of the system not originally written by her.

Gudmundson and Kiczales. Gudmundson and Kiczales outline the idea of *pointcut interfaces* [9]. This involves annotating a class with named pointcuts, which expose joinpoints of interest in that class. This named pointcut is then used by aspects when defining advice that apply to the class. They also provided a way to define such pointcuts for packages by defining them in a special Pointcuts class, and to the whole program using a similar method. They do not, however, provide a way to extend or constrain the interface without directly modifying the interface specifications in the class annotations.

Our intermediate representation (normal form) for open modules is clearly akin to the annotations of Gudmundson and Kiczales: what we have added is the notion of specifying the annotations via module specifications. Indeed, we believe it is undesirable that programmers write such annotations directly: it violates the principle of *obliviousness* (that classes are unaware of the aspects that may advise their code), and it is inflexible: often different module specifications may apply to the same class in different circumstances. It is thus important that module specifications are separated from the classes to which they apply.

Mezini and Kiczales. Mezini and Kiczales propose aspect aware interfaces [11] to define how aspects may modify classes. These interfaces annotate method declarations with the aspect and the type of advice that may apply to them. They, however, only consider execution joinpoints. Referring to the work of Aldrich on modular reasoning, they argue modular reasoning can be applied as soon as the aspect interfaces are known; and the aspect interfaces can be automatically calculated through a global analysis of the whole system.

This use of an 'initial global analysis' is similar to our claim that modular reasoning about AspectJ is facilitated by considering the whole module specification, together with the code in question. The difference, however, is that the module specification is typically very small compared to the system as a whole, and in top-down design, the module specification is available before the system is complete.

Clifton and Leavens. Clifton and Leavens also address the problem of modular reasoning via annotations that state what aspect may affect a given part of the code [7]. A distinctive feature of their proposal is that they distinguish between *spectators* (aspects that merely observe) and *assistants* (aspects that add new functionality). Assistants can only apply where explicitly allowed via an annotation. To reduce the burden of writing annotations, *aspect maps* are introduced, which allow the specification of multiple annotations in one place.

Aspect maps are similar to open modules, but far more restricted, and they lack the combining forms of *constrain* and *open*. At present it is not possible to distinguish between spectators and assistants in our design. Ideally, one might write for instance

expose to pure foo.bar..* : pointcut;

That is, the given set of joinpoints is exposed only to *pure* aspects in a subpackage of *foo.bar*.

We believe that this would be a very valuable feature, but it is independent of the proposal for open modules. Instead, we plan to introduce the new optional modifier *pure* on aspects, and provide compiler support for checking that the advice in the aspect is indeed pure. In fact, in an initial investigation, we have found it necessary to implement a

more general feature, where an aspect is annotated with the construct

pure on ⟨classname-pattern⟩ **except** ⟨pointcut⟩ :
⟨aspect-declaration⟩

That is, we assert that an aspect does not introduce side-effects at any joinpoints which have a signature owned by classes that match the pattern, except at those that match the given pointcut. This makes it possible, for example, to state that printing on *System.err* is not considered a purity violation. Details of this design, and an initial implementation using the Soot analysis framework, can be found in [13].

Dantas and Walker. Dantas and Walker have also proposed a notion of harmless advice [8], which is advice that does not modify the execution of the joinpoint it intercepts. Exposing a pointcut in a module renders it vulnerable to around advice, which may choose to bypass the execution of the joinpoint it intercepts. It may be desirable allow only harmless advice to apply to certain points in a module.

Again, we believe such classification of advice effects to be an important issue in aspect-oriented programming, but from the language design point of view, it should be treated separately from modules.

8. CONCLUSIONS

We have presented a detailed design for the addition of open modules to AspectJ. Open modules provide a convenient notation for summarising the interaction between aspects and other parts of the code. Starting with the original proposal of Aldrich, we enhanced it to encompass the whole of AspectJ.

Despite the fact that we deal with an industrial-strength language, the definition of our new feature is particularly simple thanks to an intermediate representation, which provides a normal form for hiding constructs in AspectJ. We validated our design via many concrete examples, and demonstrated its use on a more substantial one that involves eight separate aspects. The claim that our proposal constitutes a seamless extension of the existing AspectJ language was underpinned by a formal result which guarantees that the use of open modules cannot lead to precedence conflicts.

We have implemented open modules in *abc*, the extensible research compiler for AspectJ. Open modules are part of *abc*'s 2.0 release, so others can experiment with their use. Our own plan for future development is to combine open modules with an effect analysis for advice [13].

9. REFERENCES

[1] ICFP 2004. 7th annual ICFP programming contest. `http://www.cis.upenn.edu/~plclub/contest/`.

[2] abc. The AspectBench Compiler. Home page with downloads, FAQ, documentation, support mailing lists, and bug database. `http://aspectbench.org`.

[3] Jonathan Aldrich. Open Modules: Modular Reasoning about Advice. In *Proceedings of the European Conference on Object-Oriented Programming*, volume 3586 of *LNCS*, pages 144–168. Springer, 2005.

[4] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. The AspectBench Compiler for AspectJ. In *Generative Programming and Component Engineering*, volume 3676 of *LNCS*, pages 10–16. Springer, 2005.

[5] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. *abc*: An extensible AspectJ compiler. In *Aspect-Oriented Software Development (AOSD)*, pages 87–98. ACM Press, 2005.

[6] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Optimising AspectJ. In *Programming Language Design and Implementation (PLDI)*, pages 117–128. ACM Press, 2005.

[7] Curtis Clifton and Gary T. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning. In Gary Leavens and Ron Cytron, editors, *FOAL 2002*, Technical Report 02-06, Computer Science, Iowa State University, pages 33–44, 2002.

[8] Daniel S. Dantas and David Walker. Harmless advice. In *12th International Workshop on Foundations of Object-Oriented Languages*, 2005. Available from `http://homepages.inf.ed.ac.uk/wadler/fool/program/6.html`.

[9] Stephan Gudmundson and Gregor Kiczales. Addressing Practical Software Development Issues in AspectJ with a Pointcut Interface. In *ECOOP 2001 Workshop on Advanced Separation of Concerns*, 2001.

[10] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In J. Lindskov Knudsen, editor, *European Conference on Object-oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer, 2001.

[11] Gregor Kiczales and Mira Mezini. Aspect-oriented programming and modular reasoning. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 49–58, New York, NY, USA, 2005. ACM Press.

[12] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In *12th International Conference on Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 138–152, 2003.

[13] Elcin Recebli. Pure aspects. MSc dissertation, University of Oxford. Available from `http://aspectbench.org`, 2005.

[14] The AspectJ Team. The AspectJ 5 Development Kit Developer's Notebook. `http://eclipse.org/aspectj/doc/next/adk15notebook/`, 2004.

[15] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[16] Mitchell Wand. Understanding aspects: extended abstract. In *8th ACM SIGPLAN International Conference on Functional Programming*, pages 299–300, 2003.