An Implementation of Open Modules in AspectJ

Oxford University
Computing Laboratory

Neil Ongkingco
Keble College

Submitted in partial fulfillment of the requirements for the degree of
MSc in Computer Science

September 2, 2005

Abstract

Aspect-oriented programming languages provide *advice* to modify the behavior of programs. Some current languages, notably AspectJ, provide few limitations on advice, so new advice can change the behavior of existing programs without any restrictions. Changes to the base code can also affect the behavior of existing advice in an unexpected manner. Open modules is one proposal to solve the problems above. It is defined for a small functional aspect-oriented language, and has the benefit of a formal definition of its behavior. There is, however, no implementation for Open Modules in an existing aspect-oriented programming language.

AspectJ is an aspect-oriented programming language based on java. This dissertation presents an implementation of Open Modules in AspectJ, using the AspectBench compiler. The implementation also allows a module to be included in another module, and defines the effect of the inclusion on the properties of the modules.

Acknowledgements

I would like to express my gratitude to Professor Oege De Moor, who introduced me to AspectJ and Open Modules, as well as consistently providing guidance and insight during the whole of the project. I also thank Dr. Jonathan Aldrich for providing useful feedback on the proposed design for Open Modules in AspectJ, and the AspectBench team for their assistance during the implementation of Open Modules in the AspectBench compiler.

Contents

List of Tables

List of Figures

Listings

Chapter 1

Introduction

Aspect-oriented programming (AOP) is a relatively new programming methodology that was designed to enable the modularization of features that cross-cut boundaries of conventional abstraction models [4]. AOP allows the implementation of the feature to be written separately from the existing (base) code, and is then "woven" into the base code to produce the fully functional system. When applied to object-oriented programming, AOP addresses the issue of features that cross-cut the class hierarchy, such as invariant checking or logging. Implementations of such features using traditional methods are usually scattered across classes that are unrelated in the hierarchy.

Existing aspect-oriented programming languages, notably AspectJ, allow the modification of the base code in an unrestricted manner. This implies that the behavior of any existing code can be modified arbitrarily. Furthermore, as advice is woven at specific points in the base code, a change in the base code can lead to a change in the behavior of existing aspects that advise that code. This makes it difficult to alter the behavior of the program as potentially all advice affecting it must be examined before any modifications.

Open modules [2] is a proposal to restrict the ability of aspects to modify base code by specifying an interface containing the set of points in the code that can be modified by aspects. The interface enforces a contract between the code and its client aspects: that aspects will modify only the points specified in the interface, while the semantics of the points exposed in the interface will be maintained even after modifications to the base code.

1.1 Contributions

The dissertation made the following contributions:

- Extended the design of Open Modules to AspectJ.
- Introduced a novel notion of module inclusion.
- A proof that the design is consistent with AspectJ's rules of aspect precedence.
- Provided a full implementation of Open Modules in the AspectBench compiler.
- Identified scope for further improvements of the extensibility of the AspectBench compiler.

1.2 Thesis Organization

Chapter 2 contains a review of aspect-oriented programming, the modularity issues introduced by advice and a description of Open Modules and its semantics. Also contained is a short description of AspectJ and the AspectBench compiler.

Chapter 3 presents the design of Open Modules in AspectJ. The module construct and its effect on pointcut matching and aspect precedence is described.

Chapter 4 details the implementation in the AspectBench compiler and the issues involved in the integration of Open Modules in the said compiler.

Chapter 5 contains proofs that the implementation behaves similarly to Open Modules, and is consistent with the aspect precedence rules of AspectJ.

Finally, chapter 6 concludes with a summary and suggestion for future work.

Chapter 2

Background

2.1 Aspect-Oriented Programming

Aspect-oriented programming (AOP) was designed to encapsulate cross-cutting features that would normally be spread across a program [4]. One feature found in AOP languages is *advice*, which is woven at specific points of the base program to alter its behavior. This allows the implementation of a cross-cutting feature to be encapsulated by a single abstraction instead of being spread across the object hierarchy.

2.2 AspectJ

AspectJ [6, 10] is an aspect-oriented programming language based on Java. Some of the basic constructs used by AspectJ to provide aspect-oriented functionality are *joinpoints*, *pointcuts* and *advice*. *Joinpoints* are a representation of events in base code to which modifications can be applied. The set of locations in the code where a joinpoint may match is known as the joinpoint's *shadow*. *Pointcuts* specify a set of joinpoints. Pointcuts can specify any set of joinpoints in the entire program. *Advice* are code blocks that are applied to joinpoints specified by its associated pointcut. There are three basic forms of advice: *before*, *after* and *around*. *Before* advice is applied before the joinpoint, *after* advice after the joinpoint, and *around* advice is executed instead of the joinpoint. Pointcuts and advice are declared in an *aspect*, which serves as the modularization construct of the language. AspectJ also provides aspect inheritance, which behaves like class inheritance.

What follows is a short review of the features of AspectJ that are relevant to this thesis.

2.2.1 Basic Pointcuts and Advice

Listing 2.1 shows a simple aspect and the base code that it modifies. It contains an aspect named *BasicAspect*, which contains a piece of *before* advice (line 4) and an *after* advice (line 7). Both are applied to the pointcut *pc_foo*, which refers to calls to a method named *foo* without any parameters and with a void return value (line 2). The pointcut matches on the call to *foo* in line 18.

Listing 2.1: Basic pointcuts and advice

```
 1  aspect BasicAspect {
 2      pointcut pc_foo(): call(void foo()); //pointcut declaration
 3
 4      before(): pc_foo() { //Before advice
 5          System.out.println(”Before␣foo”);
 6      }
 7      after(): pc_foo() { //After advice
 8          System.out.println(”After␣foo”);
 9      }
10  }
11
12  class Base {
13      public void foo() {
14          System.out.println(”In␣foo”);
15      }
16      public static void main(String args[]) {
17          Base b = new Base();
18          b.foo();
19      }
20  }
```

The program, predictably, produces the following output:

```
Before foo
In foo
After foo
```

2.2.2 Around Advice and Formals

Around advice is executed instead of the matching joinpoint. The advice can then optionally invoke the joinpoint by using the *proceed* keyword. Listing 2.2 demonstrates the behavior of around advice.

Listing 2.2: Around advice

```
1  aspect AroundAspect {
2      pointcut pc_foo(int x) : call(double foo(int)) && args(x);
3
4      double around(int x): pc_foo(x) {
5          double ret = 0.0;
6          if (x != 0) {
7              ret = proceed(x);
8          } else {
9              ret = 0.0;
10         }
11         return ret;
12     }
13 }
14
15 class AroundBase {
16     public double foo(int x) {
17         return 1.0/x;
18     }
19
20     public static void main(String args[]) {
21         AroundBase x = new AroundBase();
22         x.foo(2); //returns .5
23         x.foo(0); //returns 0
24     }
25 }
```

The aspect *AroundAspect* contains one piece of around advice, which intercepts calls to *foo*. The pointcut *pc_foo* contains a formal parameter x, which is bound to the parameter of the function *foo* using the keyword *args*. This pointcut formal is then bound to the formal parameter x of the around advice in line 4. The result is that the advice checks the value of the parameter passed to *foo* and only invokes *proceed* if the parameter is non-zero. Otherwise, it returns 0.

2.2.3 Cflow pointcuts

Cflow pointcuts match a joinpoint when it is in the control flow of a pointcut. The control flow of a joinpoint is the set of instructions that are executed while the activation record of the joinpoint still resides in the program stack. As an example, the control flow of a function call consists of all the instructions that are executed while the activation record of the function call is still in the stack. The following listing shows the behavior of *cflow* pointcuts.

Listing 2.3: Cflow Pointcuts

```
1  aspect CFlowAspect {
2      pointcut pc_flow(): cflow(call(* foo(..))) && call(* bar(..));
3
4      before() : pc_flow() {
5          System.out.println("bar␣while␣in␣foo");
6      }
7  }
8
9  class BaseCflow {
10     public void foo() {
11         bar(); //the before advice matches on this call to bar
12     }
13     public void bar() {}
14
15     public static void main(String args[]) {
16         BaseCflow b = new BaseCflow();
17         b.foo();
18         b.bar(); //this does not match pointcut pc_flow
19     }
20 }
```

The pointcut *pc_flow* matches on calls to methods named *bar* with any return type and any set of parameters, as long as it is in the control flow of a call to a method *foo*. The call to *bar* in line 11 matches the pointcut since it occurs within the control flow of *foo*, but the call in line 18 does not.

2.2.4 Precedence

When multiple pieces of advice apply to a single joinpoint shadow, the order in which they are applied may result in different program behavior. By default, the order in which advice is applied is non-deterministic, i.e. it is up to the compiler. However, it is sometimes necessary to specify this order to ensure that the program runs properly. AspectJ provides the *declare precedence* declaration to specify the precedence order of aspects. This order is then used to determine the order of advice application

Listing 2.4: declare precedence

```
aspect PrecedenceAspect {
    declare precedence: AspectA, AspectB, AspectC;
}
```

The *declare precedence* statement above sets the precedence order for *AspectA*, *AspectB* and *AspectC*. *AspectA* has the highest precedence, while *AspectC* has the lowest precedence.

AspectJ allows multiple *declare precedence* statements, as long as they do not cause a precedence conflict. Aspect precedence is also global, in that if *AspectA* has a higher precedence than *AspectB*, then the advice of *AspectA* is always applied before advice of *AspectB* on all shadows in the program. This is true even if the order was chosen by the compiler, i.e. if the compiler non-deterministically chose to apply advice from *AspectA* before that from *AspectB* at a particular shadow, it should always apply *AspectA* before *AspectB* at all other joinpoint shadows.

2.3 AOP and Modularity

While aspect-oriented programming provides modularity to cross-cutting features, another property of program modularity is adversely affected by advice. As advice is executed at specific points in the base program, a change in the implementation of the base program can cause advice to behave in an unexpected manner.

The following example, based on an example by Aldrich [2], illustrates the problem. Listing 2.5 shows a simple *Figure* class, which is just a collection of points. It also has a *translate* method that moves the points by a specified displacement.

Listing 2.5: Simple Figure Class

```
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;
    }
}
```

Suppose a replay feature is implemented using an aspect. 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.

Listing 2.6: Replay Aspect

```
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
    }
}
```

If the implementation of *Figure* was changed such that the list can contain other figures as well as points, the behavior 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.

Listing 2.7: Modified Figure class

```
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;
    }
}
```

That such a seemingly innocuous change to *Figure* could change the behavior 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* was 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.

One solution to the problem is to define an interface between base code and its client aspects. The interface should provide a mechanism to the developer of the base code for limiting the extent of code to which aspects can be applied. The interface allows a contract to be established between the base code and the aspects: aspects will only apply to the exposed portions of the base code, while any changes to the base code must not break the aspects that advise the base code exposed by the interface.

This approach which has been taken by several previous authors [3, 5, 7]. Clifton and Leavens [3] use aspect maps to define the aspects that can be applied to specified classes or packages. Spectator aspects, which can be applied everywhere but are expected to merely "observe" the base code without modifying its behavior, are provided for features such as logging and debugging. Lieberherr et. al. [7] defines aspectual collaborations. A system is organized into collaborations and participants, which roughly correspond to packages and classes in Java. Points in the participants that can be modified are marked as *expected*, acting like abstract declarations but do not prevent instantiation of the participant. These points can be modified by providing a concrete implementation from another participant in a collaboration. Collaborations also support aspectual methods, which act in a manner similar to around advice in AspectJ. Kiczales and Mezini define aspect-aware interfaces [5] where the aspects and type of advice that can apply to a method can be specified.

These proposals all require that the aspect that will apply to a portion of code be specified in the interface. An addition of an aspect would require the modification of the existing interfaces to allow for its use. This is undesirable as it would make it difficult to integrate new aspects, especially aspects such as debugging or logging aspects that modify a large extent of code.

2.4 Open Modules

Open modules [2] is a proposal that defines an interface between the members of the module and advice that apply to them. Open modules is defined to be

> ...any module system that:
>
> - allows external advice to interactions between a module and the outside world (including external calls to functions in the interface of a module)
> - allows external advice to exposed pointcuts
> - does not allow external modules to directly advise internal events within a module, such as calls from within a module to other functions within the module (including calls to exported functions)

A module is a collection of code and signatures, which are exposed methods and pointcuts. Code can either be ordinary base code, or advice. Advice which are not part the module apply only to external calls to the exposed methods. This effectively allows the implementation of an exposed method to be changed without affecting the behavior of external aspects, as the changes are inside the module. A module can also expose a pointcut to capture internal events that cannot be expressed as a single method call. Exposing a pointcut comes with an implicit guarantee from the developer of the base code that the semantics of the pointcut will remain the same after any changes to the base code. Thus the behavior of client aspects are protected from modifications of the base code, while changes to the base code need only comply with the guarantee that the semantics of the exposed pointcuts remain the same.

Open modules also define a precedence order for the advice declared in the module. The advice that is declared last is applied first, followed by the second to the last, and so forth. In AspectJ this is equivalent to a declare precedence declaration with the aspects in the reverse order of their declaration in the module.

2.4.1 Example

Listing 2.8: Figure Module

```
module FigureModule {
    class Figure;
    aspect Aspect2;
    aspect Aspect1;
    sig {
        method Figure Figure.translate(int, int);
        //exposes the individual Point.translate() calls inside Figure
        pointcut pointTranslations();
    }
}
```

The example above shows Open Modules using a syntax closer to AspectJ than the *TinyAspect* language used in [2]. The module *FigureModule* contains the class *Figure* of the previous section, two aspects *Aspect1* and *Aspect2*, and two signatures: the first exposing *external* calls to *Figure.translate*, and the second exposing calls to *Point.translate* inside *Figure*. Advice that belongs to aspects not inside the module must match one of the signatures for it to apply to a shadow that belongs to the module.

The concept of shadow ownership is fully defined in the next chapter, for now it is sufficient to state that a call to a method declared in a class belongs to that class. Hence any calls to *Figure.translate* belong to the class *Figure*, as *translate* is declared in *Figure*.

The first signature exposes *external* calls to *Figure.translate*, and has the effect of conjoining the pointcut

```
!within(Figure) && !within(Aspect1) && !within(Aspect2)
```

to the pointcuts of external aspects when matching with joinpoint shadows that belong to the class *Figure*, in addition to the constraint that it matches a call to *Figure.translate*. *Aspect1* and *Aspect2* are also included in the expression as they may contain calls to *translate*, and Open Modules forbid the application of external advice to any code inside the module unless explicitly specified by the signature. When applied to *ReplayAspect*, the pointcut of the *before()* advice effectively becomes

```
call(* Figure.translate(int, int)) &&
    !within(Figure) && !within(Aspect1) && !within(Aspect2)
```

Thus the advice applies only to external calls to *Figure.translate*, preserving the proper behavior of *ReplayAspect* even after the modifications in Listing 2.7.

The pointcut signature *pointTranslate* is provided to allow matches with the individual *Point* translations that occur inside *Figure*. This is an example of an internal event that cannot be expressed as a single call to methods of *Figure*. By exposing *pointTranslate*, the developer of *FigureModule* makes the guarantee that *pointTranslate* will always mean the *Point* translations in *Figure*, regardless of any modifications made to *Figure*. The most natural definition of *pointTranslate* would be

```
call(* Point.translate(int, int)) && within(Figure)
```

Note that pointcut signatures do *not* have the implicit *!within* checks that method pointcuts provide, otherwise they would never match any internal events. Thus by exposing *pointTranslate*, the developer concedes control of all calls to *Point.translate* in *Figure*, *Aspect1* and *Aspect2* to external advice, as any external aspect can then have a piece of *around* advice which may completely bypass the call to *Point.translate()*. As such, great care needs to be taken when exposing internal events through pointcut signatures.

Since *Aspect1* is declared later than *Aspect2*, it is applied first. In effect, the module declaration implicitly creates the declaration

```
declare precedence: Aspect1, Aspect2;
```

Since *Aspect1* and *Aspect2* are in the same module as *Figure*, they are not affected by the signatures defined in the module. Any advice in either aspect will be applied to *Figure* as normal.

2.4.2 Formal Semantics

More formally, Open Modules was defined for a small functional aspect-oriented language *TinyAspect*. Figure 2.1, taken from [2], shows a cached Fibonacci function in *TinyAspect*.

```
val fib = fn x:int => 1
around call(fib) (x:int) =
    if (x>2)
        then fib(x-1) + fib(x-2)
        else proceed x

(*cache functions used in advice*)
val inCache = fn ...
val lookupCache = fn ...
val updateCache = fn ...

(*caching advice*)
pointcut cacheFunction = call(fib)
around cacheFunction(x:int) =
    if (inCache x)
        then lookupCache x
        else let v = proceed x
            in updateCache x v; v
```

Figure 2.1: Cached Fibonacci in *TinyAspect*

The Fibonacci function is implemented by a piece of around advice that checks if its parameter is greater than 2, and recursively calls *fib* if it is. Caching is provided by a second piece of *around* advice, which checks if a particular value has already been computed and exists in the cache. The semantics for *TinyAspect* is that the latest advice declared is applied first, hence the cache advice is called before the advice that implements recursion, which leads to the expected savings in execution time.

The semantics of *TinyAspect* is defined by a set of reduction rules that translate source code into a set of values, V. There are three types of values: expression values v, declaration values d_v and the call pointcut value $\texttt{call}(\ell)$. Declaration values are bound using $\equiv$ instead of $=$, to distinguish them from source declarations. Figure 2.2 details the form of each of these value types, as well as the forms of the possible evaluation contexts C. C shows how an expression can be reduced, with $\Box$ representing the term for reduction. In the definition of C, *bind* can be either `val` or `pointcut`. The order in the rules defining C also define the order of the reduction, i.e. reduction is first done on the left side of an application, then on the right side, then in the body of a `val` declaration, then on to the succeeding declarations.

$$\begin{array}{lcl}
v & ::= & () \mid \texttt{fn}\ x : \tau\ \texttt{=>}\ e\ \mid \ell \\
d_v & ::= & \bullet \mid \texttt{val}\ x \equiv \ell\ d_v \mid \texttt{pointcut}\ x \equiv \texttt{call}(\ell)\ d_v \\
V & ::= & v \mid d_v \mid \texttt{call}(\ell) \\
C & ::= & \Box e_2 \mid v_1 \Box \mid \texttt{val}\ x\ =\ \Box\ d \mid \mathit{bind}\ x \equiv V\ \Box \mid \texttt{pointcut}\ x\ = \Box\ d
\end{array}$$

Figure 2.2: *TinyAspect* values and contexts

Figure 2.3 contains the reduction rules for *TinyAspect*. Each state in the reduction is a pair of an expression e and an *advice environment* η, which maps labels ℓ to values. The notation $\eta[\ell]$ looks up the value of the label ℓ in the environment, while $\eta' = [\ell \mapsto v]\eta$ updates the mapping of ℓ to v, producing the new environment η'.

The application rule *r-app* is standard function application, where the actual parameter v is substituted into the formal parameter x in the expression e using the substitution notation $\{v/x\}e$. The *r-lookup* rule fetches the value associated with a label ℓ from η.

The *r-val* rule converts the body of a `val` declaration into a new label ℓ, updating the advice environment to map ℓ to the body of the `val` declaration and replacing all occurrences of the value name x with the label ℓ in the subsequent declarations d. These labels are used by pointcuts and advice to modify the values of the expression represented by the label, as the succeeding rules will show.

The rule *r-pointcut* merely converts the pointcut declaration into a pointcut value, and substitutes the pointcut value $\texttt{call}(\ell)$ to all occurrences of the pointcut name x in the succeeding declarations d.

$$\frac{}{(\eta, \texttt{fn}\ x : \tau\ \texttt{=>}\ e\ v) \mapsto (\eta, \{v/x\}e)}\ \textit{r-app} \qquad \frac{\eta[\ell] = v_1}{(\eta, \ell v_2) \mapsto (\eta, v_1 v_2)}\ \textit{r-lookup}$$

$$\frac{\ell \notin \mathit{domain}(\eta) \quad \eta' = [\ell \mapsto v]\eta}{(\eta, \texttt{val}\ x\ =\ v\ d) \mapsto (\eta', \texttt{val}\ x\ \equiv \ell\{\ell/x\}d)}\ \textit{r-val}$$

$$\frac{}{\begin{array}{c}(\eta, \texttt{pointcut}\ x\ = \texttt{call}(\ell)\ d) \mapsto \\ (\eta, \texttt{pointcut}\ x\ \equiv \texttt{call}(\ell)\ \{\texttt{call}(\ell)\ /x\}d)\end{array}}\ \textit{r-pointcut}$$

$$\frac{\begin{array}{c}v' = (\texttt{fn}\ x : \tau\ \texttt{=>}\ \{\ell'/\texttt{proceed}\}e\ ) \\ \ell' \notin \mathit{domain}(\eta) \quad \eta' = [\ell \mapsto v', \ell' \mapsto \eta[\ell]]\eta\end{array}}{(\eta, \texttt{around}\ \texttt{call}(\ell)\ (x : \tau) = e\ d) \mapsto (\eta', d)}\ \textit{r-around}$$

$$\frac{(\eta, e) \mapsto (\eta', e')}{(\eta, C[e]) \mapsto (\eta', C[e'])}\ \textit{r-context}$$

Figure 2.3: *TinyAspect* semantics

The rule *r-around* shows the changes to the advice environment and the value of a label caused by an around advice that attaches to a label ℓ. It creates a new label ℓ' mapped to the original value of the label that the around advice modifies (which is $\eta[\ell]$). The new label is used in the new value v', which is the body of the around advice, with the old value ℓ' substituted for all references to `proceed`. The advice environment is then updated to map the original label ℓ to the new value v', and the new label ℓ' to the old value $\eta[\ell]$. Thus an around advice modifies the return value of the function represented by the label ℓ, while still exposing the label for further use by other advice. This rule is responsible for advice that is last declared being applied first, as the last declared advice will be first in the chain of label-value pairs in η.

A set of type checking rules for *TinyAspect* is also provided, but for the purposes of this paper only the rule for the type of pointcuts is given. Given a label ℓ of type τ, then the pointcut $\texttt{call}(\ell)$ has the type $\texttt{pc}(\tau)$.

Open modules extends *TinyAspect* by introducing modules, defined using the *struct* and *sig* constructs. A module contains functions and advice, and

```
structure Math = struct
    val fib = fn x:int => 1
    around call(fib) (x:int) =
        if (x > 2)
            then fib(x-1) + fib(x-2)
            else proceed x

    (*caching advice*)
    pointcut cacheFunction = call(fib)
    around cacheFunction(x:int) =
        (*same as previous cache advice*)
end :> sig
    fib : int->int
end
```

Figure 2.4: Fibonacci example with Open Modules

even other modules, and defines a signature that applies to the contents of the module. Figure 2.4 shows how the Fibonacci example is encapsulated using Open Modules.

Figure 2.5 shows part of the operational semantics of *Open Modules*. These are additions to the rules for *TinyAspect* to define the semantics of *struct*s. Structures that are functors are left out, so as to focus on the effect of signatures. Modules are a sequence of declarations, and the value of a module m_v is the sequence of declarations values d_v of the declarations it contains. The signature β is a sequence of *val*, *pointcut* or *struct*s that are declared in the module. The effect of signatures is implemented by the judgement $seal(\eta, d, \beta) = (\eta', d_s)$, which maps an advice environment η, declarations d and a signature β into a new advice environment η' and sealed declarations d_s. Note that in the definition of the semantics, the label ℓ has the type τ.

The rule *r-seal* shows the conversion of a structure with a signature β into a structure that contains the sealed declarations d_{seal}, using the judgement *seal*.

The succeeding rules define *seal* itself. The rule *s-omit* shows how a *bind*, which can be a *val*, *pointcut* or *struct*, can be omitted from the signature. Note that $x : \tau$ is not part of the signature β, and hence should not be

$$\frac{seal(\eta, d_v, \beta) = (\eta', d_{seal})}{(\eta, \texttt{struct } d_v \texttt{end :>sig } \beta \texttt{ end}) \mapsto (\eta', \texttt{struct } d_{seal} \texttt{end })}\ \textit{r-seal}$$

$$\frac{}{seal(\eta, \bullet, \bullet) = (\eta, \bullet)}\ \textit{s-empty} \qquad \frac{seal(\eta, d, \beta) = (\eta', d')}{seal(\eta, bind\ x \equiv v\ d, \beta) = (\eta', d')}\ \textit{s-omit}$$

$$\frac{seal(\eta, d, \beta) = (\eta', d') \quad \eta'' = [\ell \mapsto \ell']\eta' \quad \ell \notin domain(\eta')}{seal(\eta, \texttt{val } x\ \equiv \ell'\ d, (x : \tau, \beta)) = (\eta'', \texttt{val } x\ \equiv \ell\ d')}\ \textit{s-v}$$

$$\frac{seal(\eta, d, \beta) = (\eta', d')}{\begin{array}{c} seal(\eta, \texttt{pointcut } x\ \equiv \texttt{call}(\ell)\ d, (x : \texttt{pc}(\tau), \beta)) = \\ (\eta', \texttt{pointcut } x\ \equiv \texttt{call}(\ell)\ d') \end{array}}\ \textit{s-p}$$

$$\frac{seal(\eta, d_s, \beta_s) = (\eta', d_s) \quad seal(\eta'', d, \beta) = (\eta'', d')}{\begin{array}{c} seal(\eta, \texttt{structure } x \equiv \texttt{struct } d_s \texttt{end}\ \ d, (x : \texttt{sig } \beta_s \texttt{ end}, \beta)) = \\ (\eta'', \texttt{structure } x \equiv \texttt{struct } d'_s \texttt{end}\ \ d') \end{array}}\ \textit{s-s}$$

Figure 2.5: Open Module semantics

exposed to external advice. This behavior is achieved by not mapping x to a label in the new advice environment η', thus making it invisible to external advice.

The rule *s-v* shows the effect of a *val* in the signature. A new label ℓ is created and mapped to the old label ℓ', and the declaration value `val` $x\ \equiv \ell'$ is converted to `val` $x\ \ \equiv \ell$, using the new label. Thus advice outside the module see x as ℓ, and can only advise external calls, as the old internal label ℓ' is not bound to x. Advice inside the module still associate x with the old label ℓ', and can advise internal calls.

The rule *s-p* shows the effect of *pointcut* declarations in the signature. In contrast to the previous rule, adding a pointcut to the signature exposes the pointcut to outside advice, using the same internal label ℓ.

Finally, the *s-s* rule shows the effect of signatures in nested structures. The structure x with body d_s and signature β_s is part of a larger structure with subsequent declarations d and signature β. The result is as expected: x is sealed first with the signature β_s, and then by β. In AspectJ terms, this is similar to conjoining the exposed signatures of the nested and the nesting module.

The sealing transformation effectively converts the declarations inside a *struct* so that only the calls to functions defined in the signature, either by a method signature or a pointcut, are bound to labels that can be advised by external advice. In the context of AspectJ, this is akin to converting the code in a compilation into (code, signature) pairs, where the signature is a pointcut defining the joinpoints in the code that can be advised by external advice.

One of the advantages of Open Modules is that it provides a proof that any changes to the members of the module will not affect client external aspects, as long as the constraints are enforced. For this, it provides a set of logical equivalence rules that check if two implementations of a module are indistinguishable to all clients. The rules check that the functions return the same value, and for functions that are part of a signature or are exposed using a pointcut, that both implementations trigger the external labels in the same manner. This is trivial for *val* declarations in the signature, as they are only triggered once at the start of each call, but can be very difficult to determine for internal labels exposed by *pointcuts*.

Another advantage is the relative obliviousness of the module to the external aspects that advise it. There is no need to specify the specific aspects that apply to the classes. Any external aspect may advise module members, as long as they also match one of the signatures of the module. This allows new aspects to be added to the compilation without the need to modify the existing module declarations.

The semantics formally define the behavior of Open Modules in *TinyAspect*, which is a language that is much simpler than full-featured aspect-oriented languages such as AspectJ. As an example, *TinyAspect* has a single pointcut primitive *call*, which matches only one particular function. AspectJ pointcuts allow for wildcards in a method pattern, and so can match multiple methods, methods which may not necessarily have the same parameter signature or return value. AspectJ also has a richer set of pointcut primitives, which include pointcuts that must be checked at runtime, such as *if* and *cflow*, for which no equivalent was defined in *TinyAspect*. As such, the implementation of Open Modules in AspectJ must be designed to integrate these advanced features while still providing modularity that is similar to that defined for Open Modules in *TinyAspect*.

2.5 The AspectBench compiler

The AspectBench compiler (*abc*) [1] is an AspectJ compiler developed to be an alternative to *ajc*. The compiler was designed for easy extensibility and for implementing compiler optimizations. The compiler is built on the Polyglot extensible compiler framework [9] and the Soot optimization framework [11].

Polyglot parses java source code and generates the corresponding abstract syntax tree (AST). Multiple passes over the AST set up the data structures that are used for conversion into Jimple, an intermediate representation used by the Soot framework. Jimple is finally converted to bytecode after analysis and optimizations.

Polyglot was designed to allow for extensions to the basic Java compiler and as such, AspectJ is implemented as an extension of the Polyglot java compiler. The AspectJ grammar is an extension of the Java grammar, and extensions to *abc* are also implemented as Polyglot extensions.

Abc adds passes to the Java compiler that extract the aspect-related information (*AspectInfo*) from the AST and are used later when inserting inter-type declarations and weaving advice code. Included in *AspectInfo* is the set of all advice declared along with their associated pointcuts. *Abc* has a total of 38 passes, which are grouped into 12 sets. Table 2.1 shows these pass sets in the order in which they are executed, along with a description of their purpose. Only three of these pass sets, namely *passes_patterns_and_parents*, *passes_precedence_relation*, and *passes_aspectj_transforms* are directly relevant to this dissertation.

Pass Set	**Description**
passes_parse_and_clean	Generates the AST by parsing the source files, and generates the type system to be used with the AST.
passes_patterns_and_parents	Collects basic aspect data such as aspect names and the aspect inheritance hierarchy. Also evaluates and stores the matches of class name patterns which will later be used in matching.

passes_precedence_relation	Generates the aspect precedence relation from declare precedence statements.
passes_disambiguate_signatures	Disambiguates method signatures.
passes_add_members	Adds members to class and aspect types.
passes_interface_ITDs	Processes intertype declarations in interfaces introduced by *declare parents* declarations.
passes_disambiguate_all	Removes any remaining ambiguities in the AST.
passes_fold_and_checkcode	Checks the code for other (non-syntax) errors, such as code unreachability.
passes_saveAST	Associates the current compilation with the AST to prevent recompilation.
passes_mangle_names	Mangle the names of members of intertype declarations.
passes_aspectj_transforms	Converts advice into methods, and collects all the aspect-related information into the global *AspectInfo*. Also removes AST nodes such as pointcut declarations to prepare for weaving.
passes_jimple	Converts the AST to the Jimple intermediate representation.

Table 2.1: AspectBench compiler passes

Matching in the AspectJ compiler is done by iterating through each piece of advice in the advice list for every weavable class in the compilation. In each weavable class, the pointcut associated with each advice is matched against every joinpoint shadow in the class.

Matching produces a *residue*, which conceptually is the code that checks whether an advice applies to a particular joinpoint shadow. Static matches such as a simple call to a method produce residues that represent true or false values, true indicating that the advice should be woven at the joinpoint shadow and false otherwise. Dynamic matches, caused by pointcuts such as *cflow*, need to be evaluated at run time. These produce residues that represent the code for checking the conditions at run time, which are then woven along with the advice that applies to the shadow.

Chapter 3

Open Modules in AspectJ

This chapter describes the design of the open module extension to AspectJ. Contained is the definition of the syntax for defining modules, a description of the modules themselves and their effect on internal and external aspects, and a description of module inclusion.

3.1 Syntax

A module declaration follows this general pattern:

```
module <module name> {
    [<class | aspect | [constrain] module> <member1>;]
    [<class | aspect | [constrain] module> <member2>;]
    ...
    sig {
        [[private] <pointcut | method> <signature1>;]
        [[private] <pointcut | method> <signature2>;]
        ...
    }
}
```

A module consists of a list of members, which are classes, aspects or other modules, and a list of signatures, which may either be method signatures or pointcut signatures. Module members may be constrained, and signature members may be private. Class member names may contain wildcards, similar to class name expressions in AspectJ. Aspect and module member names

⟨*module-declaraion*⟩ ::= 'module' ⟨*identifier*⟩ ⟨*module-body*⟩

⟨*module-body*⟩ ::= '{' ⟨*signature*⟩ '}'
  | '{' ⟨*module-members*⟩ ⟨*signature*⟩ '}'

⟨*module-members*⟩ ::= ⟨*module-member*⟩
  | ⟨*module-members*⟩ ⟨*module-member*⟩

⟨*module-member*⟩ ::= 'class' ⟨*classname-pattern-expr*⟩ ';'
  | 'aspect' ⟨*identifier*⟩ ';'
  | 'module' ⟨*identifier*⟩ ';'
  | 'constrain' 'module' ⟨*identifier*⟩ ';'

⟨*signature*⟩ ::= 'sig' '{' '}'
  | 'sig' '{' ⟨*signature-members*⟩ '}'

⟨*signature-members*⟩ ::= ⟨*signature-member*⟩
  | ⟨*signature-members*⟩ ⟨*signature-member*⟩

⟨*signature-member*⟩ ::= 'pointcut' ⟨*pointcut-expr*⟩ ';'
  | 'private' 'pointcut' ⟨*pointcut-expr*⟩ ';'
  | 'method' ⟨*method-constructor-pattern*⟩ ';'
  | 'private' 'method' ⟨*method-constructor-pattern*⟩

Figure 3.1: Open Module AspectJ Syntax in BNF

may *not* contain wildcards, i.e. you have to specify the exact aspect or module that is included in the module. This design decision is based on the assumption that classes will greatly outnumber aspects or modules in any given system, and the view that the decision to place a module or aspect in a module should be a deliberate choice on the part of the developer. Disallowing wildcards for aspects also has the effect of defining a strict precedence order for all aspects included in a module, which avoids problems caused by undefined aspect precedence [8].

Figure 3.1 defines the open module grammar in BNF. The grammar is an extension of the AspectJ grammar used by abc [1], and reuses several AspectJ non-terminal symbols.

3.2 Modules

Modules are the encapsulation constructs for Open Modules. Each module contains a list of member classes, aspects and modules, and a set of signatures that apply to its members.

Module declarations are placed in source files that are separate from Java or AspectJ source files. The modules themselves are not part of the Java or AspectJ namespace, and thus cannot be referenced by Java or AspectJ code. This separation allows modules to be added to or removed from a compilation with relative ease, as they will not require any modifications to the source code.

Each module in a compilation must have a different name. However, as the namespace for modules is different from that of aspects and classes, it is allowed to have a module with a name that is identical to that of a class or aspect.

The following listing shows a cached Fibonacci function using Open Modules, based on the Aldrich's Fibonacci example [2]. The Fibonacci function is implemented using the aspect *Fib* to provide the recursion, while the aspect *Cache* caches the results of previous calls to *fib()*.

Listing 3.1: Fibonacci Module Example

```
module FibMod {
    aspect Fib;
    aspect Cache;
    class Math;
    sig {
        method int Math.fib(int);
        //exposes cache misses
        pointcut Cache.cacheMiss();
    }
}

public class Math {
    public int fib(int x) { return 1; }
}

aspect Fib {
    pointcut fib(int x) : call(int Math.fib(int)) && args(x);

    int around(int x, Math m) : fib(x) && target(m){
        if (x < 3) { return proceed(x,m); }
        else { return m.fib(x−1) + m.fib(x−2); }
    }
}

aspect Cache {
    private Map cache = new HashMap();
    pointcut cacheMiss(): call(* Cache.cacheMiss(..)) && within(Cache);

    int around(int x) : Fib.fib(x){
        Integer i = (Integer)this.cache.get(new Integer(x));
        if (i != null) { return i.intValue(); }
        Integer result = new Integer(proceed(x));
        cacheMiss(x, result );
        return result.intValue();
    }
    private void cacheMiss(int x, Integer value) {
        cache.put(new Integer(x), value);
    }
}
```

3.2.1 Members

At its simplest form, a module includes a list of classes and aspects, and a set of signatures that apply to those classes and aspects. Member classes are preceded by the keyword *class*, and member aspects by *aspect*. Other modules may also be included by using the keyword *module*. The description of the behavior induced by including a module is non-trivial, and is deferred to the next section.

Member aspects must correspond to an existing aspect in the compilation. In contrast, member classes may or may not match any class in the compilation.

The order the member classes are declared is not significant, but the order of member aspects in the module declaration is used to determine aspect precedence. As such it is best to group all class members at the start or the end of the member list.

Classes and aspects may be a member of at most one module. This prevents the creation of a new module that circumvents the set of signatures that apply to a particular class or aspect. To add debugging aspects, one can create a module that includes the debugging aspect and the modules that it advises, which provides unrestricted access to members of the child modules.

3.2.2 Signatures

The set of signatures in a module defines the joinpoints of the members that are exposed to external advice. External advice are applied to module members only if they match at least one of the signature members in the list of signatures. In this sense the signature set defines a disjunction of the individual signature members.

There are two types of signatures: method and pointcut signatures.

Method Signatures

Method signatures expose a particular method or set of methods in the members to external advice. Method signatures are defined using the *method* keyword, followed by a method pattern. Method patterns are the same as those for AspectJ, and may contain wildcards.

Method signatures expose only the external calls to the method to external advice. In the Fibonacci example above, the method signature

```
method int Math.fib(int, int);
```

is equivalent to conjoining the pointcut

```
call(int Math.fib(int, int)) &&
    !within(Math) && !within(Fib) && !within(Cache)
```

to the pointcuts of external advice when matching calls to functions belonging to any of the module members.

Pointcut Signatures

Pointcut signatures are defined by the keyword *pointcut* followed by a pointcut expression. The pointcut expression in signatures is similar in power to pointcut expressions in AspectJ, except that expressions with formal parameters, such as *args(x)*, are not allowed as there are no formal parameters with which they can be bound. However, dynamic checks that do not use formal parameters, such as *args(int)*, are allowed.

In the Fibonacci example above, the named pointcut *Cache.cacheMiss()* is used to expose the internal event of a miss when trying to retrieve a value from the cache. This allows external aspects to apply advice to the call to *cacheMiss()* in *Cache*. As pointcut signatures do not allow formal parameters, the parameters to the cacheMiss function will have to be extracted using *thisJoinPoint*. This restriction, however, is not essential, and the addition of parameters to module signatures may be explored in a future design.

As has previously been mentioned, exposing internal events should be done with great care, as external aspects can weave *around* advice that does not call *proceed()*. Even in the simple example above, such an advice will completely nullify the *Cache* aspect.

Signatures and Internal Aspects

Signatures apply only to external advice, i.e. advice that are declared in aspects that are not members of the module. Aspects that are members of a module may apply their advice to other members without any restrictions.

In the example above, the *Cache* aspect applies its *around* advice to *Math*, *Fib* and itself without being restricted by the signature. Because of this, it matches the calls to *Math.fib()* inside the aspect *Fib*, while advice declared in external aspects do not.

Equivalent Signature

We note that any set of signature members can be converted to a single equivalent pointcut signature by putting the signature members in a disjunction. This idea of an equivalent signature indeed proves to be very useful in both the implementation and later in a proof about the behavior of module inclusion.

In the example above, the equivalent signature for *FibMod* is

```
(call(int Math.fib(int, int)) &&
        !within(Math) && !within(Fib) && !within(Cache)
)
|| Cache.cacheMiss()
```

Shadow Ownership

Signatures are taken into account only when matching shadows that belong to a module. Determining which shadows belong to which modules requires a definition of shadow ownership. A shadow's owning module is the module which signatures are used to constrain pointcuts that apply to that shadow. A shadow's owning module is usually the module that contains the shadow's enclosing class, i.e. the class where the shadow occurs. This is true for method bodies (execution shadows). There are notable exceptions however. Constructor calls, method calls and field references belong to the method or field's declaring class, that is, where the method or field was declared and not where the call or reference appears.

Listing 3.2: Shadow Ownership

```
module ModuleA{
    class A;
    sig {
        method * A.foo(..);
    }
}
class A {
    static void foo() {};
    static void all() {};
    static void zig() {
        foo();
    };
}
class B {
    static void ak() {}
    static void bar() {
        ak();
        A.foo();
        A.all();
    }
}
aspect ExtAspect {
    //matches all shadows, except those in ExtAspect
    pointcut pc_all() : !within(ExtAspect);
    before() : pc_all() {
        System.out.println(thisJoinPoint.getSignature());
    }
}
```

In listing 3.2, *ExtAspect* potentially matches all shadows in the program, except those in *ExtAspect* itself. However, class *A* is included in a module that only exposes calls to *foo*. The *before* advice does match the body of *bar*, and the body and call to *ak*, as the these shadows are owned by class B, and class B is not included in any modules and hence is not affected by any signatures. The calls to *A.foo()* and *A.bar()*, however, belong to class *A*, and hence is owned by *ModuleA*. Since *ModuleA* exposes only calls to *foo()*, the call to *A.all()* is not matched by the *before* advice. Note also that the call to *foo()* in *A.zig()* is not matched, since it is a call internal to *ModuleA*.

3.2.3 Precedence

The order in which the member aspects are declared in a module define the precedence of the aspects. Aspects that are declared later have a higher precedence.

In Listing 3.1, *Cache* has a higher precedence than *Fib*, as it was declared last in *FibMod*. This results in the expected behavior, as the cache should first be checked for a hit before proceeding to the recursive calls in *Fib*.

3.3 Module inclusion

As was previously mentioned, the list of module members may include other modules. Signatures are "inherited" by module members, and aspects are able to apply advice without restrictions to module members that are in the same module. This provides a way to organize modules and propagate the effects of aspects and signatures across multiple modules, and alleviates the limitations of disallowing classes and aspects from being a member of multiple modules.

Listing 3.3 shows an example of module inclusion. Three modules *ModuleA*, *ModuleB*, and *ModuleC* are shown. *ModuleB* and *ModuleC* are members of *ModuleA*. *ModuleB* is included using normal module inclusion, while *ModuleC* is included using constrained module inclusion. These inclusion modes shall be described in detail in the following sections.

Listing 3.3: Module Inclusion

```
module ModuleA {
    module ModuleB;
    aspect AspectA;
    constrain module ModuleC;
    class A;
    sig {
        method * f1(..);
    }
}

module ModuleB {
    aspect AspectB;
    class B;
    sig {
        method * f2(..);
    }
}

module ModuleC {
    aspect AspectC;
    class C;
    sig {
        method * f1(..);
        method * f2(..);
        method * f3(..);
    }
}
```

3.3.1 Module members

Member modules are declared using the *module* keyword followed by the name of the module. As with aspects and classes, a module can be a member of at most one module. In addition, cyclical membership is disallowed, as this will result in precedence conflicts should the modules in the cycle contain aspects. The above restrictions imply that module inclusion will produce a forest of one or more module trees, with each module in exactly one tree.

Member modules must also correspond to an existing module.

The members of an included module are considered to be internal for the purposes of aspect advice application. A class or aspect is considered internal by the members of the module to which it belongs and all the ancestors of that module in the inclusion tree. In the example above, the classes *B* and *C*, as well as the aspects *AspectB* and *AspectC* are considered to be internal to *ModuleA*, since *ModuleB* and *ModuleC* are members of *ModuleA*. Hence *AspectA* can apply its advice to these classes and aspects without being restricted by any signatures. It is, however, different in the context of the included modules. The members of the including (parent) module, except the included module itself, remain external to the members of the included (child) module. Thus advice in aspects *AspectB* and *AspectC* must satisfy the signature of *ModuleA* before they can be applied to *A* or *AspectA*.

Since members of included modules are considered to be internal, they also affect the equivalent pointcut of method signatures. In the example above, the signature of *ModuleA* includes *B*, *C*, *AspectB* and *AspectC* in the *!within* checks, which results in the equivalent signature

```
call(* f1(..)) &&
    !within(AspectA) && !within(A) &&
    !within(AspectB) && !within(B) &&
    !within(AspectC) && !within(C) &&
```

The order in which a member module appears in a module declaration is significant, and is used to determine the precedence of the aspects in the included module.

3.3.2 Signature Inheritance

The most significant effect of module inclusion is signature inheritance. The signature of a module is inherited by all of its member modules. The including module's signatures are not affected by the inheritance. There are two modes of signature inheritance: normal (disjunctive) inheritance and constrained (conjunctive) inheritance.

Normal Inheritance

In normal module inclusion, this inheritance involves the addition of the including module's signature to the signature of its member modules, effectively expanding the extent of code that is exposed by the included module. This allows a signature that would apply to multiple modules to be declared in a single place, avoiding duplication and possible errors. This could be especially useful for exposing logging or debugging points that are common to multiple modules.

Listing 3.4: Exposing a debug pointcut for multiple modules

```
module DebugModule {
    module Module1;
    module Module2;
    ...
    sig {
        pointcut DebugAspect.debugPointcut();
    }
}
```

In the example in listing 3.3, *ModuleB* is included in *ModuleA* using normal inheritance. As such, module inheritance is done using a disjunction, which results in *ModuleB* having the effective signature

```
method * f1(..);
method * f2(..);
```

It is noted that this form of signature inheritance is different from the result of sealing nested modules defined in [2]. The formal definition of Open Modules specify that the signature be applied to the nested structure, thereby hiding any labels not allowed by the signature of the containing structure. This is akin to conjoining the signatures of the contained and the containing modules. However, it is believed that the ability to *add* a common signature to multiple modules will be more useful in practice, and as such is the default behavior of signature inheritance. Signature inheritance that is more akin to Open Modules as defined in [2] is provided by constrained inheritance, described in the next section.

Constrained Inheritance

Constrained inheritance provides an alternate method for signature inheritance, conjoining the signature of the including module to that of the included module. This provides a mechanism for enforcing constraints that apply to multiple modules, such as an *if* pointcut that checks a boolean value at runtime to effect run-time enabling and disabling of aspects.

Listing 3.5: Run-time toggling of aspects across multiple modules

```
module ToggleAspectModule {
    constrain module Module1;
    constrain module Module2;
    ...
    sig {
        pointcut if(ToggleAspect.aspectsEnabled);
    }
}
```

In the example in Listing 3.3, *ModuleC* is included in *ModuleA* using constrained inclusion. The signature of *ModuleA* is conjoined with that of *ModuleC*, which results in *ModuleC* having an effective signature of

```
method * f1(..);
```

Private Signatures

Private signatures provide a mechanism for adding a signature that affects the member classes and aspects of a module, but is not inherited by member modules. This allows the addition of signatures to a module without worrying about its effects on member modules.

Private signatures are declared by prepending the keyword *private* to method or pointcut signatures.

Listing 3.6: Private signatures

```
module ModuleA {
    module ModuleB;
    aspect AspectA;
    constrain module ModuleC;
    class A;
    sig {
        method * f1(..);
        private method * f4(..);
    }
}
```

3.3.3 Precedence

The precedence of aspects in the included modules is determined by the order of the declarations in the including module. Precedence of modules is determined similar to the way it is determined between aspects, in that the last declared has highest precedence. If a member module is declared after an aspect, then the member aspects of the member module have higher precedence than the aspect. The same criterion is used to determine the precedence between two member modules.

In Listing 3.3, *ModuleB* is declared to be a member of *ModuleA* before *AspectA*, while *ModuleC* is declared after. This implies that the member aspect of *ModuleC*, namely *AspectC*, has a higher precedence than *AspectA*, while the member aspect of *ModuleB*, namely *AspectB*, has a lower precedence. This results in a strict precedence order equivalent to the *declare precedence* declaration

```
declare precedence: AspectC, AspectA, AspectB;
```

Since AspectJ only allows for a single global ordering of aspects, this precedence order applies to all applications of advice from the three aspects. This provides a way to fully specify the order of aspects and avoid the non-determinism caused by aspects unrelated in the precedence relation.

This strict ordering caused by module inclusion is another reason for the constraint that aspects are allowed to be a member of at most one module, as membership in multiple modules may cause precedence conflicts that are not allowed in AspectJ.

Chapter 4

Implementation

This section describes the implementation of Open Modules in the AspectBench compiler (*abc*). Open modules was implemented as an extension (*openmod*) of the AspectBench compiler. The implementation contained 2700 lines of code, and an additional 1500 lines of code for tests.

The *abc openmod* extension consists of three main components:

- the syntax extensions, which include the new AST node types;
- the internal module representation, which also contains the methods that modify the matching behavior of advice pointcuts;
- and the compiler passes, which process the *openmod* AST nodes.

4.1 Syntax Extensions

The *openmod* extensions to the AspectJ syntax was defined using Polyglot's parser generator (*PPG*). *PPG* uses a yacc-like syntax for defining grammars, and also allows for the extension of existing non-terminal symbols through the use of the *extends* keyword. The *openmod* grammar uses non-terminals of the AspectJ grammar, which itself is an extension of the Java grammar used for the *CUP* java parser.

Each rule is expected to produce a return value, typically an AST node or a list of AST nodes that represent the non-terminal on the left-hand side of the rule. These return values are then used by Polyglot to build the AST.

The top-level non-terminal of Open Modules, the *module_declaration*, was defined as an additional production of *compilation_unit*, the non-terminal

$\langle$*compilation-unit*$\rangle$ ::= ...
  | $\langle$*module-declaration-list*$\rangle$

$\langle$*module-declaration-list*$\rangle$ ::= $\langle$*module-declaration*$\rangle$
  | $\langle$*module-declaration-list*$\rangle$ $\langle$*module-declaration*$\rangle$

Figure 4.1: Grammar extensions to *compilation-unit*

that represents source files. Multiple module declarations are allowed in a single file, but once a source file contains a module, all its other contents must also be modules.

Extending the grammar also included the definition of new AST nodes to represent the non-terminals. The following table provides a summary of the data structures used.

Name	**Description**
ModuleDecl_c	A module declaration. Contains the name of the module and the module body.
ModuleBody_c	Represents the body of the module. Contains a list of module members and signature members.
ModMemberAspect_c	A member aspect. Contains the name of the member aspect.
ModMemberClass_c	Represents a member class. Contains the *classname_pattern_expr* that is used to match the classes that belong to the module.
ModMemberModule_c	A member module. Contains the module name, and a boolean value that is set if constrained inclusion is used.
SigMemberMethodDecl_c	A method signature. Contains a pointcut that is equivalent to the method signature.
SigMemberPCDecl_c	A pointcut signature. Contains the pointcut to be used as a signature.

DummyAspectDecl_c	A dummy aspect declaration associated with a ModuleDecl_c. This is used to generate a dummy aspect for each module that is used as a container for code generated by dynamic matches, specifically by cflow matches.

Table 4.1: openmod AST node data structures

Open module AST nodes are, for the most part, unaffected by the java and AspectJ compiler passes. With the exception of possible precedence conflicts with *declare precedence* statements, module files may be put into or taken out of a compilation without affecting the property of it being a valid AspectJ program. At worst, warnings may be generated, due to the effect of modules on pointcut matching and precedence. Existing *declare precedence* statements may conflict with the aspect precedence order specified by the modules, but this can be remedied by changing the aspect precedence order in the modules to be consistent with the *declare precedence* statements in the code.

Such a separation of modules from java and AspectJ code should ease the integration of modules into existing AspectJ software projects, as no changes are required to the codebase to allow for the use of modules. Indeed, using modules may reveal hitherto hidden flaws in the codebase caused by aspects that violate the signatures of the modules.

The classes that represent signature members, *SigMemberPCDecl_c* and *SigMemberMethodDecl_c*, store the equivalent pointcut of the signature member. These stored pointcuts are put in conjunction with the pointcuts of advice during matching. Storing the equivalent pointcut is straightforward for pointcut signatures, as a pointcut expression is provided with the signature declaration. Method signature declarations only contain a method pattern, and as such requires conversion to an equivalent pointcut. This is done by creating a *call* pointcut that uses the method pattern as an argument, and later adding the *!within* checks for external calls once the set of internal classes and aspects have been fully resolved. For example, a method signature

```
method * foo(..);
```

is first converted to the *call* pointcut

```
call(* foo(..))
```

Once the set of internal classes and aspects has been fully determined, the *!within* constraints that check that it is an external call are added

```
call(* foo(..)) && !within(class1) && !within(class2) ...
```

Cflow pointcuts caused a significant issue during the implementation of *openmod*, and required the addition of a node to the AST. *Abc* implements *cflow* by creating a counter that is incremented at the start of a call that matches the *cflow* argument, and is decremented at the end of the call. The following listings show an AspectJ program that uses a *cflow* pointcut and the equivalent java code that is produced after weaving.

Listing 4.1: An AspectJ program that uses a cflow pointcut

```
class A {
    public static void foo() {}
    public static void test() { foo(); }
    public static void main(String args[]) {
        test();
    }
}
aspect AspectA {
    pointcut pc() : cflow(call(* A.test(..))) &&
                    call(* foo(..)) && !within(AspectA);
    before() : pc() {
        System.out.println("Before foo() while in test()");
    }
}
```

Listing 4.2: The equivalent java program that is produced after weaving

```
class A {
    public static void foo() {}
    public static void test() {
        //tests the counter before calling the before advice
        if(AspectA.abc$cflowCounter$0.getThreadCounter().count > 0) {
            ...
            AspectA.aspectOf().before$0(staticpart);
        }
        foo();
    }
    public static void main(String args[]) {
        Counter c = AspectA.abc$cflowCounter$0.getThreadCounter();
        //increment the counter before test, and decrement it after
        c.count++
        test();
        c.count--;
    }
}
public class AspectA {
    //the cflow counter
    public static CflowCounterInterface abc$cflowCounter$0;
    //the counter initialization
    private static void abc$preClinit() {
        abc$cflowCounter$0 = CflowCounterFactory.makeCflowCounter();
    }
    //the before advice
    public final void before$0(StaticPart staticpart) {
        System.out.println("Before foo() while in test()")
    }
    ...
}
```

As the example shows, this counter is normally placed in the aspect in which the *cflow* was declared. As modules are non-aspect top level declarations, it was necessary to create a blank dummy aspect, *DummyAspect-Decl_c*, associated with each module to contain any cflow counters which may be created due to a *cflow* signature. Other than *cflow* initializations, the dummy aspect contains no other code, such as advice, which may interfere with the behavior of the rest of the program. Having a dummy aspect for each module results in a class file *ModuleName_DummyAspect.class* to be created for each module included in the compilation. The following listing shows a module declaration that has a *cflow* signature, and the dummy aspect that is generated after compilation.

Listing 4.3: CFlow module and associated dummy aspect

```
module Module {
    class A;
    __sig {
        pointcut cflow(call (* A.a()));
        pointcut cflow(call (* A.d()));
    }
}

//The dummy aspect associated with Module
public class Module_DummyAspect {
    public static CflowCounterInterface abc$cflowCounter$0;
    public static CflowCounterInterface abc$cflowCounter$1;
    private static void abc$preClinit() {
        abc$cflowCounter$0 = CflowCounterFactory.makeCflowCounter();
        abc$cflowCounter$1 = CflowCounterFactory.makeCflowCounter();
    }
     ...
}
```

The parent-child relationship between the nodes in the AST is straightforward: given a non-terminal on the left-hand side of the rule, its children are the non-terminals on the right-hand side. There is, however, a single exception: a *DummyAspectDecl_c* has the *ModuleDecl_c* to which it is associated as its child. This makes passes process the *DummyAspectDecl_c* just before the *ModuleDecl_c*, making it easy to store the association in the module declaration and ensuring that passes go though both the module and its associated dummy aspect exactly once.

4.2 Module Representation

The *openmod* extension creates an internal representation of the modules that mirrors the contents of *openmod* nodes in the AST. This is necessary as *abc* "cleans" the AST to prepare for weaving, removing non-java nodes such as pointcut declarations and expressions. Matching and weaving is done after the AST has been cleaned. As pointcut signatures contain pointcut

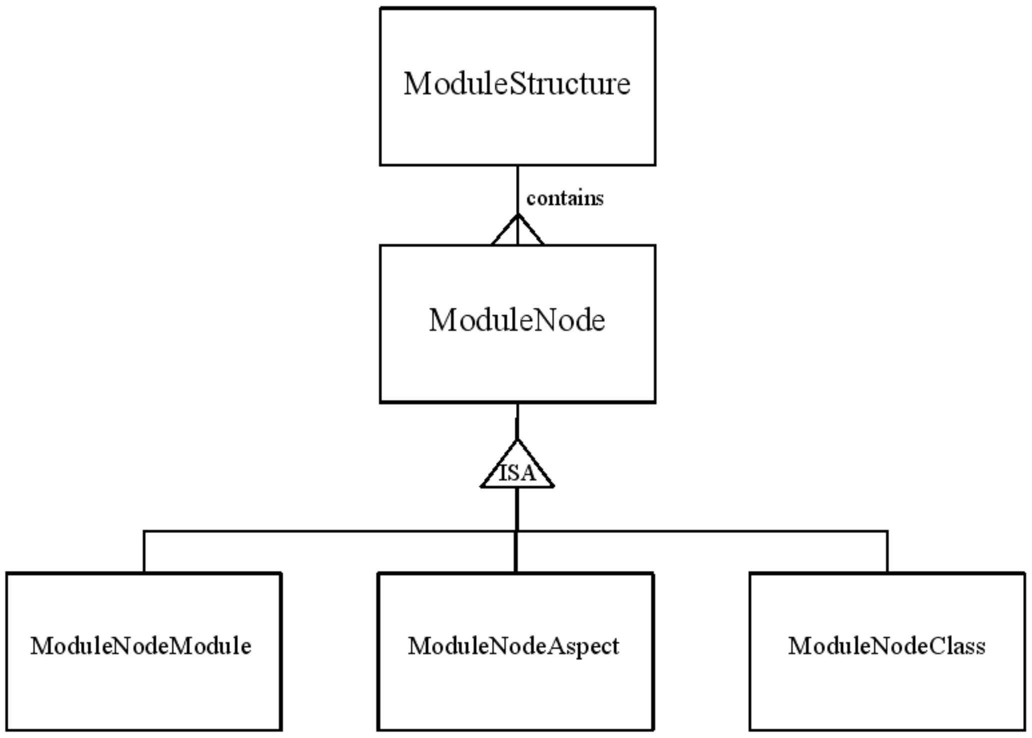

Figure 4.2: Internal Module Representation Data Structures

expressions and are used during matching, a separate internal representation of these nodes must be created for use during matching. Open modules also specify that internal aspects should apply without limitations to internal classes, requiring that modules and their members be stored in some form that allows checking for such conditions.

4.2.1 Data Structures

The internal representation of modules uses nodes to represent module, class and aspect members. The abstract class *ModuleNode* represents a module, class or aspect, which are then represented by the subclasses *ModuleNode-Module*, *ModuleNodeClass* and *ModuleNodeAspect* respectively. Each *ModuleNodeModule* has one upward link to its parent and a downward link for each of its children. *ModuleNodeAspect* and *ModuleNodeClass* only have an upward link to their parent module.

A *ModuleNodeModule* also contains equivalent pointcuts of its signatures, one for non-private signatures and one for private signatures. These two signatures are disjunctions of the individual signature members, and are updated upon the addition of a new signature member. It was necessary to separate the private pointcuts from the non-private pointcuts so that only non-private pointcuts are inherited by included modules.

ModuleNodes are contained in a singleton *ModuleStructure*. *ModuleStructure* contains the module nodes in three separate hash maps indexed by name, one hash map for each type of *ModuleNode*. Separate hash maps had to be provided as the namespace of modules is different from that of aspects and classes, thus a module may have the same name as an aspect or a class. The compiler uses several different sets of data structures: Polyglot AST nodes at the first stage of the compilation and Soot classes during matching and weaving. Indexing the set by name allows a *ModuleNode* to be accessed at many stages of the compilation process, as both Polyglot and Soot classes provide methods to access the name of a class or aspect.

4.3 Compiler Passes

The *openmod* extension introduces seven new passes into the compiler. These additional passes are mainly concerned with generating the internal representation of modules, checking non-syntax errors and generating the precedence relation implied by the order of the aspects in a module declaration. Table 4.2 summarizes the new passes in the order that they are run, and provides a short description for each.

Pass Name	**Description**
CollectModules	Creates *ModuleNodeModule*s from module declarations and adds them to the *ModuleStructure*.

CheckModuleMembers	Adds *ModuleNodeClass* and *ModuleNodeAspect*s with the appropriate parent reference to the *ModuleStructure*, and updates the parent references of modules that are members of other modules. Also checks for duplicate aspect/module inclusion, as well as for the existence of the members in the compilation.
CheckModuleCycles	Checks for any cyclical module inclusions.
CheckDuplicateClassInclude	Checks if any classes are included in more than one module.
OM_ComputePrecedence	Generates the precedence relation that is defined by the sequence of included aspects in the module declarations. This comes before AspectJ's *ComputePrecedence* pass, so any precedence conflicts will produce errors that refer to the inconsistent *declare precedence* statement.
CollectModuleAspects	Attaches the *Aspect* representation of a the dummy aspect to its associated *ModuleNodeModule*. *Aspect* is the representation of aspect declarations for *abc*'s *AspectInfo*. This pass occurs after all the *AspectInfo* has been gathered, but before the clean up of the AST prior to weaving.
CheckModuleSigMembers	Adds the signature members to their parent *ModuleNodeModule*. This pass occurs just before the clean up of the AST.

Table 4.2: Open module compiler passes

4.3.1 Internal Representation Generation

The *CollectModules*, *CheckModuleMembers*, *CollectModuleAspect* and *CheckSigMembers* passes generate the internal module representation stored in the *ModuleStructure*. *CollectModules* and *CheckModuleMembers* are run relatively early in the compilation, just after the *pass_patterns_and_parents* pass set of AspectJ, while *CollectModuleAspect* and *CollectModules* occur in the pass set *passes_aspectj_transforms*, just before the AST is cleaned up prior to Jimplification and weaving.

CollectModules only processes module declarations, creating the representative *ModuleNodeModule* and adding it to the *ModuleStructure*. It would have been possible to add the member classes, aspects and modules at this point, but it would not have been possible to check for the erroneous inclusion of a member in more than one module. As such the addition of members were deferred to another pass.

CheckModuleMembers creates *ModuleNodeClass* and *ModuleNodeAspect* nodes for class and aspect members, pointing the parent reference to the including module. It does the same for member modules, by searching for the corresponding *ModuleNodeModule* and updating its parent reference. This pass also checks the existence of a member aspect or member module in the compilation, as well as for duplicate inclusion of modules and aspects. Checking for duplicate inclusion of classes, however, is deferred to another pass, as the *abc* compiler provides no method for returning the set of classes that match a particular class name pattern. It does, however, provide a method that returns true if a given class matches a particular name pattern, pointing to an alternative method for checking duplicate class inclusions, done in the pass *CheckDuplicateClassInclude*.

Adding a member class, aspect or module to a *ModuleNodeModule* also updates its *extPointcut*. The *extPointcut* represents the *!within* checks for verifying external calls. An addition of a member conjoins the existing *extPointcut* with a *!within* pointcut with the member as a parameter.

CollectModuleAspect updates a reference in *ModuleNodeModule* to the *Aspect* representation of its associated dummy aspect. *Aspect* is the representation for aspect declarations in *AspectInfo*, and is used during matching.

The *CollectModuleAspect* pass occurs after all the *AspectInfo* has been collected, but before the clean up of the AST.

CheckModuleSigMembers adds the signature members of a module declaration. This pass occurs just before the clean up of the AST. The pass was deferred to this late stage as *cflow* pointcut signatures must be associated with their corresponding counter initializations and *CflowSetup*s before they can be used for matching. A *CflowSetup* is just an advice that implements the increment and decrement of the counter before and after the call, and is created and associated with the *cflow* pointcut during the passes in *passes_aspectj_transforms*.

4.3.2 Precedence

The *OM_ComputePrecedence* pass generates the precedence relation that is defined by the order of the inclusion of member aspects in modules. The precedence relation in *abc* is implemented as a simple map, with an aspect mapped to the set of aspects which have been defined to be of lower precedence relative to it. The generation of this relation is a straightforward matter of reading through aspect members and adding them to the relation, taking module inclusion into account.

4.3.3 Non-Syntax Error checking

Two passes deal exclusively with non-syntax error checking: *CheckModuleCycles* and *CheckDuplicateClassInclude*. In addition, the pass *CheckModuleMembers* also performs error checks on module members, as has been previously mentioned. Both passes are run after the execution of *abc*'s *passes_patterns_and_parents*.

CheckModuleCycles checks for any cyclical module inclusions by following the upward link from each module, and generating a compile error if it encounters the module again.

CheckDuplicateClassInclude checks if any class was included in more than one module. This is done by matching each class declaration with the class members of all modules, and generating a compile error if more than one module produces a match.

4.4 Matching

Matching occurs after the completion of all compiler passes. Matching determines which aspects apply to which joinpoint shadows. The results of matching are used later by the weaver.

4.4.1 Basic Matching

The matching process goes through all joinpoint shadows, and checks to see if any advice should be applied to a shadow. Each advice has an associated pointcut, which is used to match against a shadow.

The matching procedure produces a *residue*, which is conceptually the code that checks whether a particular advice should apply to a shadow. As have been previously mentioned, the resulting residue may be a static residue, which means that the advice is always applied to the shadow or it is never applied to the shadow, or a dynamic residue, which performs runtime checks to determine if the advice is applicable. Residues may be negated, or combined with other residues using conjunction or disjunction.

After the matching, the residue is woven into the code along with the advice at the joinpoint shadow. For static residues, no additional code is generated. Dynamic residues create code that perform the runtime checks on whether to invoke its associated advice.

4.4.2 Matching and Signatures

Module signatures alter the matching behavior by adding additional constraints on the pointcuts of the advice that are being matched. A module's effective signature is conjoined with advice pointcuts when they are being applied to shadows that belong to a member of the module. Otherwise matching proceeds as normal.

The effective signature of a module is the disjunction of its public and private signatures, plus any effects due to signature inheritance. Effects to the effective signature due to inheritance are resolved by simply following the upward links of the *ModuleNodeModule*, and combining the public signatures of its ancestors, taking into consideration whether the inclusion was constrained or unconstrained.

The residue that results from matching the conjunction of the effective signature and the advice pointcut against the shadow is the value that is passed to the weaver. If the residue is dynamic, then the weaver will weave the residue code together with the advice.

4.4.3 abc Implementation Issues

Though the *abc* matcher was designed for extensibility, several issues were exposed during the implementation of the *openmod* matcher. While there is a provision for adding new pointcuts and shadows, there is no easy method to modify the behavior of existing pointcuts. Implementing *openmod* through subclassing would have meant a total redefinition of all the existing matching classes to introduce the behavior added by signatures. This would have made the *openmod* extension itself inflexible, as every new pointcut introduced by other extensions must be modified to implement signature matching.

Another way to have implemented the change would have been to override the method that does the iteration through shadows, and introduce the behavior of signatures. The method, however, was static and hence cannot be overridden. In any case, overriding the method would have meant a total re-implementation, with the signature matching code inserted in the middle of the method, which would have made the overriding method ignore any changes to its super implementation.

In the end, it was decided that the easiest solution is to accept that there are existing problems in the extensibility of *abc* and to just directly modify the *abc* code to implement *openmod*.

Chapter 5

Formal Definitions and Proofs

The formal definition of Open Modules outlined in chapter 2 effectively transforms the code in a compilation into (code, signature) pairs, the signature specifying the joinpoints in the code where external advice can be applied. This chapter defines abstractions for modules, aspects and signatures of the implementation, as well as a transformation from a set of Open Modules to (code,signature) pairs, and shows that this transformation does not violate the aspect precedence rules of AspectJ.

This chapter uses Z-like notation, but takes liberties with the Z language [12] when the meaning of an expression is sufficiently clear.

5.1 Basic Definitions

$$\begin{array}{l}
[\mathcal{C}] \\
A = \mathbb{P}\,\mathcal{C} \rightarrow \mathbb{P}\,\mathcal{C} \\
\bigoplus ::= \vee | \wedge \\
\Sigma ::= p \mid \neg\, p \mid \sigma_1 \oplus \sigma_2
\end{array}$$

Figure 5.1: Definitions for code, aspects and signatures

Figure 5.1 contains the basic definitions for code, aspects and signatures.

$\mathcal{C}$ is the type of all possible code. This represents class and aspect code in a compilation.

A is the type of aspect applications. Aspect applications are defined as a function from code to code. Note that an aspect application only represents the transformations the aspect will perform on a given piece of code. It does not represent the code of the aspect itself.

Abstracting aspects as transformations of source code are based on the implementation of aspect-oriented compilers, and not the more common conceptual view of aspects as entities that dynamically observe the behavior of a program and is triggered by certain specified events.

The type $\bigoplus$ contains the logical operators $\vee$ and $\wedge$ as used in signatures.

The type Σ is the type of signatures. A signature is a primitive pointcut p or its negation, or two signatures σ_1 and σ_2 separated by an operator $\oplus : \bigoplus$.

5.2 Modules

5.2.1 Module Abstractions

$$\begin{aligned}
&M_{\oplus} = M \times \bigoplus \\
&M = \operatorname{iseq}(A \cup M_{\oplus}) \times \mathbb{P}\, C \times \Sigma \\
&M_A = \operatorname{iseq} A \times \mathbb{P}\, C \times \Sigma
\end{aligned}$$

Figure 5.2: Definition of modules and basic modules

The type of modules M is a tuple that consists of an injective sequence of aspect applications and modules, a set of code and a signature. An injective sequence is a sequence that contains no repeating elements.

Given a module (s, C, σ), the sequence s models the sequence of aspects and modules declared in a module declaration. Modules in the sequence are paired with an operator $\wedge$ or $\vee$ to model the default and constrained module inclusion modes defined for the implementation of Open Modules in AspectJ. Aspects in this sequence only represent the application of an aspect, not its code, which is included in the set of code C.

The set of code C defines the code that belongs to the module, which corresponds to the classes and aspects declared in a module declaration. This defines the set of code to which the signature applies.

The signature σ is the signature that applies to the set of code C. Since the implementation allows for both non-private and private signatures, module signatures may also be expressed as a disjunction $\sigma_{pub} \vee \sigma_{priv}$ of the public and private signature of a module.

The type M_A is a subtype of M, in that the sequence can contain only aspects. Modules that are of type M_A are referred to as basic modules.

5.2.2 Notation Conventions

In the interest of brevity, a few conventions on notation are defined for this and the following sections. The variable a and its variants are assumed to be aspect applications, that is, of type A. The variable m and the tuple (s, C, σ) and their variants are assumed to be of type M. The operator variable $\oplus$ is assumed to be of type $\bigoplus$.

As in Z, sequences are modelled as a mapping from integers to elements of a certain type. Hence the range of a sequence is the set of values that are in the sequence.

5.2.3 Valid Open Module Sets

Figure 5.3 defines the type of valid open module sets S_M. A finite set of modules S is a valid open module set if and only if it satisfies the following conditions:

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.
4. If an aspect application is in a module, then the code of that aspect must also be in that module.
5. A piece of code c is included in at most one module.

The definition of S_M uses the relation **in** and the function *code*. The relation **in** relates $x : (M \cup A)$ and $m = (s, C, \sigma)$ if x appears in s. In the context of the open module implementation, a module or aspect x is in

$$
\begin{array}{l}
S_M \subset \mathbb{P}\, M \\
\forall\, S : S_M \mid m \in S \wedge a \ \mathbf{in}\ m \bullet \\
\qquad \nexists m' \mid m' \in S \bullet m' \neq m \wedge a \ \mathbf{in}\ m' \\
\forall\, S : S_M \mid m \in S \wedge m' \ \mathbf{in}\ m \bullet m' \in S \\
\forall\, S : S_M \mid m, m' \in S \wedge m \ \mathbf{in}\ m' \bullet \\
\qquad (\nexists m'' : M \mid m'' \in S \bullet m'' \neq m' \wedge m \ \mathbf{in}\ m'') \wedge \\
\qquad \nexists \{m_1, m_2, ..., m_n\} \subseteq S \mid n \geq 1 \bullet m \ \mathbf{in}\ m_1 \wedge m_1 \ \mathbf{in}\ m_2 \wedge ...m_n \ \mathbf{in}\ m \\
\forall\, S : S_M \mid (s, C, \sigma) \in S \wedge a \ \mathbf{in}\ (s, C, \sigma) \bullet \mathit{code}(a) \in C \\
\forall\, S : S_M \mid (s, C, \sigma) \in S \wedge c \in C \bullet \\
\qquad \nexists (s', C', \sigma') \in S \bullet (s, C, \sigma) \neq (s', C', \sigma') \wedge c \in C'
\end{array}
$$

$$
\begin{array}{l}
\ \mathbf{in}\ : \mathbb{P}((M \cup A) \times A) \\
\ \mathbf{in}\ = \{((s, C, \sigma), (s', C', \sigma')) \mid ((s, C, \sigma), \oplus) \in \operatorname{ran}(s')\} \cup \\
\qquad \{(a, (s, C, \sigma)) \mid a \in \operatorname{ran}(s)\} \\
\mathit{code} : A \rightarrow \mathcal{C}
\end{array}
$$

$$
\begin{array}{l}
\mathit{toplevel} : S_M \rightarrow \mathbb{P}\, M \\
\mathit{toplevel}(S) = \{m \in S \mid \nexists m' \in S \bullet m \ \mathbf{in}\ m'\}
\end{array}
$$

Figure 5.3: Definition of valid Open Module sets S_M

a module m if x is included in the module declaration of m. The function *code* merely maps an aspect application to its corresponding aspect code.

The figure also contains the definition of the function *toplevel*, which returns the top-level modules of a valid module set, that is, the modules which are not included in any other modules in the set.

5.3 Basic Module Conversion

As previously mentioned, a basic module is a module which only includes aspects, and not other modules. A basic module can be converted into a set of (code, signature) pairs by applying the aspects to the code in the reverse order they appear in the sequence. Figure 5.4 contains the definition of the *compile* function, which performs this conversion.

$$
\begin{aligned}
&\mathit{compile} : \mathbb{P}\, M_A \to \mathbb{P}(\mathcal{C} \times \Sigma)\\
&\mathit{compile}(S) = \{(\mathit{apply}(s, C), \sigma) \mid (s, C, \sigma) \in S\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{apply} : \mathrm{iseq}\, A \times \mathbb{P}\,\mathcal{C} \to \mathbb{P}\,\mathcal{C}\\
&\mathit{apply}(\langle\rangle, C) = C\\
&\mathit{apply}(\langle a\rangle, C) = a(C)\\
&\mathit{apply}(\langle a\rangle \frown s, C) = a(\mathit{apply}(s, C))
\end{aligned}
$$

Figure 5.4: Definition of *compile*

The resulting (code,signature) pairs are now the ones that are visible to external advice. Since an external advice is applied to a joinpoint only if the conjunction of its pointcut and the signature matches the joinpoint, this effectively simulates the behavior of Open Modules as defined in [2], where only a select set of labels defined by the signature is available for external advice. However, a formal model and proof matching the behavior of Open Modules in *TinyAspect* and in AspectJ is beyond the scope of this dissertation.

5.4 Module Inclusion

The *compile* function is only defined for basic modules. To compile modules that include other modules, a conversion of modules in general to basic modules must first be defined.

Before the definition of the conversion, a short note on notation. The operator $\upharpoonright$ filters a sequence so that its members are of a certain type. For example, $s \upharpoonright A$ filters the sequence s so that it contains only entities of type A, maintained in the same order as they were in s. As an example, $\langle a_1, a_2, m_1, a_3, m_2, a_4\rangle \upharpoonright A = \langle a_1, a_2, a_3, a_4\rangle$. The right hand side of the $\upharpoonright$ operator can also be a set, in which case the result is the sequence filtered so that only elements which are in the set remain. The $\frown$ operator concatenates two sequences. The $\top$ value is the top signature value, and has the property $\sigma \wedge \top = \sigma$ for any signature σ.

The *convert* function transforms a valid open module set into a set of basic modules. The function itself is the union of the results of the function

$$\begin{aligned}
&\mathit{convert} : S_M \rightarrow \mathbb{P}\, M_A \\
&\mathit{convert}(S) = \bigcup\nolimits_{m \in \mathit{toplevel}(s)} \mathit{include}(\langle (m, \wedge) \rangle, \top)
\end{aligned}$$

$$\begin{aligned}
&\mathit{include} : \mathrm{seq}(A \cup M_\oplus) \times \Sigma \rightarrow M_A \\
&\mathit{let}\ m' = (s', C', \sigma'_{pub} \vee \sigma'_{priv}) \\
&\mathit{include}(\langle\rangle, \sigma) = \{\} \\
&\mathit{include}(\langle a \rangle, \sigma) = \{\} \\
&\mathit{include}(\langle (m', \oplus) \rangle, \sigma) = \\
&\qquad \{(s' \upharpoonright A, C', \sigma \oplus (\sigma'_{pub} \vee \sigma'_{priv}))\} \cup \sigma \oplus_s \mathit{include}(s', \sigma'_{pub}) \\
&\mathit{include}(\langle (m', \oplus) \rangle \frown s, \sigma) = \\
&\qquad \mathit{append}(s \upharpoonright A, \mathit{include}(\langle (m', \oplus) \rangle, \sigma)) \cup \mathit{include}(s, \sigma) \\
&\mathit{include}(\langle a \rangle \frown s, \sigma) = \mathit{prepend}(\langle a \rangle, \mathit{include}(s, \sigma))
\end{aligned}$$

$$\begin{aligned}
&\mathit{append} : \mathrm{seq}\, A \times \mathbb{P}\, M \rightarrow \mathbb{P}\, M \\
&\mathit{append}(s, \mathcal{M}) = \{(s' \frown s, C', \sigma') \mid (s', C', \sigma') \in \mathcal{M}\} \\
&\mathit{prepend} : \mathrm{seq}\, A \times \mathbb{P}\, M \rightarrow \mathbb{P}\, M \\
&\mathit{prepend}(s, \mathcal{M}) = \{(s \frown s', C', \sigma') \mid (s', C', \sigma') \in \mathcal{M}\}
\end{aligned}$$

$$\begin{aligned}
&\oplus_s : \Sigma \times \mathbb{P}\, M \rightarrow \mathbb{P}\, M \\
&\sigma \oplus_s \mathcal{M} = \{(s', C', \sigma \oplus \sigma') \mid (s', C', \sigma') \in \mathcal{M}\}
\end{aligned}$$

Figure 5.5: Definition of *convert*

include invoked on the top-level modules of the open module set using $\top$ as a signature and $\wedge$ as an operator. This computes the basic modules corresponding to the top-level modules without modifying their signatures. The basic modules can then be transformed into (code, signature) pairs using *compile*.

The *include* function defines a set of basic modules that correspond to a set of included modules. Given a module $m = (s, C, \sigma)$ with σ as its public signature, $\mathit{include}(s, \sigma)$ recursively computes the basic modules that correspond to modules included in m, taking into account the effect of the inheritance of the signature σ. Note that it computes the modules corresponding to the children of m and does not include the basic module that corresponds to m itself; hence *convert* is defined as an invocation of the function *include* on a sequence that contains a top-level module m and using a signature and operation that does not affect m.

The definition of *include* uses several auxiliary functions. The functions *prepend* and *append* respectively insert a sequence of aspects before and after the sequence component of a set of modules. These are used to define the effect of the order of inclusion of aspects and modules to the precedence of aspect applications. The operation $\oplus_s$ is an extension of an operator $\oplus : \bigoplus$. It applies a signature σ to all the modules in a set of modules using the operator $\oplus$, where $\oplus$ is either $\wedge$ or $\vee$.

The first two rules that define *include* provide the base cases for an empty sequence and a sequence that contains a single aspect. An empty sequence produces an empty set of basic modules. Since *include* represents the basic modules corresponding to the included modules, a sequence that contains only an aspect also produces an empty set.

The third rule provides the base case for module inclusion. When a module m' is included in a module with a public signature σ using an operator $\oplus$, it corresponds to a basic module $(s' \restriction A, C', \sigma \oplus (\sigma_{pub} \vee \sigma_{priv}))$ that contains the sequence s' of m' filtered to contain only aspects, and with a signature that is σ applied to the signature of m' using the specified operator $\oplus$. Since s' may also contain other modules, the set of basic modules for the sequence s' is also computed, this time using the public signature σ'_{pub} of m'. Since the public signature σ of the parent of m' must also apply to its children, it is applied to the result of $include(s', \sigma'_{pub})$.

The remaining rules provide the step case for sequences. The rule for a sequence $\langle (m', \oplus) \rangle \frown s$ headed by a module m' is to append the rest of the sequence $s \restriction A$ filtered so that it only contains aspects. This models the fact that aspects included in a module also apply to a module's children, and that any aspects that are included after an member module have higher precedence. It then continues to compute the basic modules induced by the rest of the sequence s. The rule for a sequence $\langle a \rangle \frown s$ headed by an aspect is to prepend the aspect to the sequence components of the basic modules induced by the rest of the sequence.

Since the rules that add to the result of the function or modify the sequence component of the resulting modules (rules 3 to 5) always filter the sequence to include only aspects, the result of an *include* contains only basic modules. As a valid open module set does not allow cyclical inclusions, it follows that the result of an *include* on a sequence s is a finite set, as long as s belongs to a module m that is a member of a valid open module set.

5.5 Aspect Order Consistency

AspectJ specifies a global aspect precedence order, that is, if an aspect a_1 is applied before a_2 at a joinpoint, it must be applied before a_2 at all joinpoints in the program. As the order of the inclusion of aspects in a module define a precedence order, it must be proven that the precedence order defined by a set of Open Modules S is consistent with a global aspect precedence. Each of the basic modules in $\mathit{convert}(S)$ specify a sequence of aspects that apply to a particular set of code, thus it must be shown that each element of $\mathit{convert}(S)$ is consistent with some global aspect order.

First, we define aspect order consistency:

Definition (Aspect Order Consistency)

Given two sequences $s, s' : \operatorname{iseq} A$ of aspects such that $\operatorname{ran}(s') \subseteq \operatorname{ran}(s)$, the sequence s' is *consistent* with s if $s \upharpoonright \operatorname{ran}(s') = s'$. Similarly, a basic module $m' = (s', C', \sigma')$ is consistent with s if s' is consistent with s.

A module induces an aspect ordering that affects itself and its child modules. The following definition specifies the aspect order induced by the sequence component of a module.

Definition (Induced Aspect Order)

Given a module $m = (s, C, \sigma)$ that is an element of a valid open module set $S : S_M$, the induced aspect order $\mathit{aspectorder}(s)$ of the sequence component of the module is

$$
\begin{aligned}
&\mathit{aspectorder} : \operatorname{iseq}(M_\oplus \cup A) \rightarrow \operatorname{iseq} A\\
&\mathit{aspectorder}(\langle\rangle) = \langle\rangle\\
&\mathit{aspectorder}(\langle a\rangle) = \langle a\rangle\\
&\mathit{aspectorder}(\langle(m', \oplus)\rangle) = \mathit{aspectorder}(\mathit{getseq}(m'))\\
&\mathit{aspectorder}(\langle(m', \oplus)\rangle \frown s) = \mathit{aspectorder}(\langle(m', \oplus)\rangle) \frown \mathit{aspectorder}(s)\\
&\mathit{aspectorder}(\langle a\rangle \frown s) = \langle a\rangle \frown \mathit{aspectorder}(s)
\end{aligned}
$$

where *getseq* is

$$\begin{aligned}&\mathit{getseq} : M \rightarrow \mathrm{iseq}(M_{\oplus} \cup A)\\ &\mathit{getseq}((s, C, \sigma)) = s\end{aligned}$$

Since m is a member of a valid open module set, it can be deduced that $\mathit{aspectorder}(\mathit{getseq}(m))$ does produce an injective sequence, that is, a sequence that has no recurring elements. This is due to the constraints on open module set that do not allow cyclical inclusions and only allow an aspect to be included in at most one module. The sequence $\mathit{aspectorder}(\mathit{getseq}(m))$ also contains all the aspects that are included in m and its descendants.

Before proceeding to prove that the basic modules induced by an open module set is consistent with the induced aspect order, we first prove a lemma about consistency.

Lemma(Aspect Subsequence Consistency)

Given a module $m = (s, C, \sigma)$ that is a member of a valid open module set S, then $(s \upharpoonright A)$ is consistent with $\mathit{aspectorder}(s)$.

Proof by Induction:
Inductive Hypothesis: Given a module $m = (s, C, \sigma)$ that is a member of a valid open module set S, then $s \upharpoonright A$ is consistent with $\mathit{aspectorder}(s)$.
Base-$\langle\rangle$: Let $s = \langle\rangle$

$$s \upharpoonright A = \langle\rangle = \mathit{aspectorder}(\langle\rangle) = \mathit{aspectorder}(s)$$

Base-$\langle a\rangle$: Let $s = \langle a\rangle$

$$s \upharpoonright A = \langle a\rangle = \mathit{aspectorder}(\langle a\rangle) = \mathit{aspectorder}(s)$$

Base-$\langle(m, \oplus)\rangle$: Let $s = \langle(m, \oplus)\rangle$

$$s \upharpoonright A = \langle\rangle$$

The empty sequence is consistent with any sequence, since for any sequence s'

$$s' \upharpoonright \mathrm{ran}(\langle\rangle) = s' \upharpoonright \varnothing = \langle\rangle$$

Thus

$$s \upharpoonright A = \langle\rangle = \mathit{aspectorder}(s) \upharpoonright \operatorname{ran}(\langle\rangle)$$

Step-$\langle(m', \oplus)\rangle \frown s'$: Let $s = \langle(m', \oplus)\rangle \frown s'$.

$$\begin{array}{l} s \upharpoonright A = (\langle(m', \oplus)\rangle \frown s') \upharpoonright A = s' \upharpoonright A \\ \mathit{aspectorder}(s) = \mathit{aspectorder}(\langle(m', \oplus)\rangle) \frown \mathit{aspectorder}(s') \end{array}$$

Since $m = (s, C, \sigma)$ is a member of a valid open module set S, then the module $m'' = (s', C, \sigma)$ is also a member of some valid open module set (an example would be S with m'' substituted for m). Thus by the inductive hypothesis, $s' \upharpoonright A$ is consistent with $\mathit{aspectorder}(s')$, that is

$$s' \upharpoonright A = \mathit{aspectorder}(s') \upharpoonright \operatorname{ran}(s' \upharpoonright A)$$

Again since $m = (s, C, \sigma)$ is a member of a valid open module set, then $\mathit{aspectorder}(s)$ contains no repeating elements. As

$$\mathit{aspectorder}(s) = \mathit{aspectorder}(\langle(m', \oplus)\rangle) \frown \mathit{aspectorder}(s')$$

then

$$\operatorname{ran}(\mathit{aspectorder}(\langle(m', \oplus)\rangle)) \cap \operatorname{ran}(\mathit{aspectorder}(s')) = \varnothing$$

And since aspects may only appear in at most one module, it is true that

$$\operatorname{ran}(\mathit{aspectorder}(\langle(m', \oplus)\rangle)) \cap \operatorname{ran}(s' \upharpoonright A) = \varnothing$$

Thus

$$\begin{array}{l} \mathit{aspectorder}(s) \upharpoonright \operatorname{ran}(s' \upharpoonright A) \\ \quad = (\mathit{aspectorder}(\langle(m', \oplus)\rangle) \frown \mathit{aspectorder}(s')) \upharpoonright \operatorname{ran}(s' \upharpoonright A) \\ \quad = \mathit{aspectorder}(s') \upharpoonright \operatorname{ran}(s' \upharpoonright A) \\ \quad = s' \upharpoonright A = s \upharpoonright A \end{array}$$

Step-$\langle a\rangle \frown s'$: Let $s = \langle a\rangle \frown s'$.

$$s \upharpoonright A = (\langle a\rangle \frown s') \upharpoonright A = \langle a\rangle \frown (s' \upharpoonright A)$$

Using reasoning similar to that used in the previous case, then by the inductive hypothesis, $s' \upharpoonright A$ is consistent with $\mathit{aspectorder}(s')$. That is

$$s' \upharpoonright A = \mathit{aspectorder}(s') \upharpoonright \mathrm{ran}(s' \upharpoonright A)$$

Prepending $\langle a \rangle$ to both sides maintains the consistency, since

$$\begin{aligned}
&(\langle a \rangle \frown \mathit{aspectorder}(s')) \upharpoonright \mathrm{ran}(\langle a \rangle \frown (s' \upharpoonright A)) \\
&\quad = \langle a \rangle \frown (\mathit{aspectorder}(s') \upharpoonright (\mathrm{ran}(\langle a \rangle \frown (s' \upharpoonright A))) \\
&\quad = \langle a \rangle \frown (\mathit{aspectorder}(s') \upharpoonright \mathrm{ran}(s' \upharpoonright A)) \\
&\quad = \langle a \rangle \frown (s' \upharpoonright A)
\end{aligned}$$

□

We can now prove the main theorem that will be used to show that $\mathit{convert}(S)$ is consistent with some global aspect order.

Theorem (Module Aspect Order Consistency)
Given a module $m = (s, C, \sigma)$ that is a member of a valid open module set S, then

$$\forall\, m' \in \mathit{include}(s, \sigma) \bullet \mathit{getseq}(m') = \mathit{aspectorder}(s) \upharpoonright \mathrm{ran}(\mathit{getseq}(m'))$$

Proof by induction:
Inductive Hypothesis: Given a module $m = (s, C, \sigma)$ that is a member of a valid open module set S, then

$$\forall\, m' \in \mathit{include}(s, \sigma) \bullet \mathit{getseq}(m') = \mathit{aspectorder}(s) \upharpoonright \mathrm{ran}(\mathit{getseq}(m'))$$

Base-$\langle\rangle$: Let $m = (\langle\rangle, C, \sigma)$.

$$\mathit{include}(\langle\rangle, \sigma) = \varnothing$$

Thus the condition is vacuously true.

Base-$\langle a \rangle$: Let $m = (\langle a \rangle, C, \sigma)$.

$$\mathit{include}(\langle a \rangle, \sigma) = \varnothing$$

Again, the condition is vacuously true.

Step-$\langle (m', \oplus) \rangle$: Let $m = (\langle (m', \oplus) \rangle, C, \sigma)$, and $m' = (s', C', \sigma'_{pub} \vee \sigma'_{priv})$

$$\begin{aligned} &\mathit{include}(\langle (m', \oplus) \rangle, \sigma) \\ &\quad = \{(s' \restriction A, C', \sigma \oplus (\sigma'_{priv} \vee \sigma'_{pub}))\} \cup \sigma \oplus_s \mathit{include}(s', \sigma'_{pub}) \end{aligned}$$

$$\mathit{aspectorder}(\langle (m', \oplus) \rangle) = \mathit{aspectorder}(\mathit{getseq}(m')) = \mathit{aspectorder}(s')$$

By the inductive hypothesis, all the elements of $\mathit{include}(s', \sigma'_{pub})$ are consistent with $\mathit{aspectorder}(s')$. Since the operator $\oplus_s$ only changes the signatures of $\mathit{include}(s', \sigma'_{pub})$, then all the members of $\sigma \oplus_s \mathit{include}(s', \sigma'_{pub})$ are also consistent with $\mathit{aspectorder}(s')$.

Finally, by the subsequence consistency lemma, $s' \restriction A$ is consistent with $\mathit{aspectorder}(s')$. Hence all the elements of $\mathit{include}(\langle (m', \oplus) \rangle, \sigma)$ are consistent with $\mathit{aspectorder}(\langle (m', \oplus) \rangle)$.

Step-$\langle (m', \oplus) \rangle \frown s$: Let $m = (\langle (m', \oplus) \rangle \frown s, C, \sigma)$.

$$\begin{aligned} &\mathit{include}(\langle (m', \oplus) \rangle \frown s, \sigma) \\ &\quad = \mathit{append}(s \restriction A, \mathit{include}(\langle (m', \oplus) \rangle, \sigma)) \cup \mathit{include}(s, \sigma) \end{aligned}$$

$$\mathit{aspectorder}(\langle (m', \oplus) \rangle \frown s) = \mathit{aspectorder}(\langle (m', \oplus) \rangle) \frown \mathit{aspectorder}(s)$$

By the inductive hypothesis, all the elements of $\mathit{include}(\langle (m', \oplus) \rangle, \sigma)$ are consistent with $\mathit{aspectorder}(\langle (m', \oplus) \rangle)$, and all the elements of $\mathit{include}(s, \sigma)$ are consistent with $\mathit{aspectorder}(s)$.

It need to be shown that all the elements of the set

$$s_a = \mathit{append}(s \restriction A, \mathit{include}(\langle (m', \oplus) \rangle, \sigma))$$

are consistent with $\mathit{aspectorder}(\langle (m', \oplus) \rangle \frown s)$. A element m_a of s_a is of the form $(s'' \frown (s \restriction A), C'', \sigma'')$ where $(s'', C'', \sigma'') \in \mathit{include}(\langle (m', \oplus) \rangle, \sigma)$. By the subsequence consistency lemma, $s \restriction A$ is consistent with $\mathit{aspectorder}(s)$. All the elements of $\mathit{include}(\langle (m', \oplus) \rangle, \sigma)$ are consistent with $\mathit{aspectorder}(\langle (m', \oplus) \rangle)$, therefore s'' is consistent with $\mathit{aspectorder}(\langle (m', \oplus) \rangle)$. Therefore $s'' \frown (s \restriction A)$ is consistent with $\mathit{aspectorder}(\langle (m', \oplus) \rangle) \frown \mathit{aspectorder}(s)$, and so is m_a.

The module m belongs to a valid open module set, which allows an aspect or module to be included in at most one module, and does not allow cyclical or multiple module inclusions. It then follows that the set of aspects contained in the modules of $\mathit{include}(s, \sigma)$ is disjoint from the set of aspects in $\mathit{aspectorder}(\langle m', \oplus\rangle)$. As all the elements of $\mathit{include}(s, \sigma)$ are consistent with $\mathit{aspectorder}(s)$, then they are also consistent with $\mathit{aspectorder}(\langle m', \oplus\rangle)^\frown$ $\mathit{aspectorder}(s)$.

Since all the elements of $\mathit{append}(s \upharpoonright A, \mathit{include}(\langle(m', \oplus)\rangle, \sigma))$ and $\mathit{include}(s, \sigma)$ are consistent with $\mathit{aspectorder}(\langle(m', \oplus)\rangle \frown s)$, then all the elements of $\mathit{include}(\langle(m', \oplus)\rangle, \sigma)$ are consistent with $\mathit{aspectorder}(\langle(m', \oplus)\rangle \frown s)$.

Step-$\langle a\rangle \frown s$: Let $m = (\langle a\rangle \frown s, C, \sigma)$.

$$\begin{aligned}&\mathit{include}(\langle a\rangle \frown s, C, \sigma) = \mathit{prepend}(\langle a\rangle, \mathit{include}(s, \sigma))\\ &\qquad = \{(\langle a\rangle \frown s', C', \sigma') \mid (s', C', \sigma') \in \mathit{include}(s, \sigma)\}\end{aligned}$$

$$\mathit{aspectorder}(\langle a\rangle \frown s) = \langle a\rangle \frown \mathit{aspectorder}(s)$$

By the inductive hypothesis, all the elements of $\mathit{include}(s, \sigma)$ are consistent with $\mathit{aspectorder}(s)$. Prepending $\langle a\rangle$ to the sequence component of every element of $\mathit{include}(s, \sigma)$ as well as to $\mathit{aspectorder}(s)$ maintains the consistency.

□

Finally, it can now be proved that all the elements of $\mathit{convert}(S)$ are consistent with a global aspect order.

Theorem (Convert Aspect Order Consistency)

Given a valid module set S, then all the elements of $\mathit{convert}(S)$ are consistent with

$$\begin{aligned}&\mathit{aspectorder}(\langle(m_1, \wedge)\rangle) \frown \mathit{aspectorder}(\langle(m_2, \wedge)\rangle) \frown ...\\ &\qquad \mathit{aspectorder}(\langle(m_n, \wedge)\rangle)\end{aligned}$$

where

$$\mathit{toplevel}(S) = \{m_1, m_2, ..., m_n\}$$

Proof

$$\mathit{convert}(S) = \bigcup_{m \in \mathit{toplevel}(s)} \mathit{include}(\langle(m, \wedge)\rangle, \top)$$

Define a set of modules m'_1 to m'_n such that $m'_1 = (\langle m_1, \wedge\rangle, \varnothing, \top)$, $m'_2 = (\langle m_1, \wedge\rangle, \varnothing, \top)$ and so on up to m'_n. The set $S \cup \{m'_1, m'_2, ..., m'_n\}$ is also a valid open module set, which allows us to use the module aspect order consistency theorem.

By the module aspect order consistency theorem, all the elements of $\mathit{include}(\langle(m_x, \wedge)\rangle, \top)$ are consistent with $\mathit{aspectorder}(\langle(m_x, \wedge)\rangle)$, where $m_x \in \{m_1...m_n\}$. Since S is an valid open module set, and the modules m_1 to m_n are top-level modules, the sequences $\mathit{aspectorder}(\langle(m_1, \wedge)\rangle)$ to $\mathit{aspectorder}(\langle(m_n, \wedge)\rangle)$ are disjoint, that is, they do not have any common elements in their ranges. Therefore all the elements of the sets $\mathit{include}(\langle(m_x, \wedge)\rangle, \top), m_x \in \mathit{toplevel}(S)$ are consistent with

$$\mathit{aspectorder}(\langle(m_1, \wedge)\rangle) \frown \mathit{aspectorder}(\langle(m_2, \wedge)\rangle) \frown ... \\ \mathit{aspectorder}(\langle(m_n, \wedge)\rangle)$$

and so is $\mathit{convert}(S)$.

$\square$

Note that the order of the concatenations that define the global aspect order in the previous theorem does not matter. This reflects the fact that each top-level module and its descendants define an aspect precedence order that is disjoint from the other top level modules.

Chapter 6

Conclusion

6.1 Summary

The dissertation has presented an implementation of Open Modules in AspectJ as an extension of the AspectBench compiler. The syntax of the conversion of Open Modules from the original *TinyAspect* language to *AspectJ* was defined, as well as its corresponding effect on the behavior of pointcut matching.

The design also introduced module inclusion, which behaves similarly to module nesting as defined in the original Open Modules, but adds additional features such as multiple inclusion modes (constrained and non-constrained) as well as private signatures.

The details of the implementation of Open Modules as an extension of the AspectBench compiler was also described. The relative ease in which most of the extension was integrated highlights the extensibility of the AspectBench compiler. However, several problems that came up during the implementation show that the extensibility is not complete, and provide starting points for further improvement.

The semantics of the open module implementation was also described formally, and was shown to transform a set of modules into (code, signature) pairs without violating the global aspect precedence rule of AspectJ.

6.2 Future work

Open Modules provide an interface that only restricts advice applications. AspectJ has other features that may affect the ability to evolve code, such as intertype declarations and *declare parents*. The analysis of the effect of these features on modularity and the extension of Open Modules to provide means to constrain these features is a logical progression from this dissertation.

The semantics defined for Open Modules in this dissertation only provide a transformation from modules to (code, signature) pairs. Although it is believed that this transformation simulates the behavior of Open Modules, it would be desirable to devise a formal model to prove the equivalence of the semantics of the implementation to Open Modules as defined for *TinyAspect*.

AspectJ also has pointcut primitives other than *call*, such as *execution*, *cflow* and *if*, which were not considered in the original definition of Open Modules. This would require a extension of the formal definition of Open Modules to include these primitives.

The implementation of Open Modules in AspectJ allows an evaluation of its effectiveness in actual software. Such an evaluation may expose problems in Open Modules, and provide starting points for its improvement or the design of alternative modularity mechanisms for aspect-oriented programs. The existence of an implementation also allows for the design and development of tools to integrate Open Modules in the development process of AspectJ programs.

Bibliography

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

[2] Jonathan Aldrich. Open modules: A proposal for modular reasoning in aspect-oriented programming. Technical Report CMU-ISRI-04-108, Institute for Software Research, Carnegie Mellon University, 2004.

[3] C. Clifton and G. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning. In *Proceedings of the workshop on the Foundations of Aspect-Oriented Languages (FOAL '02)*, 2002.

[4] G. Kiczales. Aspect-oriented programming. *ACM Computing Surveys*, 28A(4), 1996.

[5] 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.

[6] Ramnivas Laddad. *AspectJ in Action*. Manning, 2003.

[7] K. Lieberherr, D. Lorenz, and J. Ovlinger. Aspectual collaborations: Combining modules and aspects. *The Computer Journal*, 46(5):542–565, 2003.

[8] Nathan McEachen and Roger T. Alexander. Distributing classes with woven concerns: an exploration of potential fault scenarios. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 192–200, New York, NY, USA, 2005. ACM Press.

[9] 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.

[10] The AspectJ Team. The AspectJ Programming Guide.

[11] 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.

[12] J C P Woodcock and J Davies. *Using Z.* Prentice Hall, 1996.