ParaAJ: toward Reusable and Maintainable Aspect Oriented Programs

Khalid Aljasser **Peter Schachte**

The University of Melbourne, Australia
{aljasser,schachte}@csse.unimelb.edu.au

Abstract

Aspect Oriented Programming (AOP) aims to ease maintenance and promote reuse of software components by separating core concerns from crosscutting concerns: aspects of a program that cannot be confined to a single program component. In AOP languages such as AspectJ, this kind of concern is encapsulated in an *aspect* and connected to main classes using pointcuts. This removes extraneous code from the classes of the program, allowing them to focus on their core purpose and making them more maintainable and reusable. The implementation of each crosscutting concern, which would have been scattered throughout an object oriented program, is centralised in a single aspect. Unfortunately, due to the way aspects and classes are associated, and the lack of an explicit *interface* between them, these aspects may be tightly coupled to the classes of the program and so may not be as reusable or maintainable as might be expected. We propose ParaAJ (*Parametric Aspects*), as an extension to AspectJ. ParaAJ allows classes to specify which aspects should be applied, and allows applications to specify which aspects to apply to which classes in what ways. This makes it easier for classes and aspects to be developed and maintained independently, and encourages reuse of both.

Keywords: Aspect Oriented Programming, Reusability, Maintenance

1 Introduction

Maintenance is a central part of the software lifecycle, commonly identified as accounting for more than half of the cost of software development. Maintainability can be determined by several factors but probably the greatest single factor in determining how maintainable a program will be is how well structured it is, that is, how well it is organised into separate modules that can be independently developed and maintained. Each concern of the program should be localised in a single module, so that if the requirements relating to that concern change, only that module need be modified. Similarly, each module should address only one concern, so that the developer of that module can focus exclusively on that concern. This separation of concerns is also important to software reusability, as components that address all of and only one concern can generally be used in more contexts than one that combines multiple concerns.

In practise, this separation of concerns is very difficult to achieve, especially with the presence of crosscutting concerns: aspects of a program that cannot be confined to a single program component. Aspect Oriented Programming (AOP) (Kiczales et al. 1997) has been suggested as a solution to the problem of concerns that cut across multiple modules in an application. AOP allows the code in a single-concern component to be *advised*, changing its behaviour without actually modifying the code. Consider, for example, the following Java code for a simple bank account class.

Listing 1: A bank account class

```
public class BankAccount {
    private double balance = 0.0;
    public void deposit(double d) {balance += d;}
    public void withdraw(double w) {balance -= w;}
    public double getBalance() {return balance;}
}
```

Suppose we wish to add a new feature (concern) to our program, so that a graphical user interface is updated as the account balance changes. In Java, several techniques can be used to accomplish such maintenance tasks such as direct-modification, subclassing, and decorator design pattern. Using direct-modification, we could insert the monitoring code into the `BankAccount` class but, in this case, we are left without an ordinary, unmonitored bank account class. Using subclassing, we create a `MonitoredBankAccount` subclass that overrides every method to insert the monitoring logic. This approach does not scale, as each crosscutting concern we add to the bank account logic doubles the number of variations on the original `BankAccount` class, with the attendant code duplication — a nightmare for maintenance. More detailed discussion about the various options to handle such requirements is shown in our previous work (Al-Jasser et al. 2007).

Aspect oriented programming languages allow the monitoring logic to be placed in a separate program component, called an *aspect*. The following AspectJ (Kiczales et al. 1997, 2001) code adds the monitoring logic to the `BankAccount` class without changing the `BankAccount` code: With this aspect included in the application, all `BankAccount` objects will have the required monitoring logic; omitting the aspect leaves the original `BankAccount` logic unchanged. Furthermore, many aspects like this can be independently applied to a single class. This

Listing 2: Monitoring aspect in AspectJ

```
public aspect BankAccountMonitoring {
  pointcut UpdatingMethodsPC():
    execution(void BankAccount.deposit(double))  ||
    execution(boolean BankAccount.withdraw(double));
  after(): UpdatingMethodsPC(){
    // code to be executed after balance changes
  }
}
```

makes existing classes much more reusable, as their behaviour can be modified without needing to modify their code.

However, there are some limitations to current aspect oriented languages for software maintainability and reuse, which we address in this paper. One important limitation stems from the relationship between the class and the aspect. Filman and Friedman Filman & Friedman (2000) define AOP as "the desire to make quantified statements about the behavior of programs, and to have these quantifications hold over programs written by oblivious programmers." That is, aspects should be able to make sweeping changes to the behaviour of many parts of an application, without the knowledge of the developers of the classes and methods whose behaviour is affected. However, this quantification and obliviousness are rather one-sided: the developers of a method may be oblivious to the aspects that affect its behaviour, but the developer of an aspect must not be oblivious to the methods it affects (lest it affect some methods in unwanted ways). Likewise, the developer of the aspect may quantify over various features of the modules of the program, but the developers of those modules cannot influence which parts of which modules are quantified over, since they are oblivious to the quantification. This asymmetry stands in stark contrast to traditional information hiding used in other programming paradigms, where the developer of a calling module is oblivious to the implementation of the callee, and the developer of the callee is oblivious to the uses made by the caller (as long as they can adhere to the specified interface).

The asymmetry of the association between aspects and the classes was noted by Kersten & Murphy (1999), who identified four types of class $\leftrightarrow$ aspect association. In an *open* association, the aspect and class are both written with explicit knowledge of each other. In a *class-directional* association, the aspect is written with knowledge of the class, but the class is oblivious to the aspect. An *aspect-directional* association allows the class to refer to the aspect, but not the reverse. Finally, in a *closed* association, the class and the aspect are oblivious to each other. Kersten and Murphy identified open associations as undesirable because they compromise understandability and reusability, while closed associations would provide more reusable and understandable classes and aspects but were not supported by the version of the AspectJ language they were using; aspect-directional associations were also not supported by AspectJ, and class-directional associations were the approach they recommended. They also suggested that class-directional associations make classes more reusable, while aspect-directional associations make aspects more reusable, and closed associations make both reusable.

No less important than the direction of link or association between class and aspect is the degree of coupling between them. In our example, the `BankAccountMonitoring` aspect is explicitly tied to the `BankAccount` class, and so cannot be reused when we find other classes that need to be monitored. AspectJ supports abstract aspects, which allows some parts of an aspect that are independent of a specific class to be abstracted and reused by many concrete aspects. Hanenberg & Unland (2003*b*), however, point out that abstract aspects are hampered by the need to put any part of the aspect containing information specific to a particular use, such as the classes involved, in the concrete aspect. In some cases, this means little or none of the aspect can be made abstract, and therefore little reuse is possible. In this example, the need to explicitly specify the `BankAccount` class and the two methods means none of this aspect can be made abstract except the action (advice). Hanenberg and Unland give several examples of common design patterns that cannot be captured as abstract aspects in AspectJ, and propose an approach they call *parametric introductions* which allows these design patterns to be reused. Parametric introductions permit parts of an aspect, particularly types, to be abstracted out as parameters, and specified where they are used. However, this approach is still exclusively class-directional.

In this paper we argue that aspect-directional and closed associations are preferable to class-directional and open associations. Aspect-directional associations have the virtue that they are visible to the developer (who is *not* oblivious to them). In cases where the aspect is essential to the proper functioning of the class, then an aspect-directional association is the most desirable sort. For example, the developer might feel that a bank account that does not synchronise modifications of the balance would not be very useful, so the fact that updates are synchronised should be visible in the `BankAccount` class itself. Yet the synchronisation itself is a separate concern of the class that can and should be encapsulated in a separate module. Monitoring, on the other hand, is not essential to the proper functioning of a bank account, so a closed association would be more appropriate.

We also argue that the interface between class and aspect should be made explicit, through the use of parameters. This encourages the design of reusable aspects, as well as classes.

The rest of this paper is organised as follows. Section 2 gives some necessary background needed in this paper. Section 3 discusses the diversity of crosscutting concerns and the effect of this on dealing with them. Section 4 explains our approach, *parametric aspects*, and discusses the ParaAJ language that embodies it. Section 5 follows with an overview of our approach to implementation. Finally Sections 6 and 7 discuss related work and present our conclusions.

2 Background and Motivation

Aspect oriented software development focuses on modularizing concerns that cannot be modularized properly in traditional programming languages. Such concerns tend to tangle the main logic of programs with other logic that is, in most cases, scattered across the code of that main module; thus they are referred as *crosscutting concerns*. AspectJ (Kiczales et al. 1997, 2001), one of the best known aspect oriented languages, extends Java with *aspects* as containers for

such crosscutting concerns. *Weaving*, a pre-compile step, glues aspect code with its target classes. The advantage of this approach is that it allows developers to modularize their crosscutting concerns and easily get the crosscutting effect on their systems.

Aspects and classes are connected using the *pointcut* construct which, in most cases, is defined inside aspects. A pointcut selects one or more *join-point(s)* in target classes. A join-point is simply any point of execution in the program whose behaviour can be modified by an aspect. In AspectJ, all of the following are considered join-points: method call, method execution, object construction, field accessing, field updating, and exception handling. These kinds of connections are called *class-directional association* (Kersten & Murphy 1999) and can effectively disentangle crosscutting logic from main logic resulting in clean and reusable classes. However, this makes crosscutting concerns *invisible* to target classes. In other words, target classes are *oblivious* of aspects that modify their behavior. The obliviousness of target classes is one of the design decisions when AOP was introduced (Filman & Friedman 2000). Although it proved to be useful in many cases, it has an impact on the maintainability of overall system. For example, in our `Bank Account` example in Listing 1, even simple maintenance scenarios (e.g. adding a `creditInterest` method) requires modifying the `BankAccount` class and the `Monitoring` aspect: adding the new method to the class and modifying the pointcut to capture this method (see line 2 in Listing 2).

Another problem with class-directional relationships is the fact that aspects are holding too much information: the crosscutting logic and target class information (see lines 3 and 4 in Listing 2). This presents a significant problem of tight coupling between the aspect and its target (i.e. `BankAccount` class). Myers (1978) distinguishes five levels of coupling; this example exhibits control coupling, Myers' middle level. This level of coupling results from the lack of an agreed interface between the class and the aspect. If an interface could be specified, any change to either the class or the aspect that preserved the interface would not require the other to be modified.

Software reuse facilitates and helps in increasing productivity and quality of software components. AOP languages help in this regard by purifying core modules from crosscutting concerns and thus making the former reusable. Beside reusing core concerns, it would be very useful to reuse crosscutting concerns as well. Wampler (2006) stated that aspects are difficult to reuse because of the relative immaturity of aspect design and available programming techniques. AspectJ, for example, provides *abstract aspects* as a way to reuse aspects by specifying the behaviour of the aspect and leaving the definition of the pointcut to the concrete sub-aspects. Although this approach is sufficient in many cases, a better approach would be to specify a clear interface of aspects allowing them to be reused in other applications. For example, imagine a synchronization aspect that insures that certain methods of a class are executed to completion before another execution thread begins executing one of those methods on the same object. In AspectJ, we can implement synchronization as the aspect shown in Listing 3.

Adding the synchronization logic to the class requires introducing a `lock` member to it as shown in

Listing 3: Synchronization aspect

```
1 public aspect BankAccountSynchronization {
2  int BankAccount.lock;
3  pointcut TransactionPC():
4   execution(void BankAccount.deposit(double) ) ||
5   execution(boolean BankAccount.withdraw(double));
6  before():  TransactionPC(){
7      // Acquiring the lock
8  }//end before
9  after():   TransactionPC(){
10     //release the lock and notify waiting threads
11 }//end after
12 }
```

line 2. Trying to make this aspect abstract requires removing any reference to the BankAccount class (including the introduced member). Clearly, since this lock is used to achieve the synchronization logic in the advice (lines 7 and 10), we will not be able to remove it and hence we cannot make this aspect abstract.

It is worth noting that AspectJ (version 5) makes good use of the Metadata feature introduced in J2SE 5.0. Metadata allows programmers to add additional information to program elements (e.g. classes and methods) which can later be used to enhance the documentation of the class or even perform compile-time tasks. AspectJ uses metadata to simplify the definition of pointcuts by relying on the metadata tag rather than the name of the join point (e.g. class, method, member, etc.).

Listing 4: Using metadata in the bank Account Class

```
public class BankAccount{
  ...
  @Transaction
  public void deposit(double amount){ ... }

  @Transaction
  public void withdraw(double amount){ ... }
}
```

```
public aspect BankAccountAspect {
    pointcut  TransactionPC(BankAccount account,
                            double amount):
    execution(
      @Transaction  void BankAccount.*( double )
    )
    &&  target(account) &&  args(amount);
    ...
}
```

Using metadata, we shift some of the pointcut definition to the main class. For example, the `deposit()` and `withdraw()` methods in the `BankAccount` class can be *annotated* or *tagged* with a common annotation which can be used later in the aspect. Listing 4 shows the `BankAccount` class with annotations and how these annotations can be used in the aspect. This is an important step toward defining an interface between an aspect and its target classes. The first advantage of this approach is that even if we want to change the name of a certain method in the `BankAccount` class, we do not need to change the pointcut definition in the aspect (unless we change the arguments, return type, or modifiers). Furthermore, if we want to add a new `creditInterest` method, then we only need to tag it with the `@Transaction` annotation to ensure the aspect logic continues to work correctly. However, this approach would not allow different groups of methods (probably in different classes) to be synchronized separately. We will see in section 4 how our approach makes aspects more

reusable and allows them to be used multiple times in different ways in a single system.

3 Kinds of crosscutting concerns

Difficulties in maintaining and reusing aspects arise from the inherent nature of aspects themselves (Wampler 2006). Crosscutting concerns differ in their purpose and their effect on their target systems. For example, some concerns provide functionality that is essential to the intended functionality of the target module. On the other hand, others provide functionality that, while important to the overall application, are supplementary to the logic of the target module (*e.g.* logging). Some concerns augment the logic of the main concern (*e.g.* synchronisation, where the normal behaviour is only delayed), while others may change the execution path (*e.g.* authentication, where the normal behaviour may be prevented). Knowing the purpose and effect of different crosscutting concerns helps to understand and reason about them. Moreover, it helps in choosing or designing an appropriate solution for each situation.

Rinard et al. (2004) classify crosscutting concerns based on the interaction between advice and methods. According to their classification, aspects are: 1) augmenting, 2) narrowing, 3) replacing, or 4) a combination.

Augmenting aspects allow the joinpoint to execute, but also add an extra logic to it. Narrowing aspects usually add some conditional constraints to allow or completely disallow the execution of the joinpoint. Replacing aspects, as the name suggests, entirely replace the behaviour of the joinpoint with a new behaviour. We find this classification very useful in discussing and reasoning about different types of crosscutting concerns. In our view, augmenting aspects are less likely to lead to undesired interactions with other concerns within the application than narrowing aspects, which in turn are safer than replacing aspects.

Many useful aspect-oriented design patterns and idioms (Hanenberg & Unland 2003*a*, Hanenberg & Schmidmeier 2003) have been developed to handle this variety of crosscutting styles. However, design patterns and idioms do not always result in straightforward solutions. For example, the container introduction idiom (Hanenberg & Unland 2003*a*) allows an aspect to transparently introduce new members to its targets without prior knowledge of these targets. This idiom, though, requires the creation of a container interface, an aspect that adds the required members to this interface, and another aspect to let the target class implement the container interface using the `declare parents` declaration.

Likewise consider the synchronisation concern discussed in the previous section. Synchronisation is essential to the operation of the bank account in multithreaded environment, where multiple processes can access and modify an object simultaneously. Implementing synchronisation as an aspect has two disadvantages. First, the synchronised class is totally oblivious to the synchronisation aspect. This makes it difficult to know what changes to the synchronised class will lead to undesired behaviour. Second, it is difficult to write a reusable synchronisation aspect, even with the use of Java5 annotations as discussed before. We believe the ideal synchronisation solution would achieve the following:

- It would remove the scattering of the crosscutting logic from the base code.
- It would maintain an indicator of the presence of the crosscutting logic in any parts of the base code that depend on that crosscutting logic for their correct behaviour.
- It would provide a reusable implementation of the crosscutting logic.

We believe that although design patterns and idioms are very helpful in solving many programming problems, it would be preferable to have programming constructs that simplify them or make them unnecessary. This paper proposes some constructs that we believe do this. Please note, however, that these come as an extension to the AspectJ language, not a replacement. While we argue that the new constructs we propose are preferable to AspectJ facilities for many problems, in other cases AspectJ's facilites would be more appropriate.

4 The ParaAJ language

In order to overcome the maintainability and reusability limitations of AspectJ, we propose ParaAJ. ParaAJ is an extension to the AspectJ language that allows the creation of *component* aspects, *i.e.,* modular aspects that can be applied and used differently according to the supplied parameters. ParaAJ allows aspect-directional and closed associations, in addition to the class-directional associations it inherits from AspectJ. Its main distinguishing features are that aspects must be explicitly applied to the classes they affect, and that aspect applications may specify parameters that indicate how the aspect is to be applied. These features enhance maintainability, as the aspect parameters create a formal interface between aspects and classes, allowing each to be developed and maintained separately, and encouraging reuse of both aspects and classes.

Although the aspect-directional association used in our approach seems to be violating the spirit of AOP *i.e.,* making target modules oblivious of associated aspects (Filman & Friedman 2000), it does this in a systematic way. First, although a reference to the aspect is present in the target class, the code scattering problem is still avoided. Second, applying the aspect from within the class makes modular reasoning in AO programs easier. Seiter (2008) states that "obliviousness leads to programs that are difficult to develop, understand, and maintain". Seiter (2008), Clifton & Leavens (2003) also state that obliviousness makes it difficult for a developer to understand the effect of maintaining a class or aspect on the overall system.

We will illustrate how the language features combine to ease a number of software development, reuse, and maintenance tasks. Space limitations prevent us from presenting the language in full; we sketch here only its main features.

4.1 Parametric Aspects: A definition

A parametric aspect (para-aspect) is a kind of abstract aspect that provides an explicit *interface* between itself and its clients. It defines a self-contained piece of reusable functionality. A parametric aspect is considered abstract because it cannot be used as is;

it has to be *applied* to a target component and each *application* determines a separate concrete aspect.

A para-aspect can be declared with the following syntax:

⟨***visibility***⟩ **aspect** ⟨***name***⟩**(**⟨***formals***⟩**)** **{**
⟨***aspect body declarations***⟩
}

where ⟨*visibility*⟩ is a visibility modifier as can be applied to a normal aspect; ⟨*name*⟩ is an aspect name; ⟨*formals*⟩ specifies the formal parameters of the aspect, discussed in the next subsection; and ⟨*aspect body declarations*⟩ are any declarations that can appear in a normal AspectJ aspect, with slight differences described below. For example, the following aspect prints a message whenever a certain field in the target class is changed:

```
public aspect LogUpdates(field f) {
    after(): set($[f]){
        System.out.println($[f] + " has changed" );
    }
}
```

Note that in AspectJ, `set(field_pattern)` is a pointcut that captures any statement modifying the field matched by `field_pattern`. A `field_pattern` consists of modifier (optional), field type, host, and field name. For example in plain AspectJ we could write:

`set(` `int` `Foo` `.` `bar` `)`
type *host* *name*
fieldpattern

In ParaAJ, we need not, and must not, specify the host class, as that is determined by the class to which our aspect is applied. In this example, `$[f]` specifies the field whose changes should be announced. The actual name of the field will be passed in the `apply` declaration, discussed in detail below. As a simple example, we might apply the `LogUpdates` aspect to a class as follows:

```
public class MyClass{
    int x;
    apply  LogUpdates(x);
    ...
}
```

in this example, we only pass the field name. Other information (*i.e.,* type and host) are inferred from the context of the apply declaration.

As we can see from the example above, the main differences between normal aspects (as in AspectJ) and para-aspects are:

- *Aspect parameters* define an interface between an aspect and its clients.
- *Meta-symbols*, like (`$[`⟨*symbol*⟩`]`) where *symbol* is one of the aspect's formal parameters, may be used inside the body of the aspect. During compilation, meta-symbols are replaced with the corresponding actual parameter values.
- *Scope of effects*: para-aspects only affect the behavior of the classes they are explicitly applied to.

4.2 Aspect parameters

Para-aspect parameters consist of a sequence of zero or more formal parameter declarations, separated by commas. They are used verbatim in the body of the para-aspect and are evaluated at *compile-time*. That is, they do not specify *values*, but *names*.

Similar to normal parameters in constructors and methods, each formal parameter declaration consists of a meta-type and a name. The name can be any normal Java identifier. The meta-type can be any primitive Java type (*e.g.,* int, boolean, *etc.*), or `String`, or one of the following:[1]

- `ident`: any Java identifier
- `type` : a primitive type, class name, or interface name
- `field`: a field in the target class
- `method`: a method in the target class
- `pointcut`: any valid pointcut

Inside the para-aspect body, the parameter names should appear between '`$[`' and '`]`' (*e.g.,* `$[i]`), where the actual value specified in the **apply** declaration (discussed later) will be substituted. This permits different applications of the same aspect to supply different declarations. Thus it is quite sensible for the same aspect to be applied to the same class multiple times, with different parameters.

Aspect parameters can form part or full statements/declarations in the aspect body. For example, a field declaration (in its simplest form) can be defined in this way:

```
type  name;
```

In ParaAJ, we can replace either or both of the type and the name of the field with *meta* values:

```
$[t] $[n];
```

In a similar way, we can do the same thing for all other Java and AspectJ rules.

We plan to generalize the way we deal with parameters so that we can extract other information from them. For example, if a parameter *f* is of meta-type `field`, beside using it normally as `$[f]`, we will allow it to be referred to as: `$[f.type()]` or `$[f.toString()]`.

4.3 Internal vs. External application of para-aspects

Like a class declaration, a para-aspect declaration is usually written in a file by itself. Unlike a class, however, it defines only part of a type, and as such cannot be instantiated on its own. Instead it has to be *applied* to a class, adding the aspect's functionality to that class. Any number of aspects may be applied to a class, and an aspect may be applied any number of times to different targets (or the same target). Parametric aspects can be applied to a target class *internally* resulting in an aspect-directional relationship, or *externally* (from another class) resulting in a closed association between the aspect and the class. Both cases leave the para-aspect reusable and the second case allows the target class to be reused with or without the application of that aspect. Whether we

[1] We may expand this list.

Listing 5: Synchronization aspect in ParaAJ

```
public aspect Synchronization(methods ms){
  int lock = 0;
  before(): execution($[ms]){
    // acquiring the lock
  }//end before
  after(): execution($[ms]){
    //release the lock and notify waiting threads
  }//end after
  . . .
}
```

should apply an aspect internally or externally depends mainly on the aspect and on the concern it is implementing.

A synchronization aspect, for example, modifies the behaviour of its target class by ensuring that some critical regions in the code are not executed by multiple threads concurrently. A developer may feel, for example, that a bank account class that is not synchronized in a multi-threaded environment is faulty, and therefore wish the bank account class itself to specify that it is to be synchronised, maintaining a single point of control over the core bank account functionality in the source file for that class. In this case, *internal* aspect application is most appropriate.

A monitoring aspect, on the other hand, is a good example where we need an external application of the aspect. Monitoring might involve the class to be monitored and also the class where data will be displayed (*e.g.,* a GUI class). Monitoring, however, is not a core part of the bank account's functionality; it is only certain applications that may need the bank account to be monitored. However, it may be important to a bank application that displays account information that updates to an account balance should be reflected in real time. In this case, it is most appropriate that the code that displays the account details should apply the monitoring aspect to the bank account class, allowing it to be monitored by the GUI. Again, this maintains a single point of control over the monitoring logic in the class that is responsible for that functionality.

4.4 Using para-aspects: the apply declaration

Internal application is specified by inserting an `apply` declaration in the target class anywhere class members can be declared:

apply ⟨*name*⟩(⟨*actuals*⟩);

where ⟨*name*⟩ is a para-aspect name; ⟨*actuals*⟩ specifies the actual parameters that will be passed to the aspect. Internal applications affect only the class in which they are applied.

External application, on the other hand, is specified by inserting an `apply` declaration in one class (the *host*) that specifies that the aspect is to be applied to a second class (the *target*). The syntax of this form of `apply` declaration is:

⟨*target*⟩ **. apply** ⟨*name*⟩(⟨*actuals*⟩);

Each `apply` declaration effectively creates a separate instance of the aspect, whose body is the result of replacing the formal aspect parameters appearing in the body of the para-aspect with the actual parameters specified in the `apply` declaration. Any instance variables declared in the para-aspect behave as members of the target class in that each instance of the target class has that data stored in it. These instance variables, however, are normally private to the aspect and cannot be directly referenced by the members of the host or target class (see Section 4.7 below). They may only be accessed from the methods and advice in the para-aspect. Similarly for class variables and methods of the para-aspect. Further, each `apply` declaration supplies *separate* copies of the para-aspect members to the target class only visible to the advice and methods of that application of the aspect. This permits a para-aspect to be applied to the same class multiple times without having its members conflict with one another. In the reverse direction, the methods and advice of a para-aspect may not access private members classes (except for the host class, and then only if the members are passed as arguments to the `apply` declaration. This enforces information hiding between classes and para-aspects.

Since the `apply` declaration is a compile-time directive, all actual values have to be either *literal* values (*e.g.,* integers, strings, booleans, *etc.*) or names (*e.g.,* field, method, class, or type names). They cannot be general expressions.

In the body of a para-aspect, we might need to refer to the host or target class (for internal application, these will be the same). Therefore, inside a para-aspect, `thisTarget` may be used to specify the class to which the aspect is applied, and `thisHost` refers to the class in which the apply statement appears.

Referring to the monitoring concern we discussed before, we can implement it as a para-aspect as shown in Listing 6 below.

Listing 6: Monitoring aspect in ParaAJ

```
1 aspect Monitoring (field f, method m){
2   thisHost monitor = null;
3   inject void setMonitor(thisHost monitor) {
4     this.monitor = monitor;
5   }
6   after(thisTarget subject): set($[f]) &&
7                              target(subject);
8     if (monitor != null) {
9       monitor.$[m]( subject );
10    }
11  }
12 }
```

The monitoring aspect takes two parameters: a field to be monitored and a method that will be executed when the field changes. In line 2, we create an object of type `thisHost` (the class where the `apply` declaration appears). In lines 3-5, we inject the `setMonitor` method into the target class so that at runtime we can associate the *monitored* class with the actual *monitor*. The `inject` statement is discussed in more details in section 4.7. In line 6, the `set` pointcut takes the field parameter. The method `m` in the `after` advice is a one-parameter method that takes an object of the monitored class. The target class will be implicitly passed to the aspect in the `thisTarget` variable and can be used throughout the aspect.

The next step is to apply this aspect to the `BankAccount` class from the class that needs the monitoring logic as shown in Listing 7. The advantages of this approach are: 1) monitoring aspect is general and can be used and reused in other applications easily; 2) the monitored class (*i.e.,* `BankAccount`) is still separated from the crosscutting logic; and 3) the class responsible for the monitoring logic clearly specifies

Listing 7: Using the monitoring aspect

```
public class Screen {
  BankAccount.apply Monitoring(BankAccount.balance,
                               Screen.display);
  public static void main(String[] args){
        BankAccount ba = new BankAccount(...);
        ba.setMonitor(this);
  }

  public void display (BankAccount ba) {
          // required monitoring action
  }
}
```

what to monitor and how to monitor it, thus maintaining a single point of responsibility.

In some cases, it may be useful to specify multiple fields, methods, *etc.* in an `apply` declaration where the aspect definition only allows for one. This is accomplished by enclosing the multiple values in square brackets, separated by commas. The ParaAJ compiler handles this by acting as if each element of the applied aspect is repeated once for each combination of actual values specified for the formal parameters mentioned in that element.

For example, we can apply the synchronization aspect (Listing 5) to the `BankAccount` class (Listing 1) *internally*:

```
public class BankAccount {
   apply Synchronization([deposit,withdraw]);
   ...
}
```

The effect of this is to insert a single `lock` variable into the `BankAccount` class, and to advise both the `deposit` and `withdraw` methods to acquire this lock before operating and release it afterwards. Note that this implementation not only permits multiple classes to apply the `Synchronization` aspect, but also allows a class to apply it multiple times, specifying different sets of methods each time. That would have the effect of synchronizing each set of methods separately (permitting methods from different sets to operate concurrently) by giving each set its own lock.

4.5 Visibility and scope of effects

In AspectJ, an aspect may modify the behaviour of any class in the system as long as we have its source code. Such modification can vary from a simple augmenting behaviour to a complete replacement of the matched join-points. Furthermore, it can even modify the structure or internal representation of a class by inserting new fields, methods, and constructors (in AspectJ, this is called an *intertype member declaration*).

In ParaAJ, on the other hand, a para-aspect can only modify the behavior of the class to which the aspect is applied (for an *internal application, this is the class containing the* `apply` *declaration*). Furthermore, the only fields, methods, classes, *etc.* that can be used in an `apply` declaration are those that are visible in the scope of the `apply` declaration. Thus an external application may not mention private members of the target class. This preserves information hiding, allowing the target class maintainers to change implementation details without affecting the aspects externally applied to that class.

In an internal application, since the apply declaration is inside the target class (the class is both target and host), it can see the internal representation of the class and also modify it. However, in an external application, the para-aspect has full access to the host class but only limited access to the target class. External aspect applications:

- can advise only *public* parts of the target class.
- cannot have replacing advice
- injected members are only visible to the host class.

4.6 ParaAJ pointcuts

In AspectJ, pointcuts depend on *patterns* to match the required joinpoints. For example, `get` and `set` pointcuts use *field patterns* while `call` and `execution` pointcuts use *method and constructor patterns*. In either case, the pattern includes a part to indicate the target class(es). As we are inferring the target class from the `apply` statement in ParaAJ, we should not allow target information to be included in the pattern. To illustrate, the following pattern in AspectJ:

$$\texttt{set}(\underbrace{\texttt{int Foo.bar}}_{fieldpattern})$$

could be simplified and written as

```
set(int bar)
```

in ParaAJ, the compiler will infer the target class automatically from the `apply` statement. Hence, the above statement is equivalent to

```
set(int thisTarget.bar)
```

To illustrate, consider the following *tracing* aspect that traces any changes to a certain field.

```
public aspect FieldMonitor(field f) {
    after(): set($[f]){
        ...
    }
}
```

we can apply it to the `givenName` field in the following `Student` class:

```
public class Student {
    private String givenName, familyName;
    apply FieldMonitor(givenName);
     ...
}
```

the `set` pointcut in this aspect will be transformed by the parser to:

```
set(thisTarget.$[f])
```

and at a later stage during compilation (discussed in section 5.4), it will be transformed to:

```
set(Student.$[f])
```

Thus we prevent our aspect from accessing and monitoring classes other than the target class.

4.7 Injection

Members defined in para-aspects are normally visible only to the aspect, not the host or target. Furthermore, each application of an aspect has its own "copy" of each such member. This permits a single aspect to be applied several times to the same class without the separate applications interfering with one another.

However, a member defined in an aspect may be made visible to the *host* class by preceding the name of the member in the declaration with the name of the target class. In AspectJ, this is called *intertype member declaration (IMD)*. In the same way we simplified pointcut patterns in the previous section, we implemented a special type of IMD, called `inject` declaration, that removes the `target` part from the declaration. For example, intertype field declaration in ParaAJ is declared in this way:

inject ⟨***type***⟩ ⟨***name***⟩**;**

similarly, method and constructor declarations may be made visible to the host class as follows:

inject ⟨***type***⟩ ⟨***name***⟩**(**⟨***formals***⟩**){** ⟨***body***⟩**}**
inject new(⟨***formals***⟩**){** ⟨***body***⟩ **}**

If the host and target are different classes (*external application*), the member is not visible to the target, only the host, since it is the host class that contains the `apply` declaration. For example, we could implement a para-aspect to inject a unique serial number in each instance of a class as follows:

```
public aspect SerialNumber(ident name) {
    static int counter = 1;
    inject public final int $[name] = counter++;
}
```

Any class could then introduce a constant public `id` member holding a unique serial number for each instance with an apply declaration:

```
apply SerialNumber(id);
```

Note that each `apply` declaration for this aspects introduces a separate `counter` member. However, since `counter` does not have `inject` visibility, it is visible only inside the `SerialNumber` aspect; it cannot be used in classes that apply this aspect. This supports information hiding. Since it is `static`, all instances of each class that applies this aspect share a single `counter`. In contrast, as it has `inject` visibility, the introduced `id` is visible in the applying class, and since it is also `public` and `final`, it is a publicly accessible constant in each instance.

5 Implementing ParaAJ

We have begun implementing the ParaAJ extension using the AspectBench Compiler (*abc*) (Avgustinov et al. 2005). *abc* is an alternative AspectJ compiler developed by a team from Oxford and McGill universities, and BRICS[2]. One of its main design goals is *extensibility* which is exactly what we need in ParaAJ. The Front-End of *abc* has been developed with *Polyglot* (the extensible Java compiler) . Its Back-End has been built with the *Soot* optimization framework.

The ParaAJ extension requires modification to different parts of the front-end of *abc*:

- **Parser**: adding new syntax to the language,
- **Abstract syntax Tree (AST)**: adding new nodes to the language,
- **Type System**: adding Parametric Aspect type and other Meta-types (field, method, etc.), and
- **Passes**: checking and transforming the new AST nodes.

[2] Basic Research in Computer Science (`www.brics.dk`), funded by the Danish National Research Foundation.

5.1 Extending the Parser

In *abc*, modifying the grammar and including new syntax can be done using the Polyglot Parser Generator (PPG). In PPG, rules can be easily added or dropped from the grammar using `extend` and `drop` statements.

Unlike normal aspects, para-aspects include formal parameter lists to specify their interfaces. Hence, we extended the `type_declaration` rule to include a `para_aspect_declaration` rule as follows:

```
extend type_declaration ::=  para_aspect_declaration
para_aspect_declaration ::=
  ⟨modifiers⟩ aspect ⟨name⟩ (⟨formal_parameter_list⟩){
    ⟨aspect_body⟩
  }
```

An `apply` statement can appear anywhere a class member declaration (such as a field or method declaration) can. It has two different forms, one for internal application and the other for external application. The syntax for internal application is:

```
apply ⟨aspect_name⟩ (⟨argument_list⟩);
```

where ⟨*argument_list*⟩ is a list of the actual values that will be passed to the para-aspect specified by ⟨*aspect_name*⟩.

In an external application, two classes are involved: the host of the `apply` statement and the target class. The syntax is:

```
⟨class_name⟩.apply ⟨aspect_name⟩ (⟨argument_list⟩);
```

where ⟨*class_name*⟩ is the name of the target class.

Meta nodes are not as straightforward as para-aspects and the `apply` statement. Meta nodes appear inside the body of a para-aspect to replace a node with its *meta* version. For this reason, we cannot simply use one rule to implement all meta rules in our extensions. We give just one example to illustrate how to accomplish this. `Type` is a Java rule to represent *primitive* types (e.g. `int`, `char`, etc.) or *reference* types (e.g. `String`, `java.util.List`, user created types, etc.). The `type` rule is used in many rules in Java and AspectJ. For example, in Java it is used in field declarations, method headers (return types), and formal parameters. Also, in AspectJ, it is used in intertype member declarations and type patterns. In all these places, we want to allow a meta-type instead of regular type nodes. To do this, we create a `meta_type` rule extending the original `type` rule in Java:

```
extend type ::= meta_type
```

and define the `meta_type` rule as follows:

```
meta_type ::= $[ ⟨symbol⟩ ]
```

where ⟨*symbol*⟩ is an identifier corresponding to one of the aspect formal parameters. In a similar way, we can extend the syntax to allow any kind of *meta nodes* (meta field declarations, meta intertype field declarations, meta pointcuts, etc.).

5.2 Creating new AST nodes

All rules in the language grammar should return a valid node in order to construct a valid AST. As followed in Polyglot, each node has to be implemented as a pair of an interface and a class: an *interface a* implementing the top-level `Node` interface and a class *b* extending the top-level `Node_c` class and implementing the interface *a*.

Referring to our `meta_type` rule in the ParaAJ grammar, we have created a corresponding `MetaTypeNode` interface and a `MetaTypeNode_c` class to represent this kind of node. In this way, we can manipulate this node during the various compiler passes. Again, in a similar way, we can create nodes for all other grammar rules in the ParaAJ extension.

5.3 Type System

In ParaAJ, we have introduced several new types (see section 4.2) to be used in our parametric aspects. All these types have to be defined in the *type system* so that the compiler can understand them.

5.4 Compiler Passes

The polyglot compiler works as a series of *passes* that check and transform the AST. *abc* has a total of 40 passes (excluding some optional debugging passes). In ParaAJ, we have added 7 extra passes to manipulate our para-aspects. These passes are:

1. **Collect Para-Aspects names**: traverse the AST and collect all names of para-aspects.
2. **Meta Ambiguity Remover**: make sure that each meta symbol in the para-aspect corresponds to one of the formal parameters in the aspect definition.
3. **Collect Apply Statements**: traverse the AST and collect all `apply` statements. This pass also checks the correctness of the indicated para-aspect and whether the the supplied *arguments* match the aspect formal parameters or not.
4. **Transform Apply to Aspect**: transform each apply declaration to a blueprint of the para-aspect. This pass keeps the actual arguments to be used in the next pass.
5. **Replace Meta Nodes**: Replace any meta node in the generated aspect with the actual value from the `apply` statement arguments. In this pass, any meta node is converted to its corresponding non-meta version. For example, `MetaTypeNode` is converted to `TypeNode`.
6. **Build Types Again**: `BuildTypes` is a Java pass that runs right after the parsing phase. In this pass, we need to build the types for the newly created aspects.
7. **Para-Aspect Converter**: convert all para-aspects to normal AspectJ aspects so that they can be manipulated by other *abc* passes.

All of these passes get executed after completion of the initial three passes: parser, type builder, and ambiguity remover passes. We have also replaced some of the *abc* passes with our own version. For example, we have replacedthe original type builder with our type builder that can recognize the new para-aspects.

5.5 Future Work

We have implemented a large enough portion of the language to test our approach. This includes grammar, AST, type system, and compiler passes for the following:

- para-aspects
- apply declaration (internal)
- Meta nodes
  - Meta type
  - Meta field/method/constructor declaration
  - Meta intertype field/method/constructor declaration
  - Meta field/method/constructor patterns (used in pointcuts)

However, we are still completing and improving our implementation.

6 Related Work

Creating modular, maintainable, and reusable aspects has been a concern for many researchers. Parametric introductions have been proposed by Hanenberg & Unland (2003*b*) to provide a modular way to inject new members into target classes. This work provides a better solution to develop reusable aspects for cases where modular techniques in AspectJ (abstract aspects) are inadequate. However, the main focus was on parameterizing introductions rather than the whole aspect. This might reduce the chance of reusing aspects when other parts of the aspects (*e.g.,* pointcuts) need to be parameterized. Moreover, this solution does not help in resolving the invisibility issue discussed before. In other word, target classes are still oblivious to the aspects that affect them.

The idea of parametric aspects was initially introduced, without in-depth discussion, by Alvarez (2004). He proposed this idea as an extension to the AOP languages to increase the opportunities for aspect reuse. The ideas in this approach are closely related to our work and the proposal nicely shows how parametric aspects can be used to provide a solution for some of the popular design patterns (*e.g.,* the factory design pattern). In his approach, parametric aspects are declared *abstract* and instantiated as concrete aspects using the *extends* keyword.

Alvarez's language extension proposed a new keyword (*roles*) to represent parameters. Unlike ParaAJ, parameters in this language are generic (i.e. they might correspond to classes, methods, fields, etc.) In ParaAJ, on the other hand, parameters are *typed* and can be checked for correctness by the compiler. We believe that our approach is more general than Alvarez's (2004) and aspect instantiation is simpler with the `apply` declaration.

Aspect-Aware Interfaces (Kiczales & Mezini 2005) and Crosscutting Programming Interfaces (XPI) (Griswold et al. 2006) are interface-based approaches that separate aspects from the details of their target code. These approaches and similar ones like the aspect idioms and design patterns (Hanenberg & Unland 2003*a*, Hanenberg & Schmidmeier 2003) are very helpful in building reusable aspects. However, we believe that adding new programming constructs and techniques is better than these *indirect* solutions. As we discussed in section 3, these kinds of solutions add complexity to the system design and code.

Another line of research involves minimizing or eliminating the bad effect of aspects on main classes. *Open modules* (Aldrich 2005) and *pure aspects* (Recebli 2005) have been proposed to restrict aspects from modifying base code. The idea in these proposals is to define an interface containing the set of joinpoints

that can be altered by aspects. Although this idea resolves some of the maintainability problems produced by the obliviousness of classes to aspects code, it is limited in terms of aspect reuse as shown by Wampler (2006).

7 Summary

In this paper, we presented our language, ParaAJ, which extends AspectJ with the ability to parameterize aspects. Using parameters, we go a step further toward modularizing aspects and increase the chance of reusing them. Also, with the use of parameters, we can create an explicit interface between aspects and their target classes which can be checked at compile-time to insure correctness of aspect-class integration. Unlike normal aspects (as in AspectJ), ParaAJ aspects need to be *applied* to their target classes in order to take effect in the program. We showed some example aspects that can be nicely implemented as para-aspects. The paper also gives a brief description of our implementation of the language. We have implemented a large enough portion of the language to test our approach. However, we are still working on the language: implementing what is remaining and improving what we have already implemented. We believe that our language, ParaAJ, will open a new direction in writing more maintainable and reusable aspects and classes.

References

Al-Jasser, K., Schachte, P. & Kazmierczak, E. (2007), Suitability of object and aspect oriented languages for software maintenance, *in* 'ASWEC '07: Proceedings of the 2007 Australian Software Engineering Conference', IEEE Computer Society, Washington, DC, USA, pp. 117–128.

Aldrich, J. (2005), Open modules: Modular reasoning about advice, *in* 'Proceedings ECOOP 2005', Vol. 3586 of *LNCS*, Springer Verlag, Glasgow, UK, pp. 144–168.

Alvarez, J. (2004), Parametric aspects: A proposal, *in* W. Cazzola, S. Chiba & G. Saake, eds, 'ECOOP'04 Workshop on Reflection, AOP, and Meta-Data for Software Evolution (RAM-SE)', pp. 91–100.

Avgustinov, P., Christensen, A. S., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G. & Tibble, J. (2005), 'abc: An extensible AspectJ compiler', *Transactions on Aspect-Oriented Software Development* .

Clifton, C. & Leavens, G. T. (2003), Obliviousness, modular reasoning, and the behavioral subtyping analogy, *in* L. Bergmans, J. Brichau, P. Tarr & E. Ernst, eds, 'SPLAT: Software engineering Properties of Languages for Aspect Technologies'.

Filman, R. E. & Friedman, D. P. (2000), Aspect-oriented programming is quantification and obliviousness, *in* 'OOPSLA 2000 Workshop on Advanced Separation of Concerns', Minneapolis, MN.

Griswold, W. G., Sullivan, K., Song, Y., Shonle, M., Tewari, N., Cai, Y. & Rajan, H. (2006), 'Modular software design with crosscutting interfaces', *IEEE Software* pp. 51–60.

Hanenberg, S. & Schmidmeier, A. (2003), Idioms for building software frameworks in AspectJ, *in* Y. Coady, E. Eide & D. H. Lorenz, eds, 'The Second AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)'.
URL: *http://www.cs.ubc.ca/ ycoady/acp4is03/papers/hanenberg.pdf*

Hanenberg, S. & Unland, R. (2003*a*), AspectJ idioms for aspect-oriented software construction, *in* '8th European Conference on Pattern Languages of Programs (EuroPLoP)', Irsee, Germany 2003.

Hanenberg, S. & Unland, R. (2003*b*), Parametric introductions, *in* 'AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development', ACM Press, New York, NY, USA, pp. 80–89.

Kersten, M. & Murphy, G. C. (1999), 'Atlas: a case study in building a Web-based learning environment using aspect-oriented programming', *ACM SIGPLAN Notices* **34**(10), 340–352.
URL: *citeseer.ist.psu.edu/article/kersten99atlas.html*

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. & Griswold, W. G. (2001), 'An overview of AspectJ', *Lecture Notes in Computer Science* **2072**, 327–355.

Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. & Irwin, J. (1997), Aspect-oriented programming, *in* M. Akşit & S. Matsuoka, eds, 'Proceedings European Conference on Object-Oriented Programming', Vol. 1241, Springer-Verlag, Berlin, Heidelberg, and New York, pp. 220–242.

Kiczales, G. & Mezini, M. (2005), Aspect-oriented programming and modular reasoning, *in* 'Proc. of the 27th International Conference on Software Engineering', ACM, pp. 49–58.

Myers, G. (1978), *Composite/Structured Design*, Van Nostrand Reinhold, New York.

Recebli, E. (2005), Pure aspects, Master's thesis, University of Oxford.
URL: *citeseer.ist.psu.edu/739021.html*

Rinard, M., Salcianu, A. & Bugrara, S. (2004), A classification system and analysis for aspect-oriented programs, *in* 'SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering', ACM Press, New York, NY, USA, pp. 147–158.

Seiter, L. M. (2008), Balancing quantification and obliviousness in the design of aspect-oriented frameworks, *in* H. Mei, ed., 'ICSR', Vol. 5030 of *Lecture Notes in Computer Science*, Springer, pp. 318–329.

Wampler, D. (2006), The challenges of writing reusable and portable aspects in aspectj: Lessons from contract4j, *in* 'In Proceedings of International Conference on Aspect Oriented Software Development (AOSD 2006)'.