technische universität
dortmund

Diploma Thesis

Metadata Annotations for Explicit Joinpoints in AspectC++

Sven Radetzky
September 29, 2011

Adviser:
Prof. Dr.-Ing. Olaf Spinczyk
Dipl.-Inf. Christoph Borchert

Technische Universität Dortmund
Computer Science 12
Embedded System Software Group
http://ess.cs.tu-dortmund.de

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Dortmund, den September 29, 2011

Sven Radetzky

Abstract

Aspect-Oriented Software Development (AOSD) intends to improve the separation of concerns in a program by quantification of cross-cutting concerns oblivious to the underlying base code. *Aspect-Oriented Programming* (AOP) languages, such as AspectJ and AspectC++, provide numerous means to quantify and identify sections in a programs base code to advice with the code of cross-cutting concerns. These so called joinpoints are identified by pointcuts, which have their own potent description language. Recently the addition of new types of more fine-grained joinpoints into AOP languages have become an active area of interest for researchers. In combination with an annotation mechanism which allows the addition of metadata on the elements of a program, possibilities for far more fine-grained advice mechanisms in mainstream AOP languages were realized. The goal of this thesis is to provide AspectC++, an AOP language based on the C++ programming language, with a metadata annotations mechanism. Which in turn provides, in combination with an extended pointcut language, the means for a more fine-grained joinpoint model. This is an interesting concept especially in regards to the unique language features present in the C++ language, like templates and template metaprogramming. This thesis describes the design, implementation and evaluation of the described mechanism and related features in AspectC++.

Contents

1 Introduction

Aspect-Oriented Software Development (AOSD) intends to improve the separation of concerns in a program by quantification of cross-cutting concerns oblivious to the underlying base code. *Aspect-Oriented Programming* (AOP) languages, such as AspectJ and AspectC++, provide numerous means to quantify and identify sections in a programs base code to advice with the code of cross-cutting concerns. These so called joinpoints are identified by pointcuts, which have their own potent description language. Recently the addition of new types of more fine-grained joinpoints into AOP languages have become an active area of interest for researchers.

After positive results concerning the usefulness of extending the AspectJ joinpoint model it has become of interest if such an extension in the context of AspectC++ would lead to an improvement in the languages usage, maintainability of source code and support of evolving code bases.

Furthermore the unique language features present in C++, like templates and template metaprogramming, make AspectC++ a very interesting target language for this thesis.

The problem both AOP languages face in that regard is how to extend the pointcut language without letting pointcuts become too complex. An example of that phenomenon can be found in *LoopAJ* [1] which extensions of the pointcut language are all restricted by severe limitations on source code structure. This thesis will extend the joinpoint model as generic as possible so that the usage of the new joinpoints is not artificially limited by factors from outside the scope of the programming language itself. This thesis uses the approach of using metadata annotations to allow the explicit declaration of joinpoints in a programs base code, which has been used to a small degree in similar works done on the AOP language AspectJ.

The goal of this thesis is to provide AspectC++ with a metadata annotations mechanism, which in turn provides, in combination with an extended pointcut language, the means for a more fine-grained joinpoint model.

This thesis describes the design of the metadata annotations feature in AspectC++ as well as examines the possible extensions that can be made to the joinpoint model with such a feature. In addition a rudimentary implementation into the AspectC++ compiler is provided and evaluated.

1.1 Goals and Relevance

This thesis analyzes the joinpoint model of AspectC++ and examines possibilities to extend the ability of AspectC++ to address cross-cutting concerns beyond the limitations of the current joinpoint model. Furthermore it will be shown that metadata annotations are a key concept that can be used to address joinpoints based on generic language elements in the C++ base language AspectC++ is based upon. These metadata annotations will be well-defined in the scope of this thesis. For those purposes the three main goals of this thesis are:

Design new joinpoint types

Based on an extensive analysis of the current model and an analysis of limitations found in the current model.

Reflection and Refinement

Allow the annotation of language elements that are used to create joinpoints. Show how metadata annotations remove limitations concerning the design of additional joinpoint types in the AspectC++ joinpoint model.

Evaluation

Evaluate the combination of metadata annotations and an extended joinpoint model in terms of improvements concerning AOP implementation mechanisms and the level of expressiveness of the AspectC++ language.

1.2 Structure of this Thesis

Following the introduction in this chapter, chapter 2 describes the basics of metadata annotations. Chapter 3 provides an introduction to AOSD as well as a number of AOP solutions. Then chapter 4 is used to analyze the existing AspectC++ joinpoint model and defines the requirements that apply for the rest of this thesis. Chapter 5 then provides a complete definition of metadata annotations in AspectC++ as well as a summary of possibilities with regards to the conclusions from chapter 4. Chapter 6 provides a summary of the practical implementation of that solution. Whereas Chapter 7 describes the evaluation process and the collected results as well as allows comparisons with similar approaches done for different AOP languages. At the end, chapter 8 provides a summary of the conclusions drawn from this thesis as well as suggestions for future work which would be interesting in regards to further research concerning AspectC++.

2 Metadata Annotations

Metadata, literally means “data about data“ [2, 3] and can in general be separated into two types, *descriptive* and *structural* metadata. Where descriptive metadata refers to additional information a data set and structural metadata refers to additional information on the structure of data. It has become a widely used term especially, but not exclusively, in regards to web-based information systems.

metadata is present in many aspects of everyday computing. Adding information about the persons shown in a photograph on Facebook is descriptive metadata. Using ”hashtags” on Twitter is applying descriptive metadata to once “tweets“. Software to manage photo-collections supports the use of ”tags“, which are also a form of descriptive metadata, to provide additionally information to sort and group pictures. The popular XML format [4] is also often used to apply metadata to information where a XML file is used as descriptive metadata and a XML schema is used for the corresponding structural metadata.

Those are only examples of the myriad of ways metadata is used today. It has also found its way into many disciplines of computer science, including a number of programming languages which support its use in different ways.

2.1 Introduction

Today there are a number of programming languages, which support the use of structural metadata. Nonetheless this thesis focuses on the use of descriptive metadata, to inscribe additional information on elements of a programs source code, only. The rest of this chapter aims to provide an introduction into the usage of metadata in modern programming languages, with a focus on descriptive metadata. For clarity purposes descriptive metadata that is used in a programming language to add additional data to an object is from now on referred to as a *metadata annotation*.

2.2 Java

The Java programming language has an integrated *Metadata Facility* [5] since version 5.0 which supports the use of structural as well as descriptive metadata. In addition there exists an API interface for the dynamic runtime processing of metadata [6].

2.2.1 Predefined Annotations

Metadata annotations in Java can be applied to a number of Java language elements, like classes and methods but can also be used on a statement level or on arguments. For example Java version 5.0 has three predefined standard annotations, namely "`@Override`", "`@Deprecated`" and "`@SuppressWarnings`". The first two will now be used to demonstrate how annotations in Java can be applied.

2.2.1.1 `@Override`

"`Override` should be used only on methods (not on classes, package declarations, or other constructs). It indicates that the annotated method is overriding a method in a superclass." [7].

```
1  public class OverrideTester {
2    public OverrideTester() { }
3
4    @Override
5    public String toString() {
6      return super.toString () + " [Override Tester Implementation]";
7    }
8
9    @Override
10   public int hasCode() {
11     return toString.hashCode();
12   }
13 }
```

Figure 2.1: Example of using the Override annotation [7]

The example in Figure 2.1 shows the use of the `Override` annotation, the function name in line 10 contains a typo. The correct function name should read "hashCode". When compiling this code the compiler checks the superclass for a function with the same name. If such a function is not present the compiler issues a warning.

2.2.1.2 `@Deprecated`

"As you might expect, you use `Deprecated` to annotate a method that shouldn't be used anymore. Unlike `Override`, `Deprecated` should be placed on the same line as the method being deprecated (why? I'm honestly not sure)." [7].
Figure 2.2 shows how to use the `Deprecated` annotation this time the compiler won't issue a warning when compiling this code yet, the compiler should issue warnings for any other location in the source code, at which the deprecated method be used or overridden.

2.2.1.3 `@SuppressWarnings`

This annotation suppresses warnings issued by the compiler at compile time. It can be used by a programmer to suppress warnings that are not relevant to the current

```java
public class DeprecatedClass {
  @Deprecated public void doSomething() {
    // some code
  }

  public void doSomethingElse() {
    // This method presumably does what doSomething() does, but better
  }
}
```

Figure 2.2: Example of using the Deprecated annotation [7]

debugging process and can therefore improve readability of compiler output in some circumstances.

2.2.2 Custom Annotations

Java not only supports predefined annotations it also offers a programmer the ability to define his own custom annotations as well as mechanisms for the processing of such annotations dynamically at runtime [6]. Those custom annotations in turn are again standard Java classes which support all the expressiveness of the Java programming language.

(a) Defining a custom annotation

```java
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)

public @interface customAnnotation {
    public String name();
    public String value();
}
```

(b) Annotating an element with a custom annotation

```java
@customAnnotation(name="someName",  value = "Hello World")
public class TheClass {

}
```

Figure 2.3: Custom Java annotations part 1

An example of a custom annotation can be seen in figure 2.3 (a). The `@` in front of the interface marks it as an annotation. Once the annotation is defined it can be used in the programs code, as shown earlier. The two directives in the annotation definition, `@Retention(RetentionPolicy.RUNTIME)` and `@Target(ElementType.TYPE)`, specify how the annotation is to be used. `@Retention(RetentionPolicy.RUNTIME)` means that the annotation can be accessed via reflection at runtime. If this directive is not set, the annotation will not be preserved at runtime, and thus not available via

reflection. `@Target(ElementType.TYPE)` means that the annotation can only be used to annotate types (classes and interfaces typically). It also possible to specify ElementType.METHOD or ElementType.FIELD as the target types, or the target can be left out so the annotation can be used for classes, methods and fields.

In part (b) of the figure an example is given on how such an annotation is applied to an object and figure 2.4 part (c) demonstrates two ways to access the information present in such an annotation at runtime using the Java reflection mechanism. One way is to retrieve an array holding all annotations of an object and then processing each annotation through a loop for example. The other way is to retrieve only specific annotations from an object and access the information directly afterwards.

The reflection example described above focuses on classes, the methods used for retrieving annotations of methods, parameters or fields is quite similar.

(c) Accessing information of custom annotations

```
1 /* Accesing every annotation on an object */
2 Class aClass = TheClass.class;
3 Annotation[] annotations = aClass.getAnnotations();
4
5 for(Annotation annotation : annotations){
6   if(annotation instanceof customAnnotation){
7     customAnnotation myAnnotation = (customAnnotation) annotation;
8     System.out.println("name: " + myAnnotation.name());
9     System.out.println("value: " + myAnnotation.value());
10  }
11 }
```

```
1 /* Accessing only a specific annotation on an object */
2 Class aClass = TheClass.class;
3 Annotation annotation = aClass.getAnnotation(customAnnotation.class);
4
5 if(annotation instanceof customAnnotation){
6     customAnnotation myAnnotation = (customAnnotation) annotation;
7     System.out.println("name: " + myAnnotation.name());
8     System.out.println("value: " + myAnnotation.value());
9 }
```

Figure 2.4: Custom Java annotations part 2

2.2.3 Type Annotations

In Java version 7.0 the metadata syntax is extended to allow the annotation of types [8]. Although those annotations are not yet completely integrated into the Java compiler, they can be used with the use of a special Type annotations compiler. This type annotation compiler is completely backward compatible to the standard Java compiler and allows type checking based on those type annotations.

For example consider the example in figure 2.5, the type annotation compiler would

check that no object inside the list would ever equal null, therefore removing the need for the programmer to check against such behavior at multiple places in the source code as is usual. Type annotations can be applied to any place a type is used, that means variable definitions, function argument definition and also where the return type of a function is defined.

```
1 /* prevent any list element becoming == null */
2 List<@NonNull Object> list;
3
4 /* type annotation in a function definition */
5 public static @NonZero int calculate_something(@Nullable int i) {
6   ...
7 }
```

Figure 2.5: Example of a type annotation in Java 7.0

2.3 C#

The programming language C# also supports metadata on a source code level. A programmer can define so called attributes which are used to store additional data in context to many elements in a program similar to Java although the syntax is quite different. In addition like Java, it allows for the definition of custom attributes which can be accessed through a reflection mechanism to access the metadata at runtime. Figure 2.6 (a) shows how to assign metadata in C# and part (b) shows an example of how the reflection mechanism can be used.

(a) Assigning metadata to objects

```
1 [BugFixAttribute(766,"Max Mustermann","30/7/2011") Comment="Description of bug"]
2 public class MyClass {
3   ...
4 }
```

(b) Reflection on metadata

```
1   System.Reflection.MemberInfo inf = MyClass;
2   object[] attributes;
3   attributes = inf.GetCustomAttributes(typeof(BugFixAttribute), false);
4
5   // iterate through the attributes, retrieving the properties
6   foreach(Object attribute in attributes) {
7     BugFixAttribute bfa = (BugFixAttribute) attribute;
8     Console.WriteLine("\nBugID: {0}", bfa.BugID);
9     Console.WriteLine("Programmer: {0}", bfa.Programmer);
10    Console.WriteLine("Date: {0}", bfa.Date);
11    Console.WriteLine("Comment: {0}", bfa.Comment);
12  }
```

Figure 2.6: Example of metadata annotations in C#

Furthermore Walter Cazzola has extended the annotation mechanisms used in C# for his own research in [9]. A number of examples based on this example can be seen in figure 2.7.

(c) Extensions on annotations

```
1  public void function () {
2    [MyAnnotation] {
3      // allows for block statement code to be annotated
4      [anotherAnnotation] {
5  // also allows nested annotations now
6      }
7    }
8  }
```

Figure 2.7: Example of extended metadata annotations in C#

3 Aspect-Oriented Software Development

This chapter provides an introduction into *Aspect-Oriented Software Development* (AOSD). Additionally this chapter gives an overview on metadata annotations used in *Aspect-Oriented Programming* (AOP) languages and introduces works related to this thesis concerning the extension of joinpoint models in AOP languages.

3.1 Introduction

The *Object-Oriented Programming* (OOP) paradigm improves modularity by encapsulating data with methods inside objects. Most often the data can only be accessed indirectly through the associated functions. Although the concept appeared in the seventies, it took twenty years to become popular [10].

The difficult aspect concerning object-oriented design is how to decompose a system into objects, because many factors have to be considered: encapsulation, granularity, dependency, adaptability, reusability, and others. They all influence the decomposition, often in conflicting ways [11].

Existing modularization mechanisms typically support only a small set of decompositions and usually only single dominant decomposition at a time. This is known as the tyranny of the dominant decomposition [12]. This limits the ability to implement other concerns in a modular way. For example, OOP modularizes concerns in clases and only fixed relations are possible. Implementing a conern in a class might prevent another concern from being implemented as a class [13].

Aspect-Oriented Programming (AOP) is a paradigm that intends to solve this problem.

AOP is most often used in combination with OOP but can also be applied to different paradigms, for example some event-based systems can be considered aspect-oriented [14]. The following two sections introduce an example to demonstrate the problems with a pure OOP approach as well as ways AOP resolves those. Afterward a number of AOP methodologies relevant to the rest of this thesis are introduced.

(a) Addition

```
1  class Add : public Calculation {
2  private:
3    int result;
4    CalcDisplay *calcDisplay;
5    Tracer *tracer;
6  public:
7    Add () {
8      result = 0;
9      calcDisplay = new CalcDisplay ();
10     trace = new Tracer ();
11   }
12
13   void execute (int a, int b) {
14     tracer->write ("void Add.execute(int, int)");
15     result = a + b;
16     calcDisplay->update (result);
17   }
18
19   int getLastResult () {
20     tracer->write ("int Add.getLastResult()");
21     return result;
22   }
23 };
```

(b) CalcDisplay

```
1  class CalcDisplay {
2  private:
3    Tracer *tracer;
4  public:
5    CalcDisplay () {
6      tracer = new Tracer ();
7    }
8
9    void update (int value) {
10     tracer->write ("void CalcDisplay.update(int)");
11     cout << "Printing new value of calculation: " << value << endl;
12   }
13 };
```

Figure 3.1: Modeling addition, display and logging without aspects [13]

3.2 Traditional Approach

For our example lets consider an application that contains an object `Add` and an object `CalcDisplay`. `Add` inherits from the purely virtual class `Calculation` and implements the method `execute(a, b)`. It adds two integer variables, stores the result value and then tells `CalcDisplay` to print the result to the standard output. In this example our cross-cutting concern is tracing. All methods have to be traced, for this purpose exists another class `Tracer` which is used to write tracing messages using the method `write`. Figure 3.1 shows a possible implementation.

In this example we have two forms of *cross-cutting*: *code tangling* and *code scatter-*

ing. The tracing concern is scattered across the classes `Add` and `CalcDisplay` (lines 5, 10, 14 and 20 in (a) as well as 3, 6 and 10 in (b)). A concern, like tracing in our example, that is implemented across several classes is called scattered.

Most of the time a scattered concern involves code *replication*. Which means that the same code is implemented at a number of different places. In Figure 3.1 all calls to the tracing object are significantly similar.

Code tangling is the implementation of different concerns inside the same class. The tracing concern in Figure 3.1 is tangled with both the addition as well as the display concern. Code tangling and code scattering have the following consequences [13]:

Code is difficult to change

Changing a scattered concern requires modification at different places. Making modifications to a tangled concern class requires checking for side-effects against the other cross-cutting concerns.

Code is harder to reuse

Reusing an object in another software project, requires to either remove or reuse the tangled concerns as well.

Code is harder to understand

Without separation of concerns it becomes difficult to understand which code belongs to which concern.

3.3 AOP Approach

To solve cross-cutting problems, several techniques are being researched to increase the expressiveness of Object-Oriented paradigms. AOP is one of those techniques. AOP introduces a new structure, the *aspect*, to encapsulate the cross-cutting concerns behavior.

The fundamental goals of AOP are twofold [15]: first, to provide mechanisms to express cross-cutting concerns applied to other components. Second, to allow improved separation of concerns with those mechanisms.

Aspects are used to describe *advice* which is behavior which can be executed at a *joinpoint*. Joinpoints are well-defined elements in the structure of code or points in a programs execution flow where additional behavior is applied. Most commonly class and method names as well as method calls are used as joinpoints in an AOP language. Furthermore *pointcuts* are used to describe a set of joinpoints. Combining advice with pointcuts allow to execute advice behavior at several places in a program at once.

(a) Addition concern

```
1  class Add : public Calculation {
2  private:
3    int result;
4    CalcDisplay *calcDisplay;
5  public:
6    Add () {
7      result = 0;
8      calcDisplay = new CalcDisplay ();
9    }
10
11   void execute (int a, int b) {
12     result = a + b;
13     calcDisplay->update (result);
14   }
15
16   int getLastResult () {
17     return result;
18   }
19 };
```

(b) Tracing concern

```
1  aspect Tracing {
2    Tracer *tracer = new Tracer ();
3    advice call("% Calculation::%(...)") || call("% CalcDisplay::%(...)") : before () {
4      tracer->write (tjp->signature ());
5    }
6  };
```

Figure 3.2: Modeling addition, display and logging with aspects [13]

Figure 3.2 shows an AOP implementation of the example used in the previous section. The tracing concern is encapsulated in an aspect called `Tracing` which advises every function call to the classes `Calculation` and `CalcDisplay` with tracing code to print out the necessary tracing information. The code tangling and code scattering has been removed from the class `Add` allowing an easier understanding of the addressed concern.

The code handling the cross-cutting concern is completely encapsulated in the aspect instead of being embedded within other objects. This has several advantages [13]:

Aspect code can be changed

Changing aspect code does not influence other concerns in a software project.

Aspect code can be reused

Aspects are coupled with other objects by defining pointcuts. In theory, this low coupling allows reuse. In practice reuse is still difficult.

Aspect code is easier to understand

Understanding a concern is independent from other concerns.

Aspect pluggability

It is possible to enable and disable aspects and therefore concerns.

The ability to enable and disable aspects depending on circumstances makes AOP an interesting paradigm for developing *Software Product Lines* (SPL) [16, 17].

3.4 AOP Solutions

There are numerous different AOP solutions, but for the purpose of this thesis not all are interesting, therefore the focus is going to be on AspectJ and related AOP implementations. All those implementations have a couple of things in common.

They all utilize asymmetric AOP composition, which means the base program and aspects are distinguished and the base program is composed with the aspects. Where as in symmetric AOP composition every component can be composed with any other component. For example Hyper/J follows this approach [13].

3.4.1 Aspect Weaving

The integration of aspects and components is called *aspect weaving*. There are different ways to weave aspects, the approach shared by most of the relevant AOP implementations for this thesis is called *source code weaving*. This approach combines the aspect code in predefined ways with the original source code and creates its output as native source code for the target language that can be compiled using a native compiler. This approach to aspect weaving has the following advantages [13]:

High-level source modification

All modifications are done on a source language level, removing the need to have any information about the target (output) language of the native compiler.

Aspect and original source optimization

After weaving the aspects into the original source code the outputted code is optimized by the native compiler, while this doesn't allow for aspect specific optimizations, it allows for all the benefits of the native compilers optimization phases.

Native compiler portability

Any compiler for a specific source language can be used, newer versions of the same compiler with improved optimizations are also possible. Furthermore it possible to port a software project utilizing such an AOP solution to any platform that has a compiler for the source language.

But it also has disadvantages [13]:

Language dependency

Source code weaving is written explicitly for the syntax of the input language.

Limited expressiveness

Aspects are limited to the expressive power of the source language. For example, it is not possible to add multiple inheritance to a single inheritance language.

3.4.2 AspectJ

The *AspectJ* [18] programming language is an aspect-oriented extension of the Java programming language. It was developed by Gregor Kiczales who was also the person who wrote the first paper on AOP [19] back in 1997. AspectJ is an open eclipse project, which means it can be used by anyone who wants to use it. This was done to encourage a wider community of people to develop aspect-oriented software and also to provide researcher with new ideas for AOP.

Nearly all AOP languages relevant to this thesis are closely related to AspectJ. A number of ideas used in the later chapters of this thesis are based on extensions that were originally written for AspectJ.

AspectJ was designed to have a certain level of compatibility to Java, that level of compatibility set the standard for compatibility concerning other aspect-oriented extensions to modern programming languages, like AspectC# (see section 3.4.3) and AspectC++ (see section 3.4.4). AspectJ is compatible to Java in four ways [13, 18]:

Upward compatibility

All legal Java programs must be legal AspectJ programs.

Platform compatibility

All legal AspectJ programs must run on standard Java virtual machines.

Tool compatibility

It must be possible to extend existing tools to support AspectJ in a natural way, this includes IDEs, documentation tools and design tools.

Programmer compatibility

Programming with AspectJ must feel like a natural extension of programming with Java.

AspectJ supports two types of cross-cutting implementation. First, there is *dynamic cross-cutting*, which allows to advice a program at well defined *regions-in-time* [20]. Those regions-in-time are called `before()`, `after()` and `around()`, which are fairly self-explanatory [18]. The term regions-in-time comes from the uncertainty concerning aspect ordering in many instances, while advice code is always executed in a specific order that order might not be directly defined by the programmer and especially in cases where multiple aspects advice a single joinpoint the order of execution might not be obvious to the programmer.

Second, there is *static cross-cutting*, which allows to define new operations on existing types in the case of AspectJ that means mostly classes but also methods and relationships between classes [13].

AspectJ supports a number of *pointcuts*, which are expressions used in aspects to determine the joinpoints a given advice is used on. The built-in pointcuts of AspectJ include classes, methods, function parameters, method calls and relationships between classes, among others.

3.4.2.1 Metadata Annotations and AspectJ

After the introduction of the Java Metadata Facility in Java version 5.0 the AspectJ language was extended to support the use of metadata in different ways. The most basic usage in the AspectJ language concerns the definition of aspects, advice and pointcuts with the help of metadata. Figure 3.3 shows how annotations can be used to define aspects instead of the standard AspectJ syntax.

This feature in itself is hardly interesting as it only changes the syntax used in AspectJ and does not actually enhance the expressiveness of the language.

AspectJ also supports the use of annotations in pointcut expressions. The matching of annotations to objects is performed by allowing so called annotation patterns into pointcut expressions.

Those patterns also support the operator `!` for negation and simple concatenation as shown in figure 3.4. The first annotation would match any annotated element which has an annotation of type Immutable. The second annotation would match any annotated

(1a) Aspect definition without annotations

```
1 public aspect Foo {}
```

(1b) Aspect definition with annotations

```
1 @Aspect
2 public class Foo {}
```

(2a) Pointcut definition without annotations

```
1   pointcut anyCall() : call(* *.*(..));
```

(2b) Pointcut definition with annotations

```
1   @Pointcut("call(* *.*(..))")
2   void anyCall() {}
```

(3a) Advice node definition without annotations

```
1   before() : call(* org.aspectprogrammer..*(..)) && this(Foo) {
2     System.out.println("Call from Foo");
3   }
```

(3b) Advice node definition with annotations

```
1   @Before("call(* org.aspectprogrammer..*(..)) && this(Foo)")
2   public void callFromFoo() {
3     System.out.println("Call from Foo");
4   }
```

Figure 3.3: Using Java annotations for declaring aspects, advice nodes and pointcuts

element which does not have an annotation of type Persistent. The following annotation matches any annotated element which has both an annotation of type Foo and an annotation of type Bar. The fourth annotation matches any annotated element which has either an annotation of a type matching the type pattern (Foo || Bar). In other words, any annotated element with either an annotation of type Foo or an annotation of type Bar (or both). And the last annotation matches any annotated element which has either an annotation of a type matching the type pattern (com.foo..*). In other words, any annotated element with an annotation that is declared in the com.foo package or a sub-package.

Those patterns enhance the expressiveness of the AspectJ pointcut language by allowing further distinguishing characteristics to be defined for annotated elements but they only support elements that can already be addressed with AspectJ and do not increase the number of those elements.

Utilizing those annotation patterns in pointcuts is shown in figure 3.5. The first ex-

```
1 @Immutable
2
3 !@Persistent
4
5 @Foo @Bar
6
7 @(Foo || Bar)
8
9 @(com.foo..*)
```

Figure 3.4: Annotations for pointcut expressions [21]

pression matches any join point where the code executed is declared in a type with an @Secure annotation. The format of the within pointcut designator in AspectJ 5 is 'within' '(' OptionalParensTypePattern ')'. The second expression matches a staticinitialization join point of any type with the @Persistent annotation. The format of the staticinitialization pointcut designator in AspectJ 5 is 'staticinitialization' '(' OptionalParensTypePattern ')'. The following expression matches to any call to any method with the @Oneway annotation. The last expression matches against the execution of any public method in a package with prefix org.xyz, where the method returns a result that is annotated with the @Immutable annotation.

The last example also demonstrates the only use for type annotations in combination with AspectJ currently. They can be used instead as a replacement for the type patterns normally used in pointcut expressions at this place.

```
1 within(@Secure *)
2
3 staticinitialization(@Persistent *)
4
5 call(@Oneway * *(..))
6
7 execution(public (@Immutable *) org.xyz..*.*(..))
```

Figure 3.5: Annotations as parts of pointcuts in AspectJ [21]

3.4.2.2 Extensions to the AspectJ Joinpoint Model

AspectJ is supported by a huge community and a number of researchers have examined ways to extend the joinpoint model to allow advisement of smaller units than methods or method calls. Two of those works have relevance for this thesis.

LoopAJ [1] is a language extension for AspectJ which allows the advise of Java loops inside of functions, it introduces a number of new pointcut expression which can be used to match against loops and access the variables used in the loop in the advice code. LoopAJ though is fairly limited as the code-analysis used only supports a small subset

of possible Java loops. It also does not support nested loops in any way, which is a significant disadvantage considering how important such loops are in many circumstances to program performance [22].

Closure Joinpoints [23] is an AspectJ extension to allow the application of advice on block statements. The paper is language specific for Java as it is the only language to support the closures mechanism described within. Nonetheless it also provides a comprehensive analysis of problems and challenges that have to be overcome when advising local blocks in an AOP language. That analysis is quite language independent and can be applied to this thesis.

3.4.2.3 Metadata Annotations and Extensions of the AspectJ Joinpoint Model

Research into the usage of metadata annotations for extending the AspectJ joinpoint model has been conducted in *Statement Annotations* by Marc Eaddy [24]. They utilized metadata annotations on a statement level inside of methods, as a new type of pointcut which could be advised like any other AspectJ joinpoint. In addition the standard Java metadata annotations syntax was used, which also allowed for the propagation of parameters into the advice code by binding a context variable from the advice to a value in the metadata annotation class. Figure 3.6 shows a short and simple example from that paper.

Using statement annotations to expose interesting events

```
1 ... several statements ...
2 @Note("Searching for plugins")
3 ... several statements ...
4 @Note("Entering very long, but hopefully not infinite, loop")
5 while(true) { ... }
6 @Note("Loop exited successfully")
```

Aspect for logging notes

```
1 aspect LogNotesAspect {
2   before(Note noteAnnotation) : @annotation(noteAnnotation) {
3     System.out.println(noteAnnotation.value() + " [" + thisJoinPoint + "]");
4   }
5 }
```

Figure 3.6: Example of Statement Annotations in AspectJ [24]

3.4.3 AspectC#

AspectC# [25] is an aspect-oriented extension of the C# programming language. AspectC# is quite different from AspectJ in that there are no syntax extensions for advice generation, as a matter of fact AspectC# does not do source code weaving to apply advice. AspectC# itself is just a number of C# classes that use custom annotations to

integrate advice code into base code.

The basic syntax of AspectC# is similar to the alternative syntax in AspectJ that allows aspect and advice definition with the use of metadata annotations. The advice code in AspectC# is not statically weaved into the application at compile time, instead a dynamic framework is used to execute advice code based on the information found in those annotations at runtime. It is an interesting approach that allows to use aspect-oriented concepts in C# without extending the programming language itself, instead it just requires the use of the AspectC# library to extend the behavior of the program. For example without including the runtime library it is still possible to compile and run an AspectC# application. The annotations that define the aspect behavior are then simply ignored and only the base code is executed.

Figure 3.7 shows a part of the runtime library code used in AspectC#. This particular code is used to collect the advice code from `before` and `after` advices and add them to the execution of the method dynamically at runtime.

```
1  MethodJoinPoint methodJP = hasAdvice(c.getNameAndParameters());
2  if(methodJP.hasAdvice()) {
3    if(before) {
4      foreach(Advice beforeAdvice in beforeList) {
5  AspectBuilder aspectAdvice = beforeAdvice.GS_Aspect;
6  string codeBefore = aspectAdvice.getAspectCode(beforeAdvice.GS_methodName);
7  CodeExpressionStatement expressionBefore = new CodeExpressionStatement(new
       CodeSnippetExpression(codeBefore));
8  method.Statements.Add(expressionBefore);
9      }
10   }
11   string code = c.getFinalText();
12   CodeExpressionStatement expression = new CodeExpressionStatement(new
         CodeSnippetExpression(code));
13   method.Statements.Add(expression);
14   if(after) {
15     foreach(Advice afterAdvice in afterList) {
16 AspectBuilder aspectAdvice = afterAdvice.GS_Aspect;
17 string codeAfter= aspectAdvice.getAspectCode(afterAdvice.GS_methodName);
18 CodeExpressionStatement expressionAfter = newCodeExpressionStatement(new
       CodeSnippetExpression(codeAfter));
19 method.Statements.Add(expressionAfter);
20     }
21   }
22   Class.Members.Add(method);
23 }
```

Figure 3.7: Code excerpt from the AspectC# runtime library [25]

3.4.3.1 Extensions to the AspectC# Joinpoint Model

Work into extending the AspectC# joinpoint model has been done by Walter Cazzola [26] where based on his own annotation mechanism [9] AspectC# was extended to allow the direct advice of annotations on a statement level to allow the advice of code inside

a function body without the need to refactor the code for this purpose as is a common workaround in aspect development. Figure 3.8 shows such an annotation as used in a recent paper [26].

```
1 public ProceduralWall() {
2   [LoadData(Target = Polyline)] {}
3 }
```

Figure 3.8: Example of AspectC# metadata annotations [26]

3.4.4 AspectC++

AspectC++ [27] is an aspect-oriented extension of the C++ programming language developed by Olad Spinczyk and others. It is also the target language for the extensions presented in this thesis. AspectC++ was created to utilize the AOP paradigm in the realm of embedded system software, where code-size, performance and numerous other requirements prevent the use of languages like Java. Even C++ is not as widely used as it may be should as C is still the dominant language in embedded system software development. AspectC++ is closely related to AspectJ in the way advice is defined, pointcut expression are shaped and how the joinpoint model is constructed.

Of course having C++ as the base language instead of Java also leads to significant differences. Nonetheless AspectC++, like AspectJ, supports dynamic as well as static advice. Where static advice is done utilizing so called `slices` to extend existing or create new classes with aspects. Dynamic cross-cutting advice is realized with `before()`, `after()` and `around()` advice on joinpoints which are defined using pointcuts, like in AspectJ.

For this purpose AspectC++ supports a number of built-in pointcut functions, a full summary of which is available in the AspectC++ Language Reference [28]. Furthermore AspectC++ supports dynamic runtime advice using a `cflow` pointcut function. The next chapter will provide an extensive analysis of the AspectC++ joinpoint model and pointcut language.

The example for the AOP approach earlier in this chapter was written in AspectC++ and can be seen as this sections example as well (see figure 3.2).

3.4.5 Other

LogicAJ2 [29] is an aspect-oriented language which was developed at the University of Bonn. While it is related to AspectJ it has a very different joinpoint model and pointcut language. What makes it quite relevant in the context of this thesis is its ability to

express fine-grained generic advice on every element of its base language Java. Furthermore it was also designed to allow the same expressiveness as LoopAJ without putting additional constraints on the structure of the base program.

The LogicAJ2 pointcut language only has three basic built-in pointcuts (see figure 3.9) nevertheless those basic pointucts can be combined to express all possible joinpoints including the same joinpoints that can be addresed with AspectJ and LoopAJ but also many more. The disadvantage of the pointcut language lies within the complexity of the pointcut expressions, furthermore the complexity does not in fact prevent the fragile pointuct problem [30].

- `decl(join_point, declaration_code_pattern)`
- `stmt(join_point, statement_code_pattern)`
- `expr(join_point, expression_code_pattern)`

Figure 3.9: Basic pointcus of LogicAJ2 [29]

For example the paper on LogicAJ2 demonstrate how to model the AspectJ `call` pointcut in LogicAJ2 (see figure 3.10), which shows how difficult and complex it is to express even simple AspectJ pointcuts in LogicAJ2.

```
1 pointcut call(?jp, ?declType, ??modifiers, ?returnType, ?name, ??parTypes):
2
3   expr(?jp, ?name(??args) ) &&
4   decl(?method, ??modifiers, ?returnType, ?name(??par) { ??stmts } ) &&
5   equals(?method, ?jp::ref) &&
6   equals(?declType, ?method::parent::type) &&
7   parameterTypes(??parTypes, ??par);
```

Figure 3.10: Implementation of the `call` pointcut [29]

Still it does give an impression on how expressive AOP languages could be beyond the current standards found in AspectJ and related languages.

4 Joinpoint Model Extension

The AspectC++ [27] joinpoint model is the abstract model that is used to determine where aspects can take effect. Any examination of the joinpoint model can not be performed independently from an in-depth examination of the pointcut language. As the pointcut language is used to determine where specific aspects take effect in a project. Limitations in the joinpoint model are often times mirrored as limitations in the pointcut language and should therefore be analyzed together.

This chapter starts with an analysis of the joinpoint model currently used in AspectC++ in combination with an analysis of the expressiveness of the pointcut language. Afterwards the limitations of the current mechanisms are examined, specifically those limitations that could be circumvent using metadata annotations, before extensions in the context of metadata annotations are proposed.

Extensions to the joinpoint model of any aspect-oriented language have to take into account the base language used and the unique properties of those languages. Java and C++ are related in the sense that both languages utilize the OOP pradigm but they are also very different in how that paradigm is realized. For example, Java does not support multiple inheritance, automatic type conversions, operator overloading, and pointers. Because of this blindly implementing the same extensions into AspectC++ that were used in AspectJ is not a recommended course of action without first motivating those extensions in the context of AspectC++.

4.1 Analysis of the AspectC++ Joinpoint Model

The AspectC++ joinpoint model not only holds information about the different types of joinpoints supported by AspectC++ but also supports modeling of the correlations between different joinpoints.

4.1.1 Joinpoints Types

The joinpoint model uses a number of types to distinguish between different kinds of joinpoints. Those types have their own inheritance structure. There are two basic types followed by a number of different subtypes which are explained in the following paragraphs.

4.1.1.1 Basic Joinpoint Types

The basic joinpoint types are name and code joinpoints. Name joinpoints are joinpoints that can be directly addressed by using match expressions, which are special types of strings in the pointcut language of AspectC++. Code joinpoints are always children of name joinpoints that are identified by using a combination of match expressions and pointcut functions in AspectC++. Every type of code joinpoint has a corresponding pointcut function in AspectC++. To access the code joinpoints that are children of a specific name joinpoint a match expression mapping to that name joinpoint is used as an argument for the pointcut function mapping to that specific code joinpoint type.

Figure 4.1 shows an example representation of source code as a joinpoint model in AspectC++.

4.1.1.2 Name Joinpoint Types

Name joinpoints are divided into three subtypes. Those are namespaces, classes and functions. The global namespace in C++ is used as the root of the tree representation in the joinpoint model. Namespaces can only be children of other namespaces, where classes and functions can only be children of namespaces and classes. The C++ language does support exceptions from these rules, for example local classes can be defined inside of functions but the joinpoint model does simply ignore those exceptions.

4.1.1.3 Code Joinpoint Types

Code joinpoints are children of name joinpoints, they are divided into four subtypes. These subtypes are execution, call, construction and destruction. Currently code joinpoints are only used as children of function joinpoints. Every function, which is part of the project, has one execution joinpoint. Every call made to a function from another function in the program code is represented as a call joinpoint child of that function which points to the name joinpoint of the called function. Construction and destruction joinpoints are children of constructor and destructor functions of classes respectively.

Call functions are special in AspectC++ as they not only refer to so called call expressions but are also generated for operators on data types that have overloaded operator functions. This in particular will be examined later in this chapter.

4.1.2 Pointcut Language Expressiveness

Expressiveness in the context of the AspectC++ pointcut language refers to the ability of the language to identify specific joinpoints. AspectC++ has a number of built-in pointcut functions.

Match expressions are not just simple strings that are only used to identify specific joinpoints. Match expressions also support wildcard characters that allow to identify

(a) Example source code

```
1  void debug();
2
3  namespace NS {
4    class TestClass {
5    public:
6      TestClass() { /* constructor */ }
7      ~TestClass() { /* destructor */ }
8
9      void function() { debug(); /* call to debug() */ }
10   };
11 }
12
13 void debug() { /* debug function body */ }
14
15 int main() {
16   NS::TestClass a;
17   a.function(); // call to TestClass::function
18   return 0;
19 }
```

(b) Example joinpoint model

```
1  <namespace id="1" sig="::">
2    <namespace id="2" sig="NS">
3      <class id="3" sig="NS::TestClass">
4        <function id="4" sig="NS::TestClass::TestClass()">
5          <construction id="9"/>
6        </function>
7        <function id="5" sig="NS::TestClass::~TestClass()">
8          <destruction id="10"/>
9        </function>
10       <function id="6" sig="void NS::TestClass::function()">
11         <exec id="11"/>
12         <call id="19" target="7">
13         </call>
14       </function>
15       <function id="13" sig="NS::TestClass::TestClass(const NS::TestClass &amp;)" builtin
             ="1">
16         <construction id="14"/>
17       </function>
18     </class>
19   </namespace>
20   <function id="7" sig="void debug()">
21     <exec id="15"/>
22   </function>
23   <function id="8" sig="int main()">
24     <exec id="17"/>
25     <call id="20" target="6">
26     </call>
27   </function>
28 </namespace>
```

Figure 4.1: Example of AspectC++ joinpoint model types

multiple joinpoints based on patterns with them. Figure 4.2 (a) shows a number of match expressions which utilize those wildcards. For a more detailed describtion the AspectC++ language reference [28] is recommended.

Pointcut functions are not only used to identify code joinpoints. For example the `within` pointcut functions is used to select every joinpoint that is a child of the joinpoint used as an argument for the pointcut function. Furthermore arguments to pointcut functions are not only limited to match expressions but can be fully qualified pointcut expressions including pointcut functions themselves. Figure 4.2 (b) demonstrates this. The pointcut expression has two parts only joinpoints that are part of both sets of joinpoints specified are advised by the expression. That shows another feature of the pointcut language in that it allows binary operations on sets of joinpoints.

Class joinpoints for example hold additional information concerning the inheritance structure of the class, more specifically they hold references to the base classes and the derived classes of the class. To utilize that there are two pointcut functions available, `derived` and `base` which in combination with match expressions can be used to select class joinpoints based on the inheritance structure present in a program. Figure 4.2 (c) shows an example pointcut expression that would apply the around advice specified there to all constructors of all classes that are derived from the classes “Parent1” and “Parent2”.

All examples so far have utilized static pointcut functions that are examined before the execution of a program. AspectC++ also supports the use of dynamic pointcut functions in expressions. Those pointcut functions are evaluated at runtime and advice is applied based on dynamic checking against the requirements. An example of such a function is `cflow`. Without going much into detail, the function is used to dynamically check if the control flow took a specific path through the static call graph of the program. For example, figure 4.2 (d) uses this pointcut function to apply advice on any function execution that is part of the control flow of all calls that match against the match expression “% ...::target_function()” in this instance.

This section is not intended designed to provide a complete presentation of the expressiveness of the pointcut language in AspectC++, but shall provide the reader with a basic understanding of the language elements provided by it. The next section examines a number of limitations found in the current model.

4.2 Joinpoint Model Limitations

The pointcut language in AspectC++ does support complex pointcut expressions as shown in the last section. Nonetheless, a number of limitations are present for identifying unique joinpoints in the joinpoint model. The limitations evaluated in this thesis are centered around the prospect that they can be overcome or circumvented utilizing

(a) Examples of match expressions

```
"void debug()"              // matches the specific function debug
"% NS::TestClass::%(...) " // matches against all functions in the TestClass class
"% ...::%(...)"           // matches against all functions in all scopes
```

(b) Targeting specific function calls

```
advice call("% ...::target_function()") && within("% specific_scope::%(...)") : after() {
  <statements executed here>
}
```

(c) Targeting constructors of a multiple inheritance class

```
advice construction(derived("Parent1") && derived("Parent2")) : around() {
  <statements executed here>
}
```

(d) Targeting all function executions in the control flow of a specific call

```
advice cflow(call("% ...::target_function()")) && execution("% ...::%(...)") : around() {
  <statements executed here>
}
```

Figure 4.2: Example of AspectC++ match expressions and pointcuts

metadata annotations.

4.2.1 Limits for Advice of Specific Joinpoints

The two most fine-grained units of separation in the joinpoint model are functions and function calls. For example, figure 4.3 (a) shows a function that has two different calls to the same function within its function body, with the current joinpoint model there is no way to advice only one of the calls, the only possibilities are to advice all or none of the function calls and nothing in between.

This comes back to the granularity of the name joinpoints in AspectC++. The calls can only be addressed by using a match expression on the function signature in combination with the `call` pointcut function. For reducing the set of joinpoints further the `within` pointcut function is used, but the smallest unit that can be used there is the parent functions name.

To resolve this limitation in the current model a mechanism would be needed to uniquely identify specific function calls in addition to the current mechanism provided.

4.2.2 Limits for Advice Execution

Furthermore there is no support for directly applying advice at specific locations inside a function body. There are two ways to apply advice code to a function in the current joinpoint model. First advice can be applied to the execution joinpoint either `before`, `after` or `around`. The only way to apply advice *inside* of a function is to use a call joinpoint. When no call joinpoint is available for this or when no uniquely identifyable call joinpoint is available (see above) the current "best-practice" in AOP is to create a function with the only purpose of creating a new joinpoint.

The created function depends mostly on the advice code that has to be applied through it. Sometimes the function needs arguments to allow the advice code access to certain context information, e.g local variables. Sometimes the function is empty while other times it executes parts of the original function statements so that the advice can be applied at the intended position during the programs execution. Figure 4.3 (b) shows an example of such a function. Those wrapper functions itself have to be carefully tailored so that the program behavior is not changed. Furthermore other advice code has to be carefully examined, to ensure that the new function is not affected by other advice code. Figure 4.3 (c) shows an example of this. Yet there are more problems involved with this solution.

Advice is supposed to implement cross-cutting concerns. This implementation should essentially be done oblivious with regards to the underlying base code. Refactoring the base code should not be necessary to advice its behavior but in the current joinpoint model design it really is the "best-practice" available, that not only applies to the AspectC++ joinpoint model, similar problems have become apparent for AspectJ and other AOP languages as well.

To resolve this limitation new joinpoint types for advice inside the execution of functions are needed. These problems with regards to AspectC++ have been documented before, for example the analysis of aspects in the eCos kernel [31] noted some of them. The solution back then was the development of a new operating system designed with aspects in mind from the bottom-up with CiAO [32], which allows to avoid those problems during development of the software system. In this thesis the problems are not avoided but instead possible solutions are examined in the next sections.

4.3 Requirements for Joinpoint Model Extensions

AspectJ demonstrates how metadata annotations can be used to identify joinpoints by using additional identifiers to the ones provided by the base language. This thesis introduces metadata annotations into AspectC++ in the next chapter. In addition this thesis will introduce new joinpoint types to further extend the expressiveness of the AspectC++ language.

(a) Function with multiple equal calls within

```
1 void Model::process() {
2   this->elements.sort();
3   list<Elements*>::iterator it;
4   for(it = this->elements.begin(); it != this->elements.end(); it++) {
5     <change elements>
6   }
7   this->elements.sort();
8 }
```

(b) Workaround for advising code which is part of a function

```
1 void Parent::function() {
2   <statements>
3   <statements which shall be advised>
4   <more statements>
5 }
```

```
1 void Parent::function() {
2   <statements>
3   wrapper_function(<wrapper_function_args>);
4   <more statements>
5 }
6
7 void Parent::wrapper_function(<wrapper_function_sig>) {
8   <statements which shall be advised>
9 }
```

(c) Tracing aspect with changed behavior after workaround for (b)

```
1 aspect Tracing {
2   advice execution("% Parent::%(...)") : around() {
3     cout << "TRACE: enter " << tjp->signature() << endl;
4     tjp->proceed();
5     cout << "TRACE: leave " << tjp->signature() << endl;
6   }
7 };
```

Figure 4.3: Example of AspectC++ advice limitations

Metadata annotations provide means to identify a joinpoint by annotation in addition to a match expression or combination of pointcut functions currently found in the AspectC++ language. In addition metadata annotations can be applied to language elements that are not identifiable using match expressions, which allows for the expression of joinpoints in the pointcut language that are not currently supported.

This in turn allows for joinpoint types to be defined without having to consider ways to address them directly by using match expressions for example. These joinpoints would not require a unique signature to be addressed using metadata annotations. Joinpoints that fall into that category are called explicit joinpoints as they need to be explicitly annotated to be addressed. This removes a restriction from the design of new joinpoint

types in comparison to the old joinpoint types present in the current model.

The next section examines joinpoint model extensions used in the AspectJ language in the context of AspectC++. As AspectC++ is not AspectJ all extensions have to meet a number of requirements to be considered for implementation into AspectC++ in the following chapters.

Genericity

The new joinpoint types should apply for generic language elements of the C++ base language used in AspectC++. That means they should be present in a wide variety of programs without being overly specialized in the context of the base language.

Expressiveness

The new joinpoint types have to extend the expressiveness of the AspectC++ pointcut language, that means they must allow for more fine-grained advice execution in AspectC++. That means they must allow to execute advice code at places that can not traditionally be advised without using workarounds or refactoring of source code.

4.4 Extending the Joinpoint Model

This section describes the new joinpoint types introduced in this thesis. How they fit into the existing joinpoint model structure and why they provide a desirable extension of the current joinpoint model.

4.4.1 Statement Level Advice

There are several types of statements in the context of AspectC++. The C++ language supports statements in different forms as shown in figure 4.4. In this section those statement types are examined against the requirements listed in the previous section. Similar examinations have been done in the context of AspectJ and are also considered in this section.

4.4.1.1 Statement Level Advice in AspectJ

There are three papers that are relevant in the context of statement level in AspectJ with regards to this thesis. Statement level annotations (see chapter 3 section 3.4.2.3) are interesting in the context of metadata annotations. The LoopAJ [1] language extension provides joinpoints for loop-statements, although the for loops themselves have to meet restrictive requirements that are independent from the syntactic possibilities provided by the Java language. Such restrictions in the context of AspectC++ would violate the

```
1  LABEL: statement;     // labeled-statement
2
3  call_to_function();   // expression statement
4
5  if ( cond )           // condition statement
6     statement;
7  else
8     statement;
```

Figure 4.4: Examples of statements in C++

requirements mentioned above.

The most interesting concept is expressed in the paper "Closure Joinpoints" by Bodden et al. [23]. The paper examined the execution of advice on block statements. It includes an extensive analysis of problems that had to be solved in order to allow that kind of advice. The actual solution postulated in that paper is not quite as interesting in the context of AspectC++ as it utilizes a language feature unique to the Java language, the so called closures.

4.4.1.2 Genericity and Statement Joinpoints

Nevertheless, the concept of advising compound statements is quite interesting in the context of AspectC++. Compound statements are ubiquitous in most programs, they are used in combination with a number of different types of statements. This would already ensure that block joinpoints would meet the genericity requirement described above. Furthermore they allow to execute advice in the context of loop-statements and condition-statements, even the advice on try-blocks would be enabled with the addition of this joinpoint type.

Furthermore block joinpoints should be considered before considering more specialized joinpoint types based on loops, if/else clauses or even exception handling constructs in C++. Block joinpoints might provide a level of expressiveness for the AspectC++ language that makes further specialized joinpoint types less desirable.

4.4.2 Compound Statement Joinpoints

A compound statement joinpoint or short a block joinpoint is a joinpoint applied to a compound statement, which is often also called a block statement. A compound statement is a C++ language element that denotes a local semantic scope. This means any declaration within the block statement is only valid until the end of the block statement and not further. Compound statements can be used as stand-alone blocks inside of a function, but most often they are part of other statements as shown in figure 4.5.
In any given C++ source code there are already a high number of blocks present. So

```
1  // stand-alone block
2  { <statements> }
3
4  // blocks as part of if/else-if/else statements
5  if (<condition>) {
6    <statements>
7  } else if (<condition>) {
8    <more statements>
9  } else {
10   <even more statements>
11 }
12
13 // a block as part of a for loop
14 for (int i = 0; i < n; i++) {
15   <statements>
16 }
17
18 // a block as part of a switch statement
19 switch(v) {
20 case 1:
21   <statements>
22   break;
23 case 2:
24   <statements>
25   break;
26 default:
27   <statements>
28   break;
29 }
```

Figure 4.5: Example of compound statements in C++

creating a joinpoint type for blocks does fulfill the genericity requirement introduced in the last section. A complete list of statements that utilize blocks can be found in the C++ language reference [33]. Furthermore it does allow the advice of loop bodies and other generic language structures of the C++ language indirectly. Therefore it is possible for a well-designed block joinpoint mechanism to allow advice in a comparable fashion to the LoopAJ [1] mechanism as mentioned above.

4.4.2.1 Problems

An extensive analysis has been conducted into utilizing block joinpoints in the context of AspectJ [23]. Based on this analysis block joinpoints pose a number of problems for advice generation. For introducing a new joinpoint type into AspectC++ those problems have to be solved. In addition the exact behavior of `before`, `around` and `after` advice has to be defined in relation to the new joinpoint type. It is similarly important to define the exact behavior of the `JoinPoint` API of AspectC++. A good way to start designing such behavior is an examination of a similar joinpoint type and comparing it to the semantic behavior of the new joinpoint type. The most similar joinpoint to blocks is the execution joinpoint already present in the AspectC++ language.
A function is a group of statements that are executed when the function is called from some point in the program. To simplify functions lets consider the function arguments as the input data and the function return value as its output data. But that is not always

JoinPoint	Function	Block
types:		
`Result`	result type	-
`That`	object type	object type
`Target`	target type	-
`AC::Type`	encoded type of an object	type of an object
`AC::JPType`	joinpoint types	joinpoint types
static methods:		
`int args()`	number of arguments	-
`AC::Type type()`	type of the function	-
`AC::Type argtype(int)`	types of the arguments	-
`const char *signature()`	signature of the function	-
`unsigned id()`	joinpoint identification	joinpoint identification
`AC::Type resulttype()`	result type	-
`AC::JPType jptype()`	type of joinpoint	type of joinpoint
non-static methods:		
`void *arg(int)`	actual argument	-
`Result *result()`	result value	-
`That *that()`	object refered to by this	object refered to by this
`Target *target()`	target object of a call	-
`void proceed()`	execute joinpoint code	execute joinpoint code
`AC::Action &action()`	Action structure	Action structure

Figure 4.6: Comparison between function and block joinpoint behavior

true in the context of C++, functions as members of a class can access and modify other class members and arguments, also be pointers or variable references which allow the function to output information through them can be supplied as arguments. Like blocks functions declare their own semantic scope, that means any variable locally defined in the function is only valid until the end of the function body.

When comparing blocks to functions, the first step is to determine the blocks input and output data. Blocks do not have directly defined arguments, but blocks can, like functions, access and modify class member variables. Furthermore, blocks can access and modify all variables present in their parent scope. When comparing blocks to functions it is also important to notice that blocks are only executed at exactly one point in the program code which is marked by the location of the block statement.

For the definition of the behavior of the `JoinPoint` API utilized in AspectC++ it is important to realize that blocks only have a limited use for the functions defined in the API in comparison to functions. For example, there is no `target()` in the context of the `JoinPoint` API concerning block joinpoints as there are no calls to the

block joinpoint at any point in the program. There is no need to convey this kind of object type. The alternative to that behavior is to return the same value for `target` as for `that` but as mentioned before this would be contrary to the actual behavior of blocks.

Figure 4.6 shows a comparison table between the behavior of functions and blocks concerning the `JoinPoint` API. The next paragraph describes how block joinpoints are placed in the joinpoint model and the paragraph thereafter examines how differences to functions have to be incorporated when advising a block joinpoint.

4.4.2.2 Integration into the Joinpoint Model

Block statements unlike classes or functions do not have a unique identifier in the C++ language. This makes block joinpoints explicit joinpoints as described previously. Like a function joinpoint it has a child of type execution joinpoint to allow advice utilizing the built-in `execution` pointcut function. Futhermore it will be used as a container for other joinpoints during the creation of the joinpoint model to allow the use of the built-in `within` pointcut function in combination with other joinpoints.

4.4.2.3 Advice

Because of the special interactions between the block scope and its parent scope during execution a number of problems for `around` advice are present. Functions can easily be called from within the `proceed` function of the joinpoint API because all necessary information can be provided at call time. The problem lies in the fact that the code in the body of a block might need access to local variables from the parent scope, whether that parent scope is a function or even just another block. In the case of another block, it is even more complicated as the block might access local variables of all other blocks it is a child off. In regards to AspectJ [23] this poses a large problem, but in regards to AspectC++ it is less complicated as one might expect as will be shown in this section.

Executing the code of a block outside of its original parent scope requires the presence of all variables used inside the body of the block, furthermore to ensure correct execution semantic after the execution of the block changes to non-local variables inside the block have to be applied to the variables inside their respective scopes. Those variables can be separated into several cases:

(1) Global variables or functions:

> Those do not pose a problem as they are globally defined and can be accessed from outside the parent scope of the block.

(2) Class member variables or functions:

The AspectC++ joinpoint API provides a mechanism for accessing the `this` pointer of a joinpoint, therefore access problems for class members can be circumvented here.

(3) Pointer variables:

Pointer variables have to be separated into two cases again, first the block only uses the pointer to access the object and does not change the target address of the pointer itself and second the block does actually change the target address of the pointer. Only in the second case the changes on the pointer variable have to be applied to the local pointer variable inside the parent scope to ensure correct execution of the program.

(4) Local variables:

Local variables similarly have to be written back when changes on them occur in the context of the block scope.

After further consideration it was decided that all variables used inside a block have to be consistently updated in their parent scope to simplify the source code analysis that has to be conducted in this context.

4.4.2.4 Improved Expressiveness

The second requirement for joinpoint extensions mentioned above, was an improvement of the expressiveness of the AspectC++ language. The question is whether or not block joinpoints provide such an improvement.

Block joinpoints provide an AspectC++ developer with an implicit mechanism to advice part of a function body by encapsulating that part in a block statement, why such an encapsulation might require some refactoring of the code it still simplifies the process in comparison to the “best-practice” mechanism described above.

Block joinpoints provide an additional level of granularity for the `within` pointcut function in AspectC++. This allows an AspectC++ developer to select call expressions more distinctively by utilizing this feature. As an example figure 4.7 shows an annotated block joinpoint and how a call joinpoint can be uniquely identified with the help of the block joinpoint.

4.4.2.5 Conclusion

Block joinpoints are an interesting alternative to joinpoints based on a specific statement-type (if/else, for, while, switch, etc.) as has been used in AspectJ in the past. The number of problems faced in comparison to block joinpoints in Java [23] are smaller in

```
1  void SomeDataType::calculation_procedure() {
2    ...<some statements>...
3    generic_function(<parameter>);
4    ...<some statements>...
5    @check {
6      generic_function(<parameter>);
7    }
8    ...<some statements>...
9    generic_function(<parameter>);
10   ...<some statements>...
11 }
```

```
1  aspect AdviceSpecificCall {
2    advice call("void ...::generic_function(...)") && within(execution(@check)) : after()
         {
3      ...<interesting advice code>...
4    }
5  };
```

Figure 4.7: Example of improved expressiveness through a block joinpoint

AspectC++. This makes implementation of block joinpoints a logical step for extending the AspectC++ joinpoint model.

4.4.3 Type Annotations in AspectC++

Metadata annotations in the Java programming language can be applied to types. As explained in the last chapter that feature of the annotation mechanism is only of limited use in the AspectJ programming language. Furthermore even the type checking based on predefined annotations is not even part of the standard Java compiler yet and requires the use of a specialized compiler.

For metadata annotations in AspectC++ the feature of using annotations for type checking is not part of this thesis. The main focus of this thesis is the extension of the AspectC++ joinpoint model with regard to more fine-grained advice *inside* of functions. Type checking can not be used to improve the application of advice or the implementation of cross-cutting concerns in the context of AspectC++.

While utilizing type annotations is an interesting concept for AspectC++ with regard to type checking or different mechanisms. The concepts are well placed outside the scope of this thesis. Already in the course of this thesis a number of extensions will be implemented into AspectC++. The complete mechanisms for metadata annotations that will be explained in the next chapter, as well as block joinpoints as described above.

Type annotations and their possible use in AspectC++ pose an interesting problem for future work in this area.

4.4.4 Expression Level Advice

Expressions refer to a wide variety of language elements in the C++ language. One subset of those expressions are the call expressions which are already covered by an existing joinpoint type in the AspectC++ language. Limitations of the call expression joinpoint will be addressed in combination with the metadata annotations mechanism introduced in the next chapter.

Other expressions so far have been ignored in the context of the AspectC++ joinpoint model. After introducing a new advice type already on the statement level of C++ it is only a logical next step to at least consider even more fine-grained places in a programs source code where advice could be applied.

Consider for example the annotation of operators for applying advice to the execution of said operators. Already as part of the call joinpoint type the invocation of overloaded operator functions is part of the AspectC++ joinpoint model. In addition to support the annotation of any operator in this context, the joinpoint model could be extended to include function joinpoints for the built-in operators of standard types. This would allow the annotation of any operator inside a program for the application of advice. Where as in the current model, the possibility of such advice is dependent on the existence of overloaded operators in custom types.

On the subject of advice *inside* of functions there are no smaller language elements that have clearly defined semantic behavior beyond the operators mentioned above. Even LogicAJ2 [29] which is to date the most fine-grained aspect-oriented language available, does not allow for advisement on a more fine-grained layer.

5 Metadata Annotations in AspectC++

This chapter introduces metadata annotations into AspectC++. It provides the definition of syntactic and semantic rules that govern metadata annotations and are used to apply annotations to generic language elements. Metadata annotations in the context of AspectC++ are used to provide additional means to identify joinpoints on top of the current mechanisms present in the AspectC++ language.

5.1 Design Goals

Extending a programming language is not a simple task. Small changes can have unexpected repercussions and side effects that are not immediately obvious to the designer. Because of this any extension has to be carefully planned and must be designed under careful consideration of a multitude of factors.

The new joinpoint types introduced in the previous chapter are explicit joinpoints, which are joinpoints that do not have a unique signature which is used to identify such a joinpoint through a match expression. For this reason, explicit joinpoints benefit from metadata annotations as they allow the identification of these joinpoints without additional refactoring of the source code.

The analysis of the AspectC++ joinpoint model in the previous chapter (see section 4.2) showed that not only explicit joinpoints can benefit from metadata annotations. Therefore this chapter provides syntax and semantic for annotating the existing joinpoint types in AspectC++ as well.

5.1.1 Metadata Annotations and Obliviousness

Obliviousness in the context of AOSD refers to the fact that the base code does not need to be specially prepared to have advice applied to it. Aspect code can be managed independently from the base code and AOP languages support clear separation of those two types of source codes. Even the metadata annotation mechanisms used in AspectJ and AspectC# does comply with this separation. Metadata annotations are language elements of the Java and C# language used as base languages in those AOP implementations respectively. In the case of AspectC++ this is quite different. Unlike the other base languages, C++ does not support metadata annotations in any form. Extending

the AspectC++ language with a metadata annotations mechanism therefore means, that aspect code is no longer clearly separable from the basis code. A consequence of this for example is, that the basis code can no longer be compiled using a standard C++ compiler, which is entirely possible with the current AspectC++ language features, in the case that base code and aspect code are truly separated into different source files.

Metadata annotations intentionally break that behavior as those annotations will be directly applied locally to the corresponding language elements. That means that annotated source code can not be compiled using a standard C++ compiler any longer. That leads to a fundamental difference between the use of metadata annotations in AspectJ (or AspectC#) and AspectC++. Where as in the Java language metadata annotations are part of the semantic and pose no problem with regard to their presence inside of standard Java source code. The case is quite different for C++, as AspectC++ code is transformed into C++ code during the weaving process, and every metadata annotation has to be removed from the resulting source code. That means any all features that utilize metadata annotations are part of the AspectC++ language instead of the base language as in the case AspectJ.

That also leads to the fact that metadata annotations can only be used in a static context at compile time and not dynamically at runtime because at that point their is only pure C++ program left, and C++ does not support metadata annotations in any form.

5.2 Design Rules for Metadata Annotations

Based on those facts metadata annotations must follow a number of rules in the context of AspectC++. Those rules are in part based on the implementations of similar annotation mechanisms in Java and C#, as those features have shown themselves to be quite useful for metadata annotation mechanisms, these are present in the rules (1-3). In addition there are objects that might be annotated in future versions of the metadata annotation mechanism as explained at the end of the previous chapter, it is therefore a non-functional requirement to annotate elements in such a way that future extensions could be integrated without the need to change syntax or semantic rules developed in context of this thesis as described in rule number (4). Furthermore the weakest requirement concerning annotations is number (5) where a certain level of unified syntax for annotating language elements is proposed, the rules governing the placement of annotations for basic language elements should when possible have a degree of similarity.

(1) Location of metadata annotations must be unique for any language element

That means metadata annotations applied to an language element must only be found at exactly one place in the source code. This should minimize the consequences that emerge from breaking the separation of aspect code and base code.

(2) Allow multiple metadata annotations for language elements

Similar to the annotation mechanism in Java, multiple metadata annotations should be allowed for language elements that can be annotated.

(3) Allow metadata annotations to have parameters

Metadata annotations should allow the application of parameters. In the case of explicit joinpoints there are no directly defined input or output values which could be influenced by advice code, instead metadata annotations are used to provide values of interest for the advice code.

(4) Consider future extensions in the syntax development

A secondary objective for the syntax proposed in this chapter is to allow for later extensions concerning the annotation of additional even more fine-grained generic language elements, like type expressions for example.

(5) Similar syntax for all language elements

The syntactic rules for applying annotations to objects should be visually similar so that a unified structure concerning different language elements can be distinguished.

5.3 Syntax and Semantic of Metadata Annotations

This section establishes the syntactic and semantic rules that govern metadata annotations in the AspectC++ language. It is divided into three parts. In part one metadata annotations are introduced as a language element of AspectC++. Part two handles the syntax for applying metadata annotations to existing joinpoints, where as part three describes how metadata annotations are used in the contex of explicit joinpoints as discussed in the previous chapter.

5.3.1 Metadata Annotations as Language Elements

To simplify the development process for metadata annotations in AspectC++ it was decided to utilize a syntax for metadata annotations similar to the syntax used in the Java Metadata Facility. An alternative to this would have been to introduce a new keyword into the AspectC++ language to address the definition of annotations. While a valid

idea it was decided to use a syntax similar to Java to ensure that metadata annotations can be easily distinguished from other language elements in the AspectC++ language.

Similar to their use in the Java programming language, AspectC++ should allow the application of multiple annotations to an language element, which leads to the support for annotation sequences which are simple concatenations of annotations.

Furthermore similar to metadata annotations in the Java language, AspectC++ should allow the definition of parameters on annotations. This leads to the following syntax for metadata annotations in AspectC++:

annotation:
@*identifier* *annotation-param*$_{opt}$

annotation-param:
(*expression-list*$_{opt}$)

annotation-seq:
annotation
annotation-seq annotation

The identifier used in a metadata annotation has to be a valid identifier following the rules of identifiers of the C++ language, there are no further restrictions on those identifiers. Any identifier can be used and can be applied to any language element without restriction. The parameters for metadata annotations are optional, as are the round brackets used to encase the parameter list. From a semantic standpoint the parameter list is similar to the one used in postfix expressions that model function calls, but no semantic checks are performed on them at parse time.
An annotation sequence is a simple concatenation of annotations without any special syntax, but their is one general semantic rule that has to be applied. A semantic error has to be produced if the same annotation identifier is used more than once inside an annotation sequence. An number of examples of valid and invalid metadata annotations can be found in figure 5.1.

```
1 @foo                  // Annotation without parameters
2
3 @bar(i, v)            // Annotation with parameters
4
5 @foo @bar(1, "text")  // Annotation sequence
6
7 @foo @foo(i) @foo("text") // INVALID annotation sequence
```

Figure 5.1: Examples of valid metadata annotations in AspectC++

5.3.2 Metadata Annotations for Existing Joinpoints

After defining the syntax for the metadata annotations themselves, the next step in the design process is to define how metadata annotations are applied to language elements in the AspectC++ language. Metadata annotations are designed to be used in combination with joinpoints to add a new dimension of identification used in the pointcut language of AspectC++. This section describes how metadata annotations are applied to the joinpoint types of the original AspectC++ joinpoint model with the exception of namespaces where an explanation is provided for why metadata annotations are not applied to that specific language element.

5.3.2.1 Namespaces

Namespaces are a container object in the C++ language. Unlike most other objects in the C++ language they can be declared at multiple places scattered across numerous different source files (as is the case in most software projects utilizing C++). This means that using metadata annotates with namespaces has a number of problems.

First allowing the annotation of a namespace declaration would mean that annotations can be placed at multiple places in the source code and would therefore violate the first design rule mentioned above. Furthermore the meaning of such an annotation would not be clear as their are several possible meanings that could be perceived from an annotation on such a declaration. They could either be interpreted as annotations applied to the specific language elements specified within the context of this specific declaration. Or they could be interpreted as being valid for the whole namespace, that means every other namespace declaration anywhere else in the source code.

Namespaces are used as containers. Inside of them other declarations and definitions are placed in the C++ language. Their are two quite different ways annotations of namespaces could be interpreted should they be allowed:

Partial namespace annotation

> This means that an annotated namespace declaration is used as a simple way to apply the annotation to any language element defined inside the namespace. That are classes and functions in the context of the namespace and only classes and functions specifically declared within this specific declaration would be annotated. As those language elements are supposed to allow for metadata annotations themselves, this complicates the mechanism unnecessarily.

Complete namespace annotation

> The other way annotations on namespaces are interpreted is that the annotation is to be applied to the whole namespace itself. That allows for annotations on the same namespace to be at multiple places in the source code of a program. Furthermore it would complicated the verifiction of the semantic requirement that any object does not have the same annotation applied twice.

In addition namespaces themselve are already used as named containers for other language objects in C++. Adding other identifiers to them in the form of annotations has advantage from a developers perspective in AspectC++. Therefore it was decided that namespaces will not be among the language elements that metadata annotations can be applied to.

5.3.2.2 Classes

The next type of language elements to be considered are classes. Classes can have definitions in multiple files, for example forward declaration are regularly used in some header files. The C++ language only allows one class definition which is where annotations will be applied exclusively.

The syntax shown below was chosen for two reasons that will become apparent later on in this section. First as shown later, it resolves the requirement for a similar syntax across all objects. Secondly, it lessens the amount of effort that has to be applied to modify the C++ parsing process (in the later implementation). As a side note the original syntax used in the remainder of this section is based on the syntax trees found in the C++ standard [33]. The placement of metadata annotations in those syntax trees is emphasized by a bold typeface. This is for clarification purposes only and has no other meaning.

class-key:
`class`
`struct`
`union`

class-specifier:
class-head { *member-specification*$_{opt}$ }

class-head:
class-key identifier$_{opt}$ ***annotation-seq***$_{opt}$ *base-clause*$_{opt}$
class-key nested-name-specifier identifier ***annotation-seq***$_{opt}$ *base-clause*$_{opt}$
class-key nested-name-specifier$_{opt}$ *template-id* ***annotation-seq***$_{opt}$ *base-clause*$_{opt}$

Annotations on multiple classes can be considered as an additional dimension in the context of AspectC++. Where namespaces help to define and manage one dimension of separation between classes, an annotation can be used to define another and multiple

annotations can be used to define even more additional dimensions of separation between classes. This can help for the advice of multiple classes across several namespaces as an alternative to defining a pointcut variable.

Figure 5.2 shows two examples of annotated classes, of note is the placement of the metadata annotation after the identifier and before the base clause or the curly bracket respectively.

```
1 // an annotated class declaration
2 class Bar @foo("text") {
3   <member specification>
4 };
5
6 // an annotated class declaration with a base clause
7 class Foo @bar(i) : public Bar {
8   <member specification>
9 };
```

Figure 5.2: Examples for annotated classes in AspectC++

5.3.2.3 Functions

The next language element that should allow metadata annotations to be applied are functions. Functions similar to classes can be declared and defined at different places. For the placement of metadata annotations the definition is chosen. Annotations for functions allow similarly, as they do for classes the addition of new dimensions on the structure of functions independently from the separation performed by namespace and class declarations. This can improve the different ways separated concerns can be addressed with regard to the decomposition of the program without the need for declaring pointcut variables [34, 12]. This not only simplifies addressing of certain subsets of functions in a program but also prevents accidental invocations of side effects like the fragile pointcut problem [30].

function-body:

compound-statement

function-definition:

decl-specifier-seq$_{opt}$ *declarator ctor-initializer*$_{opt}$ ***annotation-seq***$_{opt}$ *function-body*

decl-specifier-seq$_{opt}$ *declarator function-try-block*

This is the part where the similarities in the syntax across different language elements concerning metadata annotations becomes apparent. The example in figure 5.3 demonstrates how similar the syntax for classes and functions is in practice.

5.3.2.4 Call Expressions

Call expressions are very interesting in the context of metadata annotations in AspectC++. The analysis conducted in the previous chapter showed clearly that the

```
1 // INVALID annotations on a function declaration
2 static void calculate_foo(int, int, int) @foo @bar;
3
4 // annotating a number of constructors of a class
5 class Foo {
6 private:
7   int _a_value;
8   bool _some_value;
9 public:
10  ...
11  Foo (int a) @bar() : _a_value(a), _some_value(false) { }
12  Foo (int a, bool s) @bar() : _a_value(a), _some_value(s) { }
13  ...
14 };
15
16 // annotating a function definition
17 static void calculate_foo (int i, int x, int y) @foo @bar {
18  ...
19 }
```

Figure 5.3: Examples for annotated functions in AspectC++

ability to identify specific call expressions can improve the expressiveness of the AspectC++ language. It is also something that can not be done under all circumstances in the current joinpoint model.

Call expressions are not limited to simple function calls. The C++ language supports operator overloading, which means that the usage of built-in operators also results in a call to a function. Therefore, when metadata annotations should be allowed for calls, they also have to be allowed for operators if the requirement for genericity is to be resolved. Genericity in this context refers to the ability to annotate any type of call with metadata annotations not only a subset of all possible calls.

Unlike the language elements discussed previously, calls have a complicated syntax. They are part of the postfix expressions defined in the C++ standard [33]. Because of this, the syntax tree is not part of this section, as it is quite extensive.

The syntax for call expressions in the sense of standard function calls is simple, the metadata annotation is simple supplied as a prefix to the call. For operators the same syntax is used and metadata annotations are simply placed as a prefix to the operator.

There are some cases of overloaded operators that pose a problem in this regard. Figure 5.4 shows a number of examples concerning the annotation of calls and operators. The problem with operators that have a suffix usage meaning ++ and −−. As the variable used as an argument for the operator and that argument itself can again be a call, it was considered best to forbid the usage of metadata annotations on those two operators completely.

The semantic of those annotations is actually very simple. They have to be completely

ignored after the initial parsing. Only when the AspectC++ compiler generates the joinpoint model those annotations become of interest, but this is explained in the next chapter.

```
1 // simple function call in statement form
2 @foo @bar call_to_function(<function_parameters>);
3
4 // function calls as part of other expressions
5 for (int i = 0; i < @foo list.size(); i++) {
6
7 }
8
9   // annotated operator
10  Date y = ...
11  ...
12  if(y @volatile == new Date()) {
13    ...
14  }
15
16  // special cases
17  @incr a.size()++; // annotates the call
18  @decr --a.size(); // syntax error
```

Figure 5.4: Examples of annotated call expressions in AspectC++

5.3.3 Metadata Annotations for Explicit Joinpoints

After defining how metadata annotations are to be applied to the language elements that are used as a basis for the AspectC++ joinpoint model, this section defines how metadata annotations are applied to explicit joinpoints.

5.3.3.1 Compound Statements

The block joinpoint type was newly introduced in the previous chapter. Block joinpoints are joinpoints over generic language elements, as also mentioned in the previous chapter. AspectC++ has no means to directly identify such a pointcut, as blocks and other generic language elements do not have unique identifiers. Metadata annotations though, allow to add identifiers to generic language elements to circumvent that limitation of the AspectC++ language.

The number of places where a block can be annotated are quite limited. Considering other C++ language elements there are only three possible choices as shown in figure 5.5. Solution (a) is similar in style to the annotation of functions and classes and the separation to other language elements is also clear. The problem with solution (b) is that after the annotation sequence, there is no explicit separator with regard to the following statements. Depending on the syntax for the annotation of statements, which includes expression statements and those can also include call expressions, therefore it would not be obvious which element is annotated by the sequence. The last possible

solution is similar to the way objects can be directly declared at the end of `struct` declarations, unlike the syntax from the `struct` type there is no semicolon at the end of the annotation sequence. Which would lead to the exact same problem already described concerning solution (b).

At the end it was decided to use the syntax shown in figure 5.5 (a) for metadata annotations on compound statements, as this syntax is also the most similar with regard to the syntax concerning other language elements defined previously in this chapter.

compound-statement:
annotation-seq$_{opt}$ { *statement-seq*$_{opt}$ }

statement-seq:
statement
statment-seq statement

(a) Before the opening curly bracket

```
1 @foo @bar {
2   <statements>
3 }
```

(b) Directly after the opening curly bracket

```
1 {
2   @foo @bar
3   <statements>
4 }
```

(c) After the closing curly bracket

```
1 {
2   <statements>
3 } @foo @bar
```

Figure 5.5: Possible locations for the annotation of compound statements

5.3.3.2 Outlook: Type Annotations

As mentioned in the design goals section of this chapter leaving room for future extension is a secondary objective for the integration process of metadata annotations into AspectC++. The syntax proposed in this chapter does not interfere with type annotations in any way. Assuming type annotations are used with a similar syntax as the one used in the Java annotations mechanism.

Figure 5.6 (a) shows a couple of examples in that regard. Going back to the annotation of classes and functions it could have been easily decided to place the metadata

(a) Clear separation between annotation types

```
1 @type void function_name(@otherType Foo x) @function {
2   ...
3 }
```

(b) Less clear separation between annotation types

```
1 @type void @function function_name(@otherType Foo x) {
2   ...
3 }
```

Figure 5.6: Examples of metadata annotations on types

annotations for those objects before the identifiers. As can be seen in figure 5.6 (b) this does not appear to be a good idea. While the syntax itself would still be uniquely defined and the meaning clearly identifiable, this syntax would not provide such a clear separation of the different types of metadata annotations used on those objects.

5.4 Metadata Annotations for Pointcut Expressions

After adding metadata annotations to generic language elements in AspectC++ in the previous section, this section is used to examine the integration of metadata annotations into the pointcut language of AspectC++, in such a form that the language can utilize the additional information on basic language elements provided by the metadata annotations in the context of applying advice.

This section focuses on design decisions behind the syntax extensions for metadata annotations in the pointcut language. A more detailed description of the actual mechanisms behind the implementation into AspectC++ is provided in the next chapter of this thesis.

5.4.1 Additional Design Goals

While the design goals used in the previous section continue to apply, additional concerns have to be considered with regard to an extension of the AspectC++ pointcut language. The first two concerns could be regarded as one combined concern but both are equally important that explicitly stating both here becomes necessary. In addition an additional concern is that advice code should be able to access and in the case of references even change the values of parameters given to annotations.

(6) No definition of additional pointcut functions

For utilizing explicit joinpoints and metadata annotations in the pointcut language of AspectC++ it should be avoided to create new pointcut functions. As a matter of fact it will be shown that new pointcut functions are not necessary for the joinpoints described in this thesis.

(7) Compatibility with existing pointcut functions

The existing joinpoint functions are more than enough to handle, in combination with metadata annotations, the identification of explicit joinpoints. Therefore the existing pointcut functions should allow metadata annotations instead of match expressions as parameters.

(8) Access to annotation parameters in advice code

Function joinpoints allow the access and even the modification of the function arguments in advice code, joinpoints with annotation should offer a mechanism to access specific annotation parameters like ordinary variables. This would also enable a programmer to use a reference as an annotation parameter the advice code could then perform operations on the parameters that would automatically be propagated back into the execution of the base code.

5.4.2 Basic Integration

The first step for integrating metadata annotations into the pointcut language is to define how metadata annotations can be used in pointcut expressions. Here it is useful to take a look at the use of metadata annotations in the AspectJ language again. AspectJ allows metadata annotations as expressions inside the pointcut language, furthermore binary operators like AND, OR or NOT can be applied to metadata annotations in pointcut expressions.

For AspectC++ the same features are quite interesting. Binary operators can be applied to match expressions and pointcut functions in AspectC++, therefore it is only logical to also allow their application to metadata annotations. In addition, design goal (7) described above requires metadata annotations to be valid arguments for pointcut functions. Furthermore binary operators in pointcut expressions should also allow the combination of metadata annotations and match expressions respectively pointcut functions.

5.4.2.1 Handling Annotation Parameters

Another problem that has to be solved when integrating metadata annotations into the pointcut language is how annotation parameters can be used in advice code. One possible solution would be to use a similar syntax for annotation parameters to the syntax

used for accessing function arguments through the `JoinPoint` API in AspectC++. Instead though another solution was chosen for this feature. Metadata annotations in pointcuts can be bound to advice arguments similarly to how function arguments can be bound. This allows for the use of annotation parameters in advice code without having to access the tjp object supplied by the AspectC++ API.

Figure 5.7 shows examples of pointcut expressions with metadata annotations as proposed in this section. It also demonstrates how bindings of variables are supposed to work. A more in-depth view at the implementation of those features is supplied in the next chapter.

```
1 // pointcut expressions that only selects execution joinpoints of function with the
      @some_annotation annotation
2 advice execution(@some_annotation && "% ...::%(...)") : around() {
3   ...
4 }
5
6 // binding a single annotation parameter
7 advice execution(@loopx(i)) : after(int i) {
8   ...
9 }
10
11 // binding multiple annotation parameters
12 advice execution(@func(a,b)) : before(int b, bool a) {
13   ...
14 }
```

Figure 5.7: Examples of metadata annotations in pointcut expressions

5.4.2.2 Call Joinpoints

When applying the changes described above to the pointcut language of AspectC++ an interesting side effect occurs. The `call` pointcut function when used with a metadata annotations as an argument does return a subset of joinpoints that can be divided into two quite distinct parts:

(a) Annotated calls

A part of the returned subset of joinpoints consists of calls annotated with the metadata annotation supplied as an argument to the pointcut function.

(b) Calls to annotated functions

The other part is call joinpoints which target function joinpoints annotated with the metadata annotations supplied as an argument of the pointcut function.

A developer using AspectC++ has to be aware of this effect. In general this should encourage developers to utilize different annotations for different types of joinpoints

even though it is not required by the language. It can be considered good programming practice. Similar to how the use of different namespaces and classes is encouraged by OOP even though everything could be placed within one namespace or one class.

5.4.3 Pointcut Expressions for Explicit Joinpoints

Based on the design goals mentioned above the identification of explicit joinpoints using metadata annotation should only utilize the existing pointcut functions. Adding new pointcut functions specifically designed for explicit joinpoints would certainly be possible, but is not necessary for the joinpoints designed in this thesis.

5.4.3.1 Compound Statement Joinpoints

The block joinpoint type introduced in the previous chapter consists of a new type of name joinpoint and a child execution joinpoint. That means block joinpoints can be addressed using the `execution` pointcut function of AspectC++.

Similarly to the behavior of the `call` pointcut function as mentioned above. The usage of metadata annotations as arguments for the `execution` pointcut function also result in ambiguous results. The pointcut function produces a subset of all execution joinpoints when used with metadata annotations. This subset consists of all named joinpoints that have a execution joinpoint in their list of children, which means all functions and blocks that are annotated with a matching metadata annotation.

This should again serve as a reminder, that developers should not reuse the same annotations in multiple contexts for pointcut expressions that use metadata annotations. To be truly distinctive, a wide variety of identifiers should be used in combination with annotations (in a software project).

5.5 Improved Expressiveness

In the last section of this chapter the limitations discussed in the previous chapter (see section 4.2) are revisited and reevaluated considering the extensions proposed in the previous two chapters.

5.5.1 Identifying Specific Calls

One limitation mentioned in the previous chapter is that different calls to the same function within the same scope can not be separately addressed in AspectC++. Metadata annotations allow to annotate specific calls of interest and to address those independently of other calls to the same function.

Similarly specific calls to overloaded operators can be addressed and only the execution

of those precise calls can be advised without changing the behavior of other executions of the same function.

5.5.2 Fine-Grained Execution Advice

Block joinpoints provide a mechanism to explicitly encapsulate parts of functions and to apply advice to those parts directly without the need for a workaround as mentioned in the discussion of joinpoint model limitations in the previous chapter.

In general block joinpoints supply a programmer using AspectC++ with a high number of additional joinpoints for the implementation of cross-cutting concerns as they are common language elements that are present at many places in existing code.

6 Implementation

This chapter describes the practical work done in the context of this thesis. It describes the implementation of the features designed in the previous two chapters into the AspectC++ compiler. In particular it shows where differences between the theoretical and practical design were necessary and examines the reasons for those differences. This chapter also provides an insight into problems and obstacles that became apparent during the implementation that were not considered in the design process.

6.1 AspectC++ Compiler Basics

As mentioned before the AspectC++ compiler is a source code weaver (see section 3.4). The AspectC++ source code is transformed into valid C++ source code that incorporates the changes invoked by the aspect code. The compiler itself is written in C++ and utilizes the PUMA framework for all parsing and transformation operations. Without describing the internal processes in detail, the relevant subjects for the implementation of the features described in the previous two chapters are:

PUMA framework extension

The whole parsing process is implemented in the PUMA framework, therefore extensions to the AspectC++ language have to be first implemented as part of the PUMA framework.

Joinpoint model extension

This refers to the extension of the abstract joinpoint model mentioned in chapter 4. It also refers to the generation of the project repository, which is the specific instantiation of the joinpoint model generated during compilation.

Pointcut language extension

The evaluation of the pointcuts in an AspectC++ project is handled inside the AspectC++ compiler source code and not by the PUMA parser. To allow for the usage of metadata annotations as part of pointcut expressions and to implement the ability to identify explicit joinpoints with them, this evaluation process has to be extended.

Weaver extension

The weaver is a part of the AspectC++ compiler that is invoked to weave the aspect code and the base code of an AspectC++ project. It produces valid C++ source code that executes the behaviors described in the advices at the appropriate places in the base source code. In the context of this thesis the weaving of block joinpoints has to be implemented on top of the existing transformations.

6.1.1 PUMA Framework

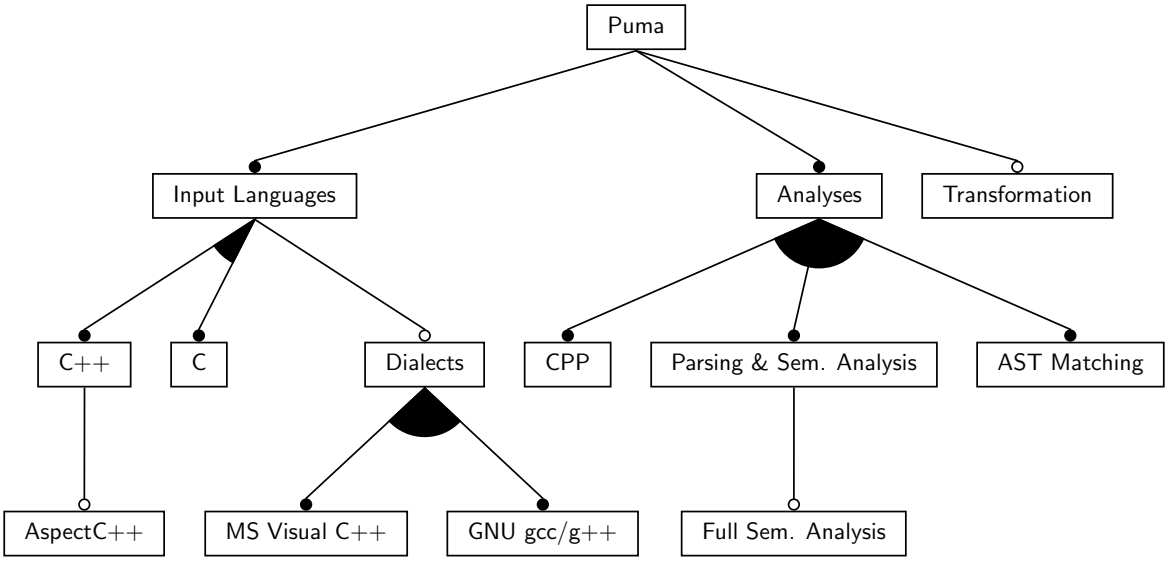

Figure 6.1: PUMA framework feature model [35]

PUMA is a code analysis and manipulation framework developed by Mathias Urban, Olaf Spinczyk and others [35]. PUMA is used to parse C, C++ or AspectC++ source code. It is part of the AspectC++ compiler. PUMA was developed utilizing AOP concepts, it is highly modular and aspects are used to implement a number of features. For example PUMA supports several common language extensions, such as GNU and MS Visual C++. In addition it also supports a number of C++ standards, such as C99 or C++11. Figure 6.1 shows an overview over the PUMA feature model [35].

Aspect-oriented concepts are used in the PUMA framework to separate parallel running features. For example the syntax and semantic are separated into different classes the interfaces are implemented by utilizing loose-coupling through aspects instead of traditional OOP transactions. The advantage of this approach is that features like dialects and different language standards can be exported into aspects. As a matter of fact the PUMA framework is also a software product-line that can be configured to support only specific features.

The AspectC++ compiler uses PUMA for a multitude of tasks. All parsing is done using PUMA. Code analysis tasks and code transformations also utilize classes that are part of the PUMA framework. This close relation between the compiler and the PUMA framework leads to the fact that to implement language extensions into AspectC++ modifications of the PUMA framework are also necessary.

6.2 PUMA Framework Extension

To integrate metadata annotations into the AspectC++ compiler the first step is to extend the PUMA framework. The syntax and semantic defined in the previous chapter have to be integrated into the parser. During the implementation the changes where introduced and debugged without the use of aspects to speed up the developing process. Nevertheless, a number of functions already required the use of aspects and advices in this phase. There are features in the PUMA framework that are quite similar to metadata annotations. For example the GNU C++ dialect supports the use of additional prefixes to existing syntax elements in the C++ language. It is planned that the metadata annotation feature should be an optional feature in PUMA, for this purpose it should be implemented in aspects similar other additional features already present in the PUMA framework.

It is important when discussing the integration of metadata annotations into the PUMA framework to differentiate several separate concerns. One, metadata annotations outside the scope of an aspect can be used to annotate language elements as defined in the previous chapter. Those annotations do not change the semantic behavior of those elements. That means the best solution would be to notice the metadata annotations and save their relevant information before continuing with the syntactic and semantic parsing process without additional changes. The AspectC++ language extension of the PUMA compiler uses a separate semantic database class to store additional information. Is the AspectC++ language support enabled at compile time the class is statically added to the standard semantic database using the slice mechanism provided by AspectC++. The PUMA framework uses static weaving to integrate that additional class into the normal semantic database. Additional advices are applied to enable the behavior concerning information that is to be saved in this database.

During the parsing process of PUMA, metadata annotations are collected and the information about identifiers, parameters and annotation targets are saved in this separate semantic database. The semantic rules concerning annotations are also enforced at this point in the execution through additional advice.

Two, metadata annotations inside of functions bodies are different in relation to the PUMA framework. For the most part they are ignored in the syntactic parsing and semantic analysis. The semantic rules for the metadata annotations themselves still

apply during the parsing process, but their information is not saved inside the semantic database at this point in time.

As mentioned above the implementation of the additional cross-cutting concerns introduced by metadata annotations uses aspects. For example take the advice code displayed in figure 6.2 this is code that is used for the syntactic parsing of metadata annotations applied to a compound statement.

Three, metadata annotations inside of an aspect, more specifically inside of pointcut expressions are quite different. To integrate metadata annotations into the pointcut language with the syntax described in the previous chapter it is necessary to allow the parsing of a metadata annotation as a valid postfix expression. This enables the annotation to be used as function parameters for example. But this has to be valid only within pointcut expressions. This in combination with the syntax outside of aspects would lead to an ambiguous parsing process, should those cases not be clearly separated. For the purpose of solving this problem, the semantic had to be extended to not only keep track of entering and leaving aspects or classes as it did already. But to provide additional interfaces to keep track whether or not the current elements are part of pointcut expressions. Furthermore, unlike the metadata annotations outside of pointcut expressions, the metadata annotations inside of pointcut expressions need to be compatible to binary expressions in combination with match expressions and other metadata annotations as described in the previous chapter as well.

This concludes the changes that had to occur within the PUMA framework. The next step in the implementation process take place in AspectC++ compiler source code itself and not in an additional library. For this step it is important to note that aspects can only be used in the context of the PUMA framework, the AspectC++ compiler is "only" a C++ project not an AspectC++ project.

```
1 advice within (derived(csyntax())) && execution("bool ...::CmpdStmt::parse(...)") && that
      ("Puma::CCSyntax") : around () {
2   JoinPoint::template Arg<0>::ReferredType &s = *tjp->arg<0>();
3   if (s.tag_spec_seq()) {
4     tjp->proceed();
5   } else {
6     tjp->proceed();
7   }
8 }
```

Figure 6.2: Example advice for parser modification concerning metadata annotations

6.3 Joinpoint Model Extension

The joinpoint model has to be extended in three areas. The first area is that joinpoints can have additional information in the form of metadata annotations. The possibility

was implemented for every type of joinpoint although it is not necessarily used the interface functions are present inside the source code. The decision whether and which information is added is implemented in the functions used to generate the joinpoint model.

The second area is the introduction of block joinpoints into the hierarchy of the joinpoint model. Block joinpoints are children of either other block joinpoints or function joinpoints. Every block joinpoint has one child that represents the corresponding execution joinpoint.

The joinpoint model generation is separated into two parts during the weaving process. During the first generation phase of the joinpoint model the function bodies are ignored, the metadata annotations are collected and inserted into the joinpoint model for classes and functions only. During the second generation phase the function bodies are evaluated using a mechanism to traverse the syntax trees constructed by the PUMA framework, which is also supplied as part of the PUMA framework. In this second phase the call and block joinpoints are generated. For support of mechanisms like the `without` pointcut function later on, the call joinpoints are placed as children of a block joinpoint should the call expressions be part of the block statements syntax tree.

There are additional requirements that had to be placed on the extensions of the joinpoint model that emerge from the implementation of the pointcut language extensions and are explained as part of the next section in this chapter.

6.4 Pointcut Language Extension

While the PUMA framework parses the pointcut expressions and performs a number of semantic checks on the expressions, the evaluation of pointcut expressions as part of the AspectC++ language definition is performed later as part of the AspectC++ compiler source code. Based on the syntax tree of a pointcut expression an internal pointcut expression tree is generated by the compiler. During the evaluation of a pointcut expression, every joinpoint in the joinpoint model is evaluated against the pointcut expression.

For the purpose of utilizing the changes introduced in this thesis in the AspectC++ compiler. A number of changes have to be introduced into the evaluation of pointcut expressions. First metadata annotations must become valid elements of this tree and similar to match expressions must match against all joinpoints that are annotated as described by the annotation in the pointcut expression.

Metadata annotations are similar to match expressions as they can be used to identify name joinpoints like classes and functions. It is important to note here that metadata annotations have to be declared as being name joinpoints in the context of the pointcut language. This puts restrictions on the joinpoints they can match against with regard

to the extensions of the pointcut language concerning block and call joinpoints. As only name joinpoints can be expressed with match expressions, an annotated call joinpoint must be represented as a name joinpoint in the joinpoint model. The same applies to a block joinpoint, which is why the block joinpoint and the execution joinpoint of the block are modeled as two separate elements in the joinpoint model.
Furthermore many pointcut functions have to be extended in some regards. For example `within` has to be extended to recognize block joinpoints as another type of container for joinpoints. The `execution` pointcut function also must recognize a block joinpoints as a valid parent for execution joinpoints. The `call` pointcut function ordinarily checks whether an argument supplied to it matches against the target function of a call to determine the matching call joinpoints. In addition to that it must recognize annotated calls as valid matches as well.

A major problem concerning the utilization of metadata annotations in pointcuts became apparent in this phase of the implementation process When pointcut expressions are evaluated the information provided by the evaluator only consists of whether or not a joinpoint is a match concerning the pointcut expression. Due to the nature of metadata annotations, any joinpoint can possible have several annotations applied to itself. For the purpose of weaving the advice code later in the weaving process it can be important to recognize which annotation was matched in context of a joinpoint. The current joinpoint model evaluation procedure does not support this.

This problem is unsolved in the current implementation, to assure only expected behavior in this regard, it is currently forbidden to use multiple annotations in combination with annotation parameters, as the problem only occurs in this context. Figure 6.3 shows an example of this.

```
1 void some_function() {
2   @foo(2, "Note foo") @bar("Note bar", 4) {
3     ...<statements>...
4   }
5 }
```

```
1 aspect ProblemParameters {
2   advice execution(@foo(a,b)) || execution(@bar(b,a)) : after(int a, const char* b) {
3     ...<advice code>...
4   }
5 };
```

Figure 6.3: Example of the ambiguous metadata annotation parameters

6.4.1 Metadata Annotation Parameters

Bindings of variables between metadata annotations and advices were a huge problem during the implementation. The feature is described in the previous chapter. As-

pectC++ supports the binding of variables by utilizing specific built-in pointcut functions (`args`, `that`, `target`) in a pointcut expression. The problem with integrating a similar feature into the metadata annotations is that those parameters can arrive from additional sources. In theory they can arrive from several metadata annotations that have to be evaluated in combination with each other.

This leads to a number of rather severe restriction when combining metadata annotations with those pointcut functions.

During the evaluation of a pointcut expression it is important to check whether or not the metadata annotation parameters the variables declared through other means try to apply to the same advice variable. These additional steps require an extension of the binding class that is used for the tracking of the bindings of variables.

Due to time constraints during the development it was not possible to implement that feature successfully yet, Instead should metadata annotation parameters and other pointcut functions try to bind advice function arguments for the same advice node an error message is currently produced.

6.5 Weaver extension

With all the changes made previously to this point in the implementation process, it is surprising that the weaving of call joinpoints does not need to be modified, the changes introduced in the context of this thesis do only require the changes described above to be implemented correctly.

6.5.1 Block joinpoint weaving

As described in chapter 4 block joinpoints require a static code analysis when `around` advice is to be applied to them.

In the same chapter a comparison was conducted between function and block joinpoints. For the purpose of weaving it was therefore decided to encapsulate the block inside a special type of function for the purpose of advice weaving.

The C++ language supports local classes inside of functions, that are only valid in the scope of the parent function. Furthermore local classes in AspectC++ have the advantage to be ignored in the creation of the joinpoint model. The information collected by the static code analysis is enough to export the block statement body into a static function inside a local block, more precisely the static code analysis provides the necessary information to determine the parameters necessary for this local class function.

The C++ language allows the use of variable references which means that changes

```
1 class TestClass {
2 ...
3 void debug (int i) {
4   ...
5   {
6     class __class_exec_block5 {
7     public:
8       static void __exec_old_block5(int &i, TestClass *__TThis__) {
9         {
10          cout << "x: " << __TThis__->x << endl;
11          i = i + __TThis__->x;
12          cout << "i + x: " << i << endl;
13        }
14      }
15    };
16    typedef TJP___exec_old_block6_0  <void, void, void, AC::TL < int, AC::TLE > >__TJP;
17    __TJP tjp;
18    tjp.block_func (&__class_exec_block5::__exec_old_block5, i, this);
19    AC::invoke_BlockTesting_BlockTesting_a0_around<__TJP> (&tjp);
20  }
21  ...
22 }
23 ...
24 };
```

Figure 6.4: Example of a block statement exported into a local class.

on those references are directly applied to the referenced variables as well. This feature simplifies `around` advice on block joinpoints. For discovering the used variables inside of a block statement a static code analysis of the block statement body code is required and has to be performed. During the implementation of the three advice types for block joinpoints it became also apparent that encapsulating the block statement in a local class was necessary for all three and not only for `around` advice. Therefore, in the implementation the static code analysis is performed for any advice that is applied to a block statement. In the original function the block statement can be replaced with a call to the local class function and the arguments can be supplied as referenced variables to ensure that no semantic changes occur during execution of the program. This allows to weave a block execution joinpoint nearly exactly the same as a normal function execution joinpoint, with one difference that is desecribed later in this section. Nevertheless it still simplifies the weaving of block joinpoints.

The description above simplifies a number of considerations that had to be applied concerning the static code analysis. In addition to the cases listed in chapter 4, the additional encapsulation inside of a new class scope requires additional changes to the block statement body. These changes concern the use of the `this` pointer, as the semantic meaning of that pointer is changed inside the local class. Therefore that pointer has to be replaced with an additional parameter that also supplied by the static code analyis.

An example of that behavior is shown in figure 6.4. Another noteworthy aspect of the implementation is that unlike with ordinary function joinpoints, the “block function”

has to supplied as a pointer to the `TJP` template instantiation in the case of around advice on the block joinpoint. This is because the function needs to be executed inside the `proceed()` function of the template instantiation, and the fully qualified function name is only valid inside of the scope of the parent function, where as the template is defined outside of the scope of the parent function.

6.5.2 Conclusion

At the end of the weaving process all metadata annotations have to be removed from the source code. For metadata annotations that are part of pointcut expressions that is already implicitly done when removing the aspects in the clean up phase of the source code weaver. For all other metadata annotations it is important to have access to their positions in the source code throughout the whole weaving process. This feature was actually one of the first features implemented after finishing the first few modifications in the PUMA framework. Because without this feature no testing would have been possible during the implementation of this project.

Leaving the project with the implementation problems mentioned during this chapter unsolved is not optimal for the evaluation process described in the next chapter. Nevertheless, the extensions to AspectC++ developed during the course of this thesis, even with the limitations placed on them as described above, are evaluable improvements in the context of the AspectC++ language development.

7 Evaluation

With one focus in recent research, with regards to AspectC++, being on embedded system software, especially small ("deeply") embedded operating systems, the CiAO (*CiAO is Aspect-Oriented*) operating system [32] was chosen as one evaluation target for this thesis. The goal is to demonstrate how metadata annotations in AspectC++ can be used to optimize the implementation of cross-cutting concerns even in an aspect-oriented software project.

Afterward the extension of the joinpoint model developed in the context of this thesis is compared to related works that were conducted in the context of the AspectJ language. In the end of this chapter the results collected are discussed.

7.1 Refactoring Aspects in CiAO

The CiAO operating system [32] was developed with the goal of creating an aspect aware operating. CiAO in this regard is special, as unlike earlier works concerning the use of AspectC++ in embedded operating systems [31], it was designed from the beginning as an aspect-oriented software product-line (SPL) that utilizes aspects to implement cross-cutting concerns.

The CiAO operating system design follows three fundamental principles [32]:

The principle of loose coupling.

> Make sure that aspects can hook into all facets of the static and dynamic integration of system components. The binding of components, but also their instantiation (e.g. placement in a certain memory region) and the time and order of their initialization should all be established (or at least be influenceable) by aspects.

The principle of visible transitions.

> Make sure that aspects can hook into all control flows that run through the system. All control-flow transitions into, out of and within the system should be influenceable by aspects. For this they have to be represented on the joinpoint level as statically evaluable, unambiguous joinpoints.

The principle of minimal extensions.

> Make sure that aspects can extend all features provided by the system on a fine granularity. System components and system abstraction should be fine-grained, sparse, and extensible by aspects.

CiAO is interesting in the context of metadata annotations in AspectC++ as it is a software system that is specifically tailored to the requirements, possibilities and even the limitations of the standard AspectC++ language. It is interesting to evaluate whether or not even in such a software system the extensions provided in this thesis can be used to simplify the implementation of cross-cutting concerns.

CiAO offers a fine granularity of joinpoint, this means that at any point in the execution flow of the program advices can be applied. This is realized by encapsulating the source code into numerous namespaces, classes and functions on an excessive level. CiAO is also an example of how far two dimensional separation, as is used in ordinary AspectC++, can be driven.
The table in figure 7.1 shows a couple of numbers associated with the CiAO version used in this evaluation. The principles utilized in the design process of CiAO resulted in a highly configurable and complex system.

File type	Number of files	Non-blank lines
Header files (*.h)	384	16639
C++ source files (*.cpp)	181	4919
Aspect source header (*.ah)	701	13603

Figure 7.1: Statistics of the CiAO operating system

7.1.1 Named Pointcuts in CiAO

In CiAO a huge number of the named pointcuts are used in advice code. In numbers that means that of there are 218 named pointcuts that are currently defined inside the CiAO source code.

In actuality those numbers are slightly off though, as some of those named pointcuts are defined in multiple places, while others are only renaming of others. Where renaming refers to named pointcuts that are identical in the match expressions they consist of.

A programmer could use metadata annotations to annotate some of those functions instead of defining, sometimes several layers of, named pointcuts. The problem with such a high number of named interconnected pointcuts is that it requires nearly complete system knowledge to understand even small changes in aspects and advices.

Furthermore there are several cases in the current version of the CiAO source code

that show another significant problem with that many named pointcuts. For example the “asReturningVoid()” pointcut is never actually used anywhere in the source code. Instead there are several pointcut expressions that use a “returningVoid()” named pointcut that does not actually exist. The worst part of this is, that those pointcuts match exactly to zero joinpoints in their current form. There are clearly cases where named pointcuts based on match expressions are a better choice instead of metadata annotations. For example “asReturningStatusType()” matches against all functions that return a value of type “StatusType“, the number of joinpoint matches in the current source code are 40 function joinpoints. Metadata annotations provide no improvement with regard to this named pointcut.

7.1.2 Refactoring Aspects with Metadata Annotations

Metadata annotations allow the definition of additional dimensions for identifying locations of interest in a programs source code. In CiAO the construction of some aspect related requirements resulted in the existence of empty functions inside the CiAO source code which have only two reasons for their existence. One, they provide call and execution joinpoints for receiving advice from several sources.

And two, they provide a way for advice code to invoke the behavior encapsulated in the advises for these joinpoints. Which would be difficult if these functions were not present to be invoked from inside of advices.

Introducing explicit joinpoints into CiAO to refactor empty functions is not possible because of these interactions without considerable more effort in regard to time, than this evaluation allows.

7.1.3 Results

The refactoring that was planned in the context of this evaluation process turned out to be far more complex than first examinations implied. The CiAO system as it is currently designed encapsulates behavior into functions on such a fine-grained scale that refactoring joinpoint identification with the use of metadata annotations is possible only when significant changes in the global structure of the source code occur.

The interactions between the 701 aspect headers, which are part of the CiAO source code are very complex and any form of change has numerous side effects in regard to those interactions.

7.2 Comparing Related Works

This section begins the second part of this chapter, it evaluates how the extensions developed as part of this thesis to the AspectC++ joinpoint model compare against

extensions developed in the context of the AspectJ joinpoint model.

7.2.1 Closure Joinpoints

The closure joinpoints paper by Eric Bodden [23] was one of the sources of inspiration that lead to the design of the AspectC++ block joinpoint in this thesis. Based on this relation a comparison between the two is logical.

Unlike block statements in C++, closures in Java are quite different from a pure language element viewpoint. Where block statements are generic language elements that are used at numerous places in nearly all C++ programs and therefore are implicitly provided by C++ application programmers in practice. Closures are a very specialized language element in the Java programming language.

In the context of applying advice to those two constructs, the differences disappear. Because to apply advice to a block in AspectC++ we need to annotate the block with a metadata annotation, where in AspectJ the programmer has to redefine the block statement as a closure. Overall the amount of effort required for the advice of both is about equal.

Furthermore for accessing variables used in those constructs in the advice code those variables have to be explicitly announced, either as annotation parameter in AspectC++ or as closure arguments in AspectJ.

Overall the effort required for the advice of block statements in AspectJ and AspectC++ is equal. The differences between the languages presented themselves through the fact that implementing block joinpoints in AspectC++ was significantly less problematic.

7.2.2 Statement Annotations

Another interesting paper in the context of metadata annotations was written by Marc Eaddy concerning statement annotations inside of functions [24]. This paper proposed a language extension to annotations in Java to allow the annotation of single statements.

In comparison we can express all examples provided in that paper with our block joinpoint mechanism in AspectC++. In addition as mentioned above the block joinpoints introduced in AspectC++ also have all the features of closure joinpoints.

Figures 7.2 show an example from the paper and the equivalent implementations utilizing AspectC++ and block joinpoints. The annotation in the context of block joinpoints and AspectC++ have to be applied to empty compound statements, but in combination with optimizations supplied by the C++ compiler this does not affect the runtime of the final program. A difference during the execution comes from the fact that unlike closures in AspectJ, block joinpoints in AspectC++ does not have a unique signature

which leads to difference in the output generated here. Even though the code itself models the identical behavior.

(a) Example for use of statement annotations in AspectJ [24]

```
...several statements...
@Note("Searching for plug-ins...")
...several statements...

@Note("Entering very long, but hopefully not infinite loop")
while(true) { ... }
@Note("Loop exited successfully")
```

```
aspect LogNotesAspect {
  before(Note noteAnnotation) : @annotation(noteAnnotation) {
    System.out.println(noteAnnotation.value() + "[" + thisJoinPoint + "]");
  }
}
```

(b) Previous example implemented in AspectC++

```
...several statements...
@Note("Searching for plug-ins...") { }
...several statements...

@Note("Entering very long, but hopefully not infinite loop") { }
while(true) { ... }
@Note("Loop exited successfully") { }
```

```
aspect LogNotesAspect {
  advice execution(@Note(const char*)) : before(const char *s) {
    cout << s << "[" << tjp->signature() << "]" << endl;
  }
};
```

Figure 7.2: Statement annotations (AspectJ) and block joinpoints (AspectC++)

7.3 Discussion

CiAO as an evaluation was a bad choice with regards to the language features developed in context of this thesis. The design of CiAO was deeply driven by the limitations of the ordinary AspectC++ language. Those limitations lead to the implementation of fine-grained advice on a function level. Separation of concerns is achieved by utilizing numerous namespaces and functions. While only able to utilize two dimension in this regard, CiAO uses a huge spectrum of identifiers to separate the different features and control flows during execution.

8 Conclusion and Future Work

Metadata annotations allow the definition of additional dimensions for identifying and separating joinpoints independent from the primary decomposition that can be applied to a program in C++. This improves the abilities of programmers to implement cross-cutting concerns by decoupling the application of advice from the tyranny of the primary decomposition problem [34].

In addition this thesis extended the AspectC++ joinpoint model through mechanisms that allow for the execution of advice inside of functions without additional workarounds. These explicit joinpoints were only possible because of the extensions provided by the metadata annotation mechanism. An example of such an explicit joinpoint is the block joinpoint type introduced in this thesis.

The current implementation of those features is still not fully matured. As there still remain a number of problems that need to be addressed in the context of pointcut language integration and the application of metadata annotation parameters in combination with binding of variables to advices, as was shown in the implementation chapter of this thesis.

It was shown that the granularity of the joinpoint model in AspectC++ does not need to end at the function level. A valid concern in this regard is that extensions and features as used in AspectJ are not necessarily equally valid in the context of AspectC++ because of the inherent differences between those two languages. The lack of problems regarding the implementation of block joinpoints in AspectC++, without the added complication provided by metadata annotations, also can be seen as an interesting discovery in the context of this thesis.

An evaluation of AspectJ extension proposals that were dismissed due to their lack of interoperability with the AspectJ language in its current form, with regard to their application in the context of AspectC++ could prove interesting.

8.1 Future Work

This thesis should only be considered the beginning of research into the combination of metadata annotations and the AspectC++ language. Significant problems have to be solved regarding the application of metadata annotations in combination with parameters and binding of variables in the context of advices.

Type annotations are another interesting concept and the ideas mentioned in chapter 4 should only be considered starting points for further research in this context.

In addition, because of to the shortcomings in the evaluation in the scope of this thesis, designing an aspect-oriented software project utilizing metadata annotations and block joinpoints could be another topic for research.

Furthermore the C++11 standard was recently finalized, even though to date still no compiler completely supports the extensions provided in this new standard. The consequences and possibilities regarding both the ordinary AspectC++ language and the metadata annotations feature specifically can be considered open questions for future work as well.

Bibliography

[1] Harbulot, Bruno ; Gurd, John R.: A join point for loops in AspectJ. In: *Proceedings of the 5th international conference on Aspect-oriented software development*. New York, NY, USA : ACM, 2006 (AOSD '06), p. 63–74

[2] Baca, Murtha ; Gilliland-Swetland, Anne: *Introduction to Metadata: Pathways to Digital Information*. Getty Publications, 1998. – ISBN 0892365331

[3] Duval, Erik ; Hodgins, Wayne ; Sutton, Stuart ; Weibel, Stuart L.: Metadata Principles and Practicalities. In: *D-Lib Magazine* 8 (2002), April, Nr. 4. http://www.dlib.org/dlib/april02/weibel/04weibel.html

[4] Goldfarb, Charles F. ; Prescod, Paul: *The XML Handbook*. Upper Saddle River, NJ 07458, USA : Prentice-Hall PTR, 1998

[5] Buckley, Alex ; Kiczales, Gregor ; Lea, Doug: *JSR 175: A Metadata Facility for the Java™Programming Language*. http://jcp.org/en/jsr/detail?id=175. Version: August 2011

[6] Darcy, Joe ; Chapman, Bruce ; Lea, Doug ; Niemeyer, Patrick D.: *JSR 269: Pluggable Annotation Processing API*. http://jcp.org/en/jsr/detail?id=269. Version: August 2011

[7] McLaughlin, Brett: *Annotations in Tiger, Part 1: Add metadata to Java code*. http://www.ibm.com/developerworks/java/library/j-annotate1/. Version: August 2011

[8] Buckley, Alex ; Ernst, Michael ; Lea, Doug: *JSR 308: Annotations on Java Types*. http://jcp.org/en/jsr/detail?id=308. Version: August 2011

[9] Cazzola, Walter ; Cisternino, Antonio ; Colombo, Diego: [a]C#: C# with a customizable code annotation mechanism. In: *Proceedings of the 2005 ACM symposium on Applied computing*. New York, NY, USA : ACM, 2005 (SAC '05), p. 1264–1268

[10] Watt, D. A.: *Programming Language Concepts and Paradigms*. Prentice Hall, 1990

[11] Gamma, E. ; Helm, R. ; Johnson, R. ; Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995

[12] Ossher, Harold ; Tarr, Peri: Using multidimensional separation of concerns to (re)shape evolving software. In: *Commun. ACM* 44 (2001), October, p. 43–50. – ISSN 0001–0782

[13] Roo, A.J. de: *Towards more robust advice : message flow analysis for composition Filters and its Application*, University of Twente, Master's thesis, 2007

[14] Filman, Robert E. ; Friedman, Daniel P.: Aspect-Oriented Programming is Quantification and Obliviousness, Addison-Wesley, 2000, p. 21–35

[15] Gradecki, J. D. ; Lesiecki, N.: *Mastering AspectJ: Aspect-Oriented Programming in Java*. John Wiley and Sons, 2003. – ISBN 0471321044

[16] Lohmann, Daniel ; Spinczyk, Olaf ; Schröder-Preikschat, Wolfgang: Lean and Efficient System Software Product Lines: Where Aspects Beat Objects. In: Rashid, Awais (Hrsg.) ; Aksit, Mehmet (Hrsg.): *Transactions on Aspect-Oriented Software Development II* Bd. 4242. Springer Berlin / Heidelberg, 2006, p. 227–255. – "10.1007/11922827_8"

[17] Lohmann, Daniel ; Spinczyk, Olaf: Developing embedded software product lines with AspectC++. In: *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. New York, NY, USA : ACM, 2006 (OOPSLA '06). – ISBN 1–59593–491–X, p. 740–742

[18] Kiczales, Gregor ; Hilsdale, Erik ; Hugunin, Jim ; Kersten, Mik ; Palm, Jeffrey ; Griswold, William: An Overview of AspectJ. In: Knudsen, Jørgen (Hrsg.): *ECOOP 2001 - Object-Oriented Programming* Bd. 2072. Springer Berlin / Heidelberg, 2001, p. 327–354

[19] Kiczales, Gregor ; Lamping, John ; Mendhekar, Anurag ; Maeda, Chris ; Lopes, Cristina ; Loingtier, Jean-Marc ; Irwin, John: Aspect-oriented programming. In: Aksit, Mehmet (Hrsg.) ; Matsuoka, Satoshi (Hrsg.): *"ECOOP'97 - Object-Oriented Programming"* Bd. 1241. Springer Berlin / Heidelberg, 1997, p. 220–242

[20] Endoh, Yusuke ; Masuhara, Hedehiko ; Yonezawa, Akinori: Continuation Join Points. In: *Proceedings of the Foundations of Aspect-oriented languages workshop at AOSD '06*, 2006 (FOAL '06), p. 7–16

[21] Team, AspectJ: *The AspectJ 5 Development Kit Developer's Notebook*. http://www.eclipse.org/aspectj/doc/released/adk15notebook/index.html. Version: August 2011

[22] Falk, Heiko ; Marwedel, Peter ; Catthoor, Francky: Control Flow Driven Splitting of Loop Nests at the Source Code Level. In: Jerraya, Ahmed A. (Hrsg.) ; Yoo, Sungjoo (Hrsg.) ; Verkest, Diederik (Hrsg.) ; Wehn, Norbert (Hrsg.): *Embedded Software for SoC*. Springer US, 2004. – ISBN 978–0–306–48709–5, p. 215–229

[23] Bodden, Eric: Closure joinpoints: block joinpoints without surprises. In: *Proceedings of the tenth international conference on Aspect-oriented software development*. New York, NY, USA : ACM, 2011 (AOSD '11), p. 117–128

[24] Eaddy, Marc ; Aho, Alfred: Statement annotations for fine-grained advising. In: *Proccedings of the Workshop on Reflection, AOP and meta-data for Software Evolution*, 2006 (RAM-SE '06)

[25] Kim, Howard: *AspectC#: An AOSD Implementation for C#*, Dissertation, 2002

[26] Cazzola, Walter ; Colombo, Diego ; Harrison, Duncan: Aspect-oriented procedural content engineering for game design. In: *Proceedings of the 2009 ACM symposium on Applied Computing*. New York, NY, USA : ACM, 2009 (SAC '09)

[27] Spinczyk, Olaf ; Gal, Andreas ; Schröder-Preikschat, Wolfgang: AspectC++: an aspect-oriented extension to the C++ programming language. In: *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded applications*. Darlinghurst, Australia, Australia : Australian Computer Society, Inc., 2002 (CRPIT '02), p. 53–60

[28] Urban, Matthias ; Spinczyk, Olaf: *AspectC++ Language Reference*. 1.7, April 2011. http://www.aspectc.org/fileadmin/documentation/ac-languageref.pdf

[29] Rho, Tobias ; Kniesel, Günther ; Appeltauer, Malte: Fine-grained generic aspects. In: *Proceedings of the Foundations of Aspect-oriented languages workshop at AOSD '06*, 2006 (FOAL '06), p. 29–35

[30] Koppen, Christian ; Stoerzer, Maximilian: PCDiff: Attacking the fragile pointcut problem. (2004), August

[31] Lohmann, Daniel ; Scheler, Fabian ; Tartler, Reinhard ; Spinczyk, Olaf ; Schröder-Preikschat, Wolfgang: A quantitative analysis of aspects in the eCos kernel. In: *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*. New York, NY, USA : ACM, 2006 (EuroSys '06), p. 191–204

[32] Lohmann, Daniel ; Hofer, Wanja ; Schröder-Preikschat, Wolfgang ; Streicher, Jochen ; Spinczyk, Olaf: CiAO: an aspect-oriented operating-system family for resource-constrained embedded systems. In: *Proceedings of the 2009 conference on USENIX Annual technical conference*. Berkeley, CA, USA : USENIX Association, 2009 (USENIX'09), p. 16–16

[33] *ISO/IEC 14882:2003: Programming languages: C++*. 2003 http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=38110

[34] Tarr, Peri ; Ossher, Harold ; Harrison, William ; Sutton, Jr. Stanley M.: N degrees of separation: multi-dimensional separation of concerns. In: *Proceedings of the 21st international conference on Software engineering*. New York, NY, USA : ACM, 1999 (ICSE '99). – ISBN 1–58113–074–0, 107–119

[35] Urban, Matthias ; Lohmann, Daniel ; Spinczyk, Olaf: The aspect-oriented design of the PUMA C/C++ parser framework. In: *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*. New York, NY, USA : ACM, 2010 (AOSD '10), p. 217–221

List of Figures