AspectC++: Language Proposal and Prototype Implementation [*]

[Position Paper, Category A]

Andreas Gal, Wolfgang Schröder-Preikschat, and Olaf Spinczyk
University of Magdeburg
Universitätsplatz 2
39106 Magdeburg, Germany
{gal,wosch,olaf}@ivs.cs.uni-magdeburg.de

ABSTRACT

The success of aspect-oriented programming (AOP) raises and falls with user-friendly tool support. With AspectJ[1] the first complete and powerful language extension for AOP has been created. With this paper we intend to extend the AspectJ approach to C/C++. We will present and discuss a proposal for a set of language extensions we call AspectC++ to facilitate aspect-oriented programming with C/C++ and we will illustrate our prototype implementation of a compiler for this new language.

1. MOTIVATION

While there is currently a heavy focus on aspect-oriented programming (AOP) with Java[2], other languages like C and C++ experience far less attention in the effort of creating and establishing extensions for aspect-oriented programming. This is especially dissatisfying as a large fraction of applications is still being developed in C and C++. These applications cannot simply be considered as legacy, as for a number of domains specific features of the C/C++ language like minimal requirements in terms of run-time support are crucial. To be able to exploit the advantages of aspect-oriented programming (AOP) also with C/C++, it is inevitable to provide an AOP framework.

A key component of such a framework is a set of language extensions to support the independent implementation of aspects. AspectJ [6] is an example for such a set of language extensions for the Java language. The popularity of AspectJ was supported by the fact that a working AspectJ compiler is publicly available and many programmers already gathered experience with aspect-oriented programming using AspectJ. As C++ and Java have grammatically and semantically a lot in common, it seems natural to use AspectJ as foundation when creating a set of extensions for the C/C++ language to support aspect-oriented programming.

In this paper we will present such an extension to the C++ language, called AspectC++. While AspectC++ is conceptually very similar to AspectJ, it was also necessary to adopt and streamline what we learned from AspectJ accordingly to the specific peculiarities of the C/C++ syntax and semantics.

The outline of this paper is as follows: In section 2 we will describe in detail the AspectC++ language and the differences to AspectJ. Following in section 3 the architecture of our prototype implementation of an AspectC++ compiler is explained. Related work is discussed in section 4, followed by the conclusions and a road map of our future work in section 5.

2. THE ASPECTC++ LANGUAGE

AspectC++ is a general-purpose aspect-oriented extension to the C++ language. We will use a simple example (figure 1) we have taken from the AspectJ introduction to illustrate the features of AspectC++.

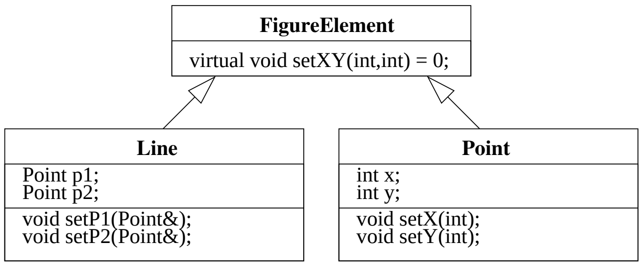

Figure 1: Class diagram of our example scenario

*This work has been partly supported by the German Research Council (DFG), grant no. SCHR 603/1-1 and SCHR 603/2.
[1] AspectJ is a trademark of Xerox Corporation.
[2] Java is a registered trademark of Sun Microsystems, Inc. in the U.S. and other countries.

2.1 Language Introduction

In AspectC++ join points are defined as points in the component code where aspects can interfere. Join points can refer to code, types (*classes*, *structs*, and *unions*), objects and control flows.

Pointcut expressions can be used to identify a collection of such join points. They are composed from *pointcut designators* and a set of algebraic operators. In figure 2 the pointcut designators of AspectC++ are listed.

Additional to the pointcut designators *named pointcuts* can be declared to ease the composition of complex pointcut expressions:

```
pointcut moves() =
  calls("void FigureElement::setXY(int,int)") ||
  calls("void Point::set%(int)") ||
  calls("void Line::setP%(Point &)");
```

In this example *match expressions* are used filter-out specific join points. Match expressions are similar to C++ signatures, but can contain additionally the wildcard character "%". For example, "void %::set%(%)" could be used to match the same set of join points (in this example scenario).

Advice

An advice declaration can be used to specify code that should run at the join points specified by a pointcut expression:

```
advice moves() : void after() {
  cout << "A figure element was moved." << endl;
}
```

Context information from the join point can be exposed by using pointcuts with arguments and match expressions that contain identifiers instead of type names everywhere where context should be exposed:

```
advice calls("void FigureElement::setXY(x,y)") :
  void after(int x, int y) {
    cout << "A figure element was moved to ("
         << x << ", " << y << ")." << endl;
  }
```

Different kinds of advice can be declared, including an *after* advice that runs after the join point, a *before* advice that is executed before the join point, and an *around* advice, which is executed in place of the join point. An around advice has to request the execution of the original join point code using *proceed()*:

```
advice moves() : void around() {
  cout << "A figure element will move." << endl;
  proceed();
  cout << "A figure element was moved." << endl;
}
```

If the advice is not recognized as being of a predefined kind, it is regarded as a new method to be added to all join points contained in the pointcut expression. The pointcut expression must contain in this case only join points of the type class.

```
pointcut all_classes() = classes("%");
advice all_classes() : void print() {
  cout << "Address: " << (void*)this << endl;
}
```

Aspect

While named pointcut declarations can appear everywhere where declarations are allowed, an advice can only be defined inside an *aspect declaration*. Aspects in AspectC++ implement in a modular way crosscutting concerns and are an extension to the class concept in C++. Additionally to attributes and methods, aspects may also contain advice declarations. Aspects can be derived from classes and aspects, but it is not possible to derive a class from an aspect. By default, for every aspect a single object is instantiated at run-time. Using the *of*-clause it is also possible to bind the instantiation of aspects to certain pointcuts. Pointcuts used with an *of*-clause must contain join points of a uniform type (for example classes, objects, or cflows):

```
aspect classCounter of all_classes() {
  int count;
public:
  classCounter() {
    count = 0;
  }
  advice all_classes() : void bump();
};
void classCounter::bump() {
  count++;
  cout << "Called " << count << " times." << endl;
}
```

This example adds a method *bump()* to all classes. The *of*-clause causes the aspect *classCounter* and thus the private field *count* to be instantiated separately for every class. Thus, each invocation of *bump()* will only count the number of invocations of the *bump()* method *of this class*. Omitting the *of*-clause would cause the default case to be used where only a single instance of the aspect class is created. With this modification only a single *count* variable would be maintained and used by all instances of *bump()*.

2.2 Rationale of AspectC++

So far we have presented the basic concepts of AspectC++, which are mostly adopted from AspectJ. The reason for this similarity is to allow people who learned "thinking aspect-oriented" by using the popular AspectJ implementation to easily switch over to AspectC++, if they are also already familiar with C++. Nevertheless AspectC++ is quite different in some points. The following paragraphs will discuss the reasons for the different design decisions.

Grammar

The parser of AspectJ is integrated into the Java parser and the grammar of AspectJ shares rules with the standard Java part. This allows aspect code and component code to be mixed. For instance, it is allowed to define named pointcuts anywhere in the program, even as attributes of normal classes. Thus AspectJ is really more a Java language extension than a separate aspect language. This makes it not feasible to use AspectJ without modifications together with other component languages than Java. While we would have preferred to have a separate and portable aspect language with its own grammar, where component code fragments are only used as quoted strings, we are aware that the "look and feel" would then differ significantly from AspectJ. To avoid

calls(*"match expression"*) receptions(*"match expression"*) executions(*"match expression"*)	*calls* finds all calls to methods with a signature matching the given *match expression*, *receptions* refers to the code called through an abstract interface with the given signature, and *executions* matches the body of a function. Constructors and destructors are treated like regular methods.
gets(*"match expression"*) gets(*"match expression"*) [*old*] sets(*"match expression"*) sets(*"match expression"*) [*old*] sets(*"match expression"*) [*old*] [*new*]	*gets* and *sets* locate any direct access to fields with a signature matching the *match expression*. Field access through C++ references or pointers are not recognized. *old* and *new* can be used in the aspect code to refer to the old and the new value of the field.
classes(*"match expression"*) derived(*pointcut*) bases(*pointcut*) objects(*"match expression"*) instanceof(*pointcut*)	*classes* filters out all classes, structures, and unions with matching names. With *derived* it is possible to refer to all types in the pointcut expression and all classes derived from them. *bases* can be used to find all base types of classes in a pointcut. *objects* finds objects with matching names, while *instanceof* can be used to locate objects of a certain type.
handles(*pointcut*) within(*pointcut*)	With *handles* for every class in the *pointcut* exception handlers are returned that handle exceptions of that type. *within* matches all join points declared in methods of types in the *pointcut*.
cflow(*pointcut*) each(cflow(*pointcut*))	*cflow* captures join points occurring in the dynamic execution context of join points in the *pointcut*. *each(cflow(...))* can be used only in conjunction with the *of*-clause to refer to execution contexts and not the join points occurring in them.
callsto(*pointcut*)	*callsto* can be used to locate all calls to join points in the given *pointcut*.

Figure 2: Pointcut designators in AspectC++

this problem the grammar of AspectC++ has been integrated into the C++ grammar. Figure 3 shows the grammar extensions. It should be understood as an extension to the C++ grammar listed by Stroustrup [9].

A main design goal was to keep the grammar extension simple. Compiler writers, who want to implement the AspectC++ language, should not be forced to use a hand-coded parser. Instead they should be able to continue using compiler generators and the lexical analysis should not require context information to work. Parsing C++ is already much harder than parsing Java due to ambiguities in the grammar, which require either the syntactical and semantical analysis phases to be mixed or an extremely weak grammar. The AspectC++ extension should not shake such a fragile building more than necessary.

Property-Based Pointcut Designators

One of the main problems with the AspectJ grammar comes from the *property-based pointcut designators*. For example in AspectJ the pointcut

```
calls(void FigureElement.set*(int, int))
```

refers to all calls to methods of class `FigureElement` with two integer argument, void result type, and a name beginning with "set". The symbol "*" is used as a wildcard in identifiers. While this syntax might be familiar for Java programmers it can not be used unmodified in the C++ context. For instance, "*" is a valid part in C++ type declarators, which are used to declare pointers. Allowing "*" to be a wildcard character in pointcut designator therefore leads to ambiguities. For this reason we replaced "*" with "%" in AspectC++. Furthermore the "." between the class name `FigureElement` and the match expression "`set*`" is misleading for C++ programmers, because in C++ "::" is used to specify qualified names. To avoid this problem we have replaced "." with "::" in pointcut designators. For reasons we will discuss below, it is further required to specify pointcut designators as quoted strings. The following example shows the AspectC++ version of the AspectJ pointcut shown above as it results from the described modifications:

```
calls("void FigureElement::set%(int, int)")
```

The quoting of pointcut designators frees the lexical analyzer from the burden to accept wildcards in identifiers in specific contexts and simplifies the grammar significantly: pointcuts can now be parsed with the normal C++ expression syntax.

It is necessary to detect identifiers, which are used to expose context information, in the quoted pointcut designator strings. Furthermore simple regular expression matching is not sufficient to safely identify groups of function signatures. Therefore the pointcut designators must be syntactically analysed. With our solution this parsing step can be postponed and is the job of a separated parser. Thus no additional rules in the C++ grammar are needed.

Generalized Type Names

"Generalized Type Names" (GTNs) are used in AspectJ to select a group of types (i.e. classes and interfaces) with a single pattern. For example

```
subtypes(Object) && !java.io.*
```

refers to all classes that are not part of the `java.io` package. As it is shown here, GTNs may contain algebraic operators

aspect-name:
  identifier

declaration:
  pointcut-declaration

class-head:
  aspect identifier$_{opt}$ aspect-clauses$_{opt}$ base-clause$_{opt}$

aspect-clauses:
  aspect-clauses aspect-clause$_{opt}$

aspect-clause:
  of pointcut
  dominates aspect-name

pointcut:
  constant-expression

member-declaration:
  pointcut-declaration
  advice-declaration

pointcut-declaration:
  pointcut declaration

advice-declaration:
  advice pointcut : declaration

Figure 3: The AspectC++ extensions to the C++ grammar

like "||", "&&", and "!" and the wildcard character "*" for pattern matching. Together with further functions like "subtypes()" this can be used to form complex expressions that select arbitrary sets of classes. GTNs can be used in AspectJ to introduce new attributes, methods, base classes or implemented interfaces into the referred set of classes. For example

```
aspect A {
  private Registry otherPackage.*.r;
}
```

introduces a new private attribute "r" of type `Registry` into each class that matches the GTN "otherPackage.*".

While AspectJ explicitly distinguishes between *pointcuts and advice* on the one hand and *GTNs and introductions* on the other, AspectC++ handles both in a uniform way. For example in AspectC++

```
aspect A {
private:
  advice classes("otherNamespace::%") : Registry r;
};
```

can be used to introduce a member "`Registry r`" into each class in namespace "`otherNamespace`". This design decisions has the following advantages:

- With the concept of named pointcuts in combination with the uniform handling, it is possible to have named GTNs. While this is already a useful feature it is even possible to have virtual or pure virtual GTNs in AspectC++.
- The grammar does not require special rules for GTNs and introductions.
- The users benefit from the coherent language design. Everything that is declared inside of an aspect body and begins with the keyword `pointcut` or `advice` will have an impact on other classes while everything else will become part of the instantiated "aspect object".

Regarding the implementation, this extended join point model requires using typed join points. For example, the AspectC++ compiler has to ensure that only code join points are used in conjunction with an *before*, *after*, or *around* advice.

3. PROTOTYPE IMPLEMENTATION

Our prototype implementation of a compiler for the AspectC++ language is a C++ preprocessor-like compiler based on PUMA [8]. PUMA is a source code transformation system for C++.

The architecture of our AspectC++ compiler is shown in figure 4. First the AspectC++ source code is scanned, parsed and a semantical analysis is performed. Then the planing stage is entered. In the planing stage the pointcut expressions are evaluated and the join point sets are calculated. A plan for the weaver is created containing join points and the operations to be performed at the join points (i.e. adding advice code). While the planing stage is mainly independent from the component language C++, the weaver is now responsible to transform the plan into concrete manipulation commands based on the C++ syntax tree generated by the PUMA parser. The actual code manipulation is then performed by the PUMA manipulation engine. The output of the prototype compiler is C++ component source code with the aspect code woven in. The produced output source code does not contain AspectC++ language constructs anymore and thus can be translated to executable code using conventional C++ compilers.

As the proposed AspectC++ language was designed with the difficulties of the C++ grammar in mind, adding the language extensions to the C++ grammar is quite straight forward. We would have preferred an implementation where the additional grammar rules of AspectC++ can be dynamically added to the C++ grammar, but PUMA did not give us this choice. The PUMA scanner has also been modified to recognize the new AspectC++ keywords. We discovered that it is possible to restrain the number of new keywords to *aspect*, *pointcut* and *advice* by slightly changing the grammar rule for an aspect clause (figure 5).

The modified grammar will accept a superset of valid AspectC++ aspect clauses, but it is possible to filter out invalid aspect clauses during the semantic analysis. The reason for this quirk is to prevent incompatibilities with plain C++ where "'of" and "dominates" are valid identifiers. Such collisions can still occur with the three new keywords, but eliminating those would require a major restructuring of the AspectC++ grammar.

The AspectC++ specific parts of the compiler are implemented as a plugin for the PUMA system. As described above, we have split this plugin into two major parts: the planner and the weaver.

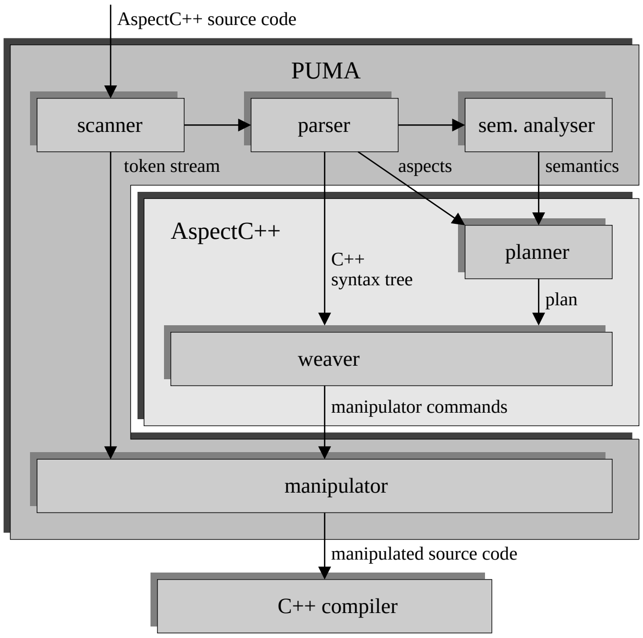

Figure 4: Architecture of the AspectC++ compiler

aspect-clause:
    identifier pointcut
    identifier aspect-name

Figure 5: Modified aspect clause grammar

The planner is using internally sets of join points to represent pointcuts. Thus, the algebraic operators are implemented as operations on join point sets. As in AspectC++ join points can represent locations in the control flow as well as other kinds of join points (i.e. classes), special attention has to be paid to the propagation of the join point type when join point sets are merged. The planner performs an extensive dynamic type checking on the join point sets to ensure the syntactical and semantical correctness of the manipulated source code. The C++ language dependent weaver is responsible to materialize the plan generated by the planner. By doing so the weaver heavily depends on code transformation operations offered by the PUMA library.

For the end-user the AspectC++ prototype compiler behaves like a compiler front-end (*ac++*). In contrast to traditional compilers our front-end transforms a whole program at once. While aspects can be implemented in the same files as the component code, from our experience we suggest to putting aspect declarations in dedicated aspect files.

4. RELATED WORK

Using aspect-oriented implementation techniques in conjunction with C++ does not seem to be very popular in the AOP community. Only very few contributions related to C++ can be found in the proceedings of relevant conferences and workshops of the last years. We explain this with the overall "Java hype" and, more than that, with the lack of tool support. For instance, L. Dominick describes "lifecycle control aspects when applying the Command Processor pattern" and complains "because no weaver technology was available, C preprocessor macros were used" [3].

A very interesting approach is followed by FOG [11]. FOG is a meta-compiler for C++ supporting a superset of the language. Similar to the AspectC++ implementation it is a source to source translator, but the language concept differs. In FOG the C++ "One Definition Rule" is replaced by a "Composite Definition Rule". This allows, for instance, to define multiple classes with the same name, which FOG will then merge into a single class. Functions and attributes can be easily added this way to classes. Function code can be extended with a similar mechanism. While FOG seems to be ideally suited for subject-oriented programming [4][7] the join point model is much less powerful in comparison to AspectJ/C++. Especially the algebraic operations on "pointcuts" and the notion of control flow are useful in many aspect implementations.

More powerful than the FOG approach is OpenC++ [1]. It allows a compiled C++ meta-program to manipulate the base-level C++ code. Classes, types, and the syntax tree of the base-level are visible on the meta-level and arbitrary transformations are supported. This implementation has a lot in common with the PUMA code transformation system. OpenC++ allows aspect-oriented programming, but language extensions that especially support AOP are not provided.

AspectC [2] is an aspect-oriented extension to plain C, which is currently under development to study crosscutting concerns in operating system code. It is not planned to support C++ component code with this implementation [5]. AspectC also adopts the key concepts from AspectJ, but the non-object-oriented nature of C forces AspectC to leave out many useful features like using inheritance to compose aspects. As C is basically a subset of C++ our AspectC++ tools can be used with C as well.

5. CONCLUSIONS AND FUTURE WORK

Much of the work presented in this paper is based on AspectJ. This holds especially for the language introduction in section 2.1. However, we have attempted to streamline the language extensions for the use in conjunction with C++.

A major change compared to the AspectJ language is the modified join point model, which allows to have class, object and control flow join points. The result is a more coherent language design and we would suggest to consider changing the AspectJ join point model accordingly. While it was necessary to make some visible changes to the syntax and grammar of AspectJ, we have preserved most language concepts. This should enable users experienced with AspectJ and C++ to get familiar with AspectC++ without much effort.

The prototype implementation of an AspectC++ compiler is still in an early stage. The compiler cannot yet deal correctly with all AspectC++ language constructs in all possible contexts. However, the clear architecture of the compiler and the powerful tool support in form of PUMA ease the further development of the AspectC++ compiler. As the prototype implementation of the compiler we see also the AspectC++ language as still under development and we are open for suggestions how to further optimize the syntax and semantics.

Concerning future extensions we see room for improvement in the area of the match expressions. We have implemented a PUMA mechanism for creating syntax tree fragments from a textual representation [10]. Currently we are analyzing the possibility to use this syntax-oriented textual representation to create more powerful match expressions.

We also intend to add mechanisms to AspectC++ to manipulate the C++ class hierarchy (for example adding a new base class to certain classes). AspectJ has also limited support for this feature, but we are still investigating a more general approach. We believe that our modified join point approach was already the right step in this direction.

6. REFERENCES

[1] S. Chiba. Metaobject Protocol for C++. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 285–299, Oct. 1995.

[2] Y. Coady, G. Kiczales, M. Feeley, N. Hutchinson, J. S. Ong, and S. Gudmundson. Exploring and Aspect-Oriented Approach to OS Code. In *Proceeding of the 4th ECOOP Workshop on Object-Orientation and Operating Systems (ECOOP-OOSSWS'2001)*, pages 55 – 59. Universidad de Oviedo, June 2001. ISBN 84-699-5329-X.

[3] L. Dominick. Aspect of Life-Cycle Control in a C++ Framework. In *Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'99*, Lisbon, Portugal, June 1999.

[4] W. Harison and H. Ossher. Subject-Oriented Programming (a Critique on Pure Objects). In *Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 411–428, Woshington, D.C., Sept. 1993. ACM.

[5] G. Kiczales, July 2001. Personal communications.

[6] G. Kiczales, E. Hilsdale, J. Hugonin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In J. L. Knudsen, editor, *ECOOP 2001 – Object-Oriented Programming*, volume 2072 of *LNCS*. Springer-Verlag, June 2001.

[7] H. Ossher and P. Tarr. Operation-Level Composition: A Case in (Join) Point. In *Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'98*, Brussels, Belgium, July 1998.

[8] O. Spinczyk and M. Urban. The PUMA Project Homepage, 2001. `http://ivs.cs.uni-magdeburg.de/~puma/`.

[9] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1997.

[10] M. Urban. The PUMA User's Manual, 2000. `http://ivs.cs.uni-magdeburg.de/~puma/`.

[11] E. D. Willink and V. B. Muchnick. Weaving a Way Past the C++ One Definition Rule. In *Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'99*, Lisbon, Portugal, June 1999.