AspectC++: An Aspect-Oriented Extension to C++ *

Olaf Spinczyk† **Andreas Gal**⋆ **Wolfgang Schröder-Preikschat**†

University of Magdeburg†
Universitätsplatz 2
39106 Magdeburg, Germany
Email: {olaf,wosch}@ivs.cs.uni-magdeburg.de

University of California, Irvine⋆
Department of Information and Computer Science
Irvine, CA 92717-3425, USA
Email: gal@uci.edu

Abstract

Small embedded systems are forced to operate under extreme resource constraints. At the same time these systems are very complex and many concerns in the implementation of such systems are of highly crosscutting nature. The concept of aspect orientation can be applied to allow a modular implementation of these concerns. With AspectJ[1] the first complete and powerful language extension for aspect-oriented programming (AOP) has been created. However, the costs of the Java run-time environment are not feasible for most embedded applications.

This paper presents AspectC++ (Gal & Spinczyk 2001), a new language extension to C/C++ that provides powerful language elements to facilitate aspect-oriented programming even in domains with tight resource limitations.

Keywords: Aspect-Oriented Programming, C++

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++. Especially most embedded systems cannot effort the run-time and code size overhead imposed by Java-based implementations, thus C/C++ is still the preferred implementation language. Efficiency is the crucial factor in this domain as the hardware costs are typically multiplied with a large number of pieces. 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 modular implementation of aspects. AspectJ (Kiczales, Hilsdale, Hugonin, Kersten, Palm & Griswold 2001) is an example for such a set of language extensions for the Java language. It combines the abstraction mechanisms of the object-oriented core language Java with new language features for aspect-oriented programming. 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 adapt and streamline what we learned from AspectJ according to the specific peculiarities of the C/C++ syntax and semantics.

The outline of this paper is as follows: Section 2 gives a brief introduction of the concept and vocabulary of aspect-oriented programming. In section 3 we will give an introduction to the language concepts using many code examples. The rationale of the language and a brief comparison with the corresponding language elements from AspectJ are given in section 4. Following in section 5 is a description of the architecture of our prototype implementation. In this section we also explain some of our efficiency considerations. Related work is discussed in section 6, followed by the conclusions in section 7.

2 Aspect-Oriented Programming

Aspect-oriented programming tries to solve the problem that often a single dimension of functional decomposition is not sufficient to implement all aspects of a program in a modular way. This means that code that stems from a single design decision is widely spread over the system. It cannot be encapsulated in a single function, class or module. This so called *aspect code* is tangled with the normal *component code* that fits into the functional decomposition scheme. A widely used example for this effect is the synchronization code in non-sequential programs, which is also quite common in the embedded systems domain. There is a separate design decision that says where synchronization code needs to be and what the code has to do, but it cannot be encapsulated cleanly.

AOP helps out of this dilemma, because an aspect-oriented language environment allows to implement such crosscutting concerns in modular units called *aspects* and a tool - the *aspect weaver* - inserts code fragments that are derived from the aspect code to wherever they are needed. These insertion points are called *join points*.

Figure 1 shows how aspect code is woven by the aspect weaver into component code. The generated code fragments are strongly tangled, but the source code is organized in a clean modular way.

Aspect weaving can be done at compile-time or at run-time. In this paper we will focus on *compile-time aspect weaving*, because the additional cost for having an aspect weaver in the target system performing run-time code manipulation is usually unacceptable for embedded systems with tight resource constraints.

Copyright ©2002, Australian Computer Society, Inc. This paper appeared at the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002), Sydney, Australia. Conferences in Research and Practice in Information Technology, Vol. 10. James Noble and John Potter, Eds. Reproduction for academic, not-for profit purposes permitted provided this text is included.

[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.

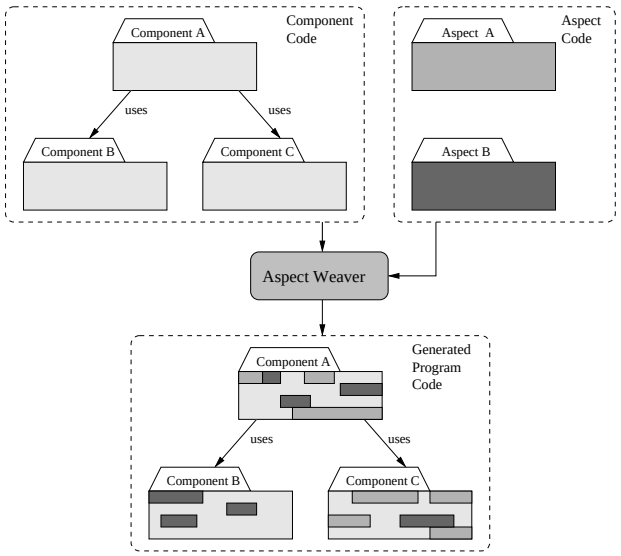

Figure 1: Weaving aspect code into component code

3 Introduction to AspectC++

3.1 Pointcuts

In AspectC++ join points are defined as points in the component code where aspects can interfere. A join point refers to a method, an attribute, a type (*class*, *struct* or *union*), an object, or a point from which a join point is accessed.

A *pointcut* is a set of join points described by a *pointcut expression*. Pointcut expressions are composed from *match expressions* used to find a set of join points, *pointcut functions* used to filter or map specific join points from a pointcut, and algebraic operators used to combine pointcuts.

Match expressions are strings containing a search pattern. AspectC++ can match type names as well as attribute and method signatures. The wildcard symbol "%" allows flexible searches. Figure 2 shows examples for match expressions.

Match expressions alone are not sufficient to clearly identify a join point. For example, consider a method match expression. It will match not only all implementations of matching methods, but also all calls to those methods. To filter specific types of join points AspectC++ supports pointcut functions. In figure 3 the most important built-in pointcut functions of AspectC++ are listed.

To support reuse of complex pointcut expressions AspectC++ allows to define named pointcuts. A named pointcut can contain formal arguments, which represent context exposed from the join points. This context information can be used by the aspect code, which is inserted at or called from the join point location. The following example shows the definition of a named pointcut *IRQ_level_call*. It refers to all calls of the function *void IRQ::level(int)* and exposes the single integer argument of that call.

```
pointcut IRQ_level_call(int level) =
  call("void IRQ::level(int)") && args(level);
```

Pointcuts are the key language element to deal with the crosscutting nature of aspects. They can describe points in the static structure or the dynamic control flow of a program in a very flexible way. There is no restriction to a specific function, class, module, or other unit of functional decomposition.

3.2 Advice

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

```
advice IRQ_level_call(level) :
  void after(int level)
{
  cout << "Interrupt level set to " << level << endl;
}
```

With this advice each call to *IRQ::level(int)* is followed by the execution of the advice code, which prints the number of the new interrupt level.

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

```
advice IRQ_level_call(level) :
  void around(int level)
{
  cout << "Interrupt level will change now." << endl;
  proceed();
  cout << "Interrupt level set to " << level << endl;
}
```

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

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

To obtain more information about the current join point, the advice code can use the object referenced by *thisJoinPoint*. The methods defined on *thisJoinPoint* provide access to static join point attributes like a string representation of the join point, the argument types, a unique ID, and the location of the join point in the source code as well as a dynamic part, which contains for example the current argument values of a call join point. This language feature allows to implement generic advice code. For instance, advice code can iterate over the arguments of a function that is unknown at implementation time. Possible applications are execution trace aspects or argument marshalling for a remote method invocation.

Even introductions can use *thisJoinPoint*. For example the following code fragment extends the introduction already presented above with printing of the class name.

```
advice all_classes() : void print() {
  cout << "Address of " << thisJoinPoint->toString()
       << " object: " << (void*)this << endl;
}
```

More details about the implementation of *thisJoinPoint* will be discussed in section 5.2.

3.3 Aspects

While named pointcut declarations can appear everywhere where declarations are allowed, 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 of C++. Additionally to attributes and methods, aspects may also contain advice declarations.

matching types:	
`"int"`	matches the built-in scalar type *int*
`"Chain"`	matches the class, struct or union *Chain*
`"Memory%"`	matches all classes, structs or unions having a name starting with *Memory*
matching attributes:	
`"Chain* Chain::next"`	matches the attribute *next* of the class *Chain*
`"% State::%"`	matching all attributes of any type of the class *State*
matching methods:	
`"void reset()"`	matches the function *reset* having no return value and no parameters
`"int main(int, char**)"`	matches the function *main* having exactly the given signature
`"% printf(...)"`	matches *printf* with any number and types of arguments and any result type
`"void %(int, %)"`	matches functions with any name, taking an int as first argument, and having a second argument of any type
`"void %::clear()"`	matches the *clear* methods without arguments of any class
`"void Mode::set%(...)"`	matches all methods of *Mode* having a name that starts with *set*

Figure 2: Examples for match expressions used to find join points

The example in figure 4 shows an aspect tracking the construction and destruction of objects of the class *Node*. The first advice in the aspect introduces both, an inner class *Counter* and a member variable *counter* of that type. Thus, when an object of the class *Node* is constructed or destructed, the constructor and destructor of *Counter* are also executed. Inside the constructor/destructor the static *aspectOf()* method of the aspect is invoked to access the aspect attributes. The second advice is used to initialize the counters of the aspect before *main()* is executed and to print the statistics after the execution of *main()* has finished.

This example aspect is very useful to find memory leaks. If another set of classes should be observed, it is sufficient to change the pointcut *clients*. It is even possible to implement more sophisticated aspects that record object addresses, locations of object instantiations, or detect invalid use of dynamic memory at run-time.

AspectC++ offers virtual pointcuts and aspect inheritance to support the reuse of aspects. A virtual pointcut can be redefined in a derived aspect and the inherited advice will use the new pointcut definition. Pointcuts can also be pure virtual. In this case the pointcut definition has to be overwritten by a derived aspect before it can be instantiated. Aspects can also inherit from a class, but it is not possible to derive a class from an aspect.

Figure 5 shows how the aspect from the previous example can be converted to a generalized abstract aspect. The named pointcut *clients* is converted into a pure virtual pointcut. This makes the aspect implicitly abstract and it will not be applied on the component code. A second aspect inherits from the generalized abstract and overwrites the *clients* pointcut turning the aspect from an abstract aspect into a concrete aspect, which will be applied to the component code by the aspect weaver.

4 Rationale of AspectC++

So far we have presented the basic concepts of AspectC++, from which a number are 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.

4.1 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 this problem the grammar of AspectC++ has been integrated into the C++ grammar. Figure 6 shows the grammar extensions. It should be understood as an extension to the C++ grammar listed by Stroustrup (Stroustrup 1997).

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.

4.2 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

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

refers to all calls to methods of class `FigureElement` with two integer arguments, 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 of 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

methods:	
call(*pointcut*)	*call* filters all join points from the *pointcut* that are method calls while *execution* extracts those join points referring to the method implementation. *callsto* gives all corresponding call join points to the execution join points in the argument pointcut.
execution(*pointcut*)	
callsto(*pointcut*)	
attributes:	
get(*pointcut*)	*get* and *set* select any attribute access join points from the *pointcut*. A field access through C++ references or pointers is not recognized.
set(*pointcut*)	
types:	
classes(*pointcut*)	*classes* delivers only the classes, structures, and unions from a join point. With *derived* it is possible to refer to all types in the pointcut expression and all classes derived from them. *base* can be used to find all base types of classes in a pointcut while *instanceof* can be used to locate objects of certain types.
derived(*pointcut*)	
base(*pointcut*)	
instanceof(*pointcut*)	
scope:	
within(*pointcut*)	*within* derives all join points declared in methods of types in the *pointcut*.
control flow:	
cflow(*pointcut*)	*cflow* captures join points occurring in the dynamic execution context of join points in the *pointcut*. *reachable* results in all join point from which the argument join points can be reached. This set is determined by a static analysis of the control flow graph and the class hierarchy.
reachable(*pointcut*)	
context:	
that(*type pattern*)	*that* returns all join points where the current C++ *this* pointer refers to an object which is an instance of the type described by *type pattern*. The type pattern can either be a pointcut containing type join points or the name of an formal argument of the pointcut. In the latter case the type is derived from the argument and the value of the argument is bound to the corresponding context information at run-time. *target* is very much similar than *that* but finds all join points where the target object of a call or object access is an instance of the type in *type pattern*. *args* receives a list of type patterns and filters all methods or attributes with a matching signature.
target(*type pattern*)	
args(*type pattern* , . . .)	

Figure 3: Pointcut functions in AspectC++

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 "::". For reasons we will discuss below, it is further required to specify pointcut designators as quoted strings – match expressions. The following example shows the AspectC++ version of the AspectJ pointcut shown above as it results from the described modifications:

```
call("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.

Simple regular expression matching is not sufficient to safely identify groups of signatures. Therefore the pointcut designators must be syntactically analyzed. 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.

4.3 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

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

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

introduces a new 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

```
aspect Instances {
  pointcut clients() = classes("Node");
  int created, destroyed;
  advice clients() :
    class Counter {
      public:
        Counter()  { Instances::aspectOf()->inc(); }
        ~Counter() { Instances::aspectOf()->dec(); }
    } counter;
public:
  void inc() { created++; }
  void dec() { destroyed++; }
  void reset() { created = destroyed = 0; }
  advice execution("% main(...)") : void around() {
    reset();
    proceed();
    cout << "Statistics: " << created << " created, " << destroyed
         << " destroyed." << endl;
  }
};
```

Figure 4: An aspect to track instantiation of objects

```
aspect AbstractInstances {
  virtual pointcut clients() = 0;
  ...
};

aspect NodeInstances : public AbstractInstances {
  pointcut clients() = classes("Node");
};
```

Figure 5: Aspect inheritance and virtual pointcuts

useful feature it is even possible to have virtual or pure virtual GTNs in AspectC++, thus introductions can be parameterized and reused by derived aspects.

- 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.

5 Implementation

5.1 Architecture

Our prototype implementation of a compiler for the AspectC++ language is a C++ preprocessor-like compiler based on PUMA (Spinczyk 2001). PUMA is a source code transformation system for C++.

The architecture of our AspectC++ compiler is shown in figure 7. First the AspectC++ source code is scanned, parsed and a semantical analysis is performed. Then the planning stage is entered.

In the planning stage the pointcut expressions are evaluated and the join point sets are calculated. In AspectC++ join points can represent locations in the control flow as well as static join points (i.e. classes). Therefore special attention has to be paid to the propagation of the join point type and advice execution

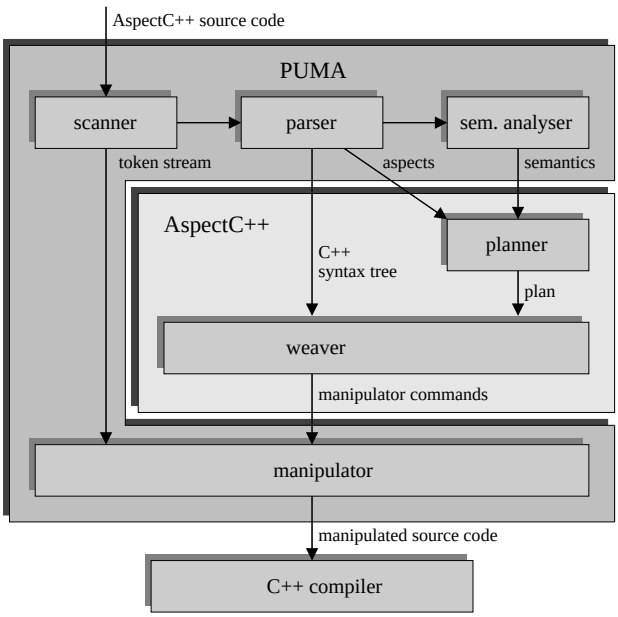

Figure 7: Architecture of the AspectC++ compiler

conditions when join point sets are merged. The planner performs an extensive type checking on the join point sets to ensure the syntactical and semantical correctness of the manipulated source code. Finally a plan for the weaver is created that contains join points and the operations to be performed at the join points (advice code) in a well defined order.

While the planning stage is mainly independent from the component language C++, the weaver is now responsible to transform the plan into concrete

aspect-name:
identifier

declaration:
pointcut-declaration
advice-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 6: The AspectC++ extensions to the C++ grammar

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 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. The AspectC++ specific parts of the compiler are implemented as a plug-in for the PUMA system.

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.

5.2 Run-time Overhead

Targeting the area of small embedded systems prohibits expensive run-time environments to support the programing language. Thus AspectC++ does not introduce any run-time or memory space overhead. Most of the language features like intercepting method calls and executions or object access can be implemented completely at compile-time.

Some of the pointcut functions like *that* and *target* require run-time type checking on objects. This is implemented by generating a virtual function in the class where the join point is located and redefined in the derived classes. The virtual function is used to identify the actual type of an object even if the join point is located in a base class. Of course this implementation causes some run-time overhead, but the cost can be directly related to a function that is needed in the program. AspectC++ does not force users to compile their code with C++ RTTI enabled, because this causes a global overhead regardless whether it is really used or not.

Another good example for our efficiency-aware design is the implementation of the "pseudo-object" *thisJoinPoint* (see section 3.2). Figure 8 shows a trace aspect that prints the name of each executed function of class *Interesting* and the class that is generated by AspectC++ to implement this aspect. The third part shows a fragment of the manipulated class *Interesting*.

A comparison of the first two code fragments shows that *thisJoinPoint* points to an instance of an individually generated class. AspectC++ generates the attributes of the class on demand, by doing a code analysis of the advice code.

The third fragment shows how the object is instantiated. In this example the object only contains a static part that can be completely allocated and initialized at compile-time.

6 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 "life-cycle control aspects when applying the Command Processor pattern" and complains "because no weaver technology was available, C preprocessor macros were used" (Dominick 1999).

A very interesting approach is followed by FOG (Willink & Muchnick 1999). FOG is a metacompiler 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 ex-

Aspect Code:

```
aspect Trace {
   pointcut methods() = execution("% Interesting::%(...)");
public:
   advice methods() : void before() {
      cout << "executing " << thisJoinPoint->toString());
   }
};
```

Generated code:

```
class Trace {
   // ...
public:
   class __Tjps__advice_0 {
      const char* _to_string;
   public:
      __Tjps__advice_0(const char* s) : _to_string(s) {}
      const char* toString() { return _to_string; }
   };
   void __advice_0(__Tjps__advice_0* thisJoinPoint) {
      cout << "executing " << thisJoinPoint->toString());
   }
};
```

Instantiation of the *thisJoinPoint* object:

```
class Interesting {
public:
   void __old_foo(int) {
      // ...
   }
   void foo(int arg0) {
      static Trace::__Tjps__advice_0 __tjp_Trace__advice_0
         ("void Interesting::foo(int)");
      TestTrace::aspectOf()->Trace::__advice_0(&__tjp_Trace__advice_0);
      __old_foo(arg0);
   }
   // ...
};
```

Figure 8: Implementation of thisJoinPoint

tended with a similar mechanism. While FOG seems to be ideally suited for subject-oriented programming (Harison & Ossher 1993)(Ossher & Tarr 1998) 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++ (Chiba 1995). 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 (Coady, Kiczales, Feeley, Hutchinson & Ong 2001) is an aspect-oriented extension to plain C, which is currently under development to study cross-cutting concerns in operating system code. It is not planned to support C++ component code with this implementation (Kiczales 2001). 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++ the AspectC++ language extension can be used with C as well.

7 Conclusions and Future Work

Much of the work presented in this paper is based on the concepts of AspectJ. This should enable users experienced with AspectJ and C++ to get familiar with AspectC++ without much effort. However, we have attempted to streamline the language extensions for the use in conjunction with C++ and added several new features. Special care has been taken to avoid unnecessary resource requirements, thus AspectC++ can be considered a powerful *and* efficient aspect-oriented programming language. This allows to combine the concepts of aspect-orientation and object-orientation even in application domains with tight resource limitations like the area of small embedded systems. In our own projects we use AspectC++ to implement the operating system family PURE (Beuche, Guerrouat, Papajewski, Schröder-Preikschat, Spinczyk & Spinczyk 1999), which even runs on 8-bit micro-controllers with only a few KBytes of memory.

The prototype implementation of an AspectC++ compiler is still in an early stage. The compiler cannot yet deal 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. AspectC++ and a set of working code examples is available for public download from `http://www.aspectc.org/`.

References

Beuche, D., Guerrouat, A., Papajewski, H., Schröder-Preikschat, W., Spinczyk, O. & Spinczyk, U. (1999), The PURE Family of Object-Oriented Operating Systems for Deeply Embedded Systems, *in* 'Proceedings of the 2nd IEEE Internation Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'99)', IEEE Computer Society, St. Malo, France.

Chiba, S. (1995), Metaobject Protocol for C++, *in* 'Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)', pp. 285–299.

Coady, Y., Kiczales, G., Feeley, M., Hutchinson, N. & Ong, J. S. (2001), Structuring Operating System Aspects, *in* 'Communications of the ACM', pp. 79 – 82.

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

Gal, A. & Spinczyk, O. (2001), 'The AspectC++ Project'. `http://www.aspectc.org/`.

Harison, W. & Ossher, H. (1993), Subject-Oriented Programming (a Critique on Pure Objects), *in* 'Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA)', ACM, Woshington, D.C., pp. 411–428.

Kiczales, G. (2001). Personal communications.

Kiczales, G., Hilsdale, E., Hugonin, J., Kersten, M., Palm, J. & Griswold, W. G. (2001), An Overview of AspectJ, *in* J. L. Knudsen, ed., 'ECOOP 2001 – Object-Oriented Programming', Vol. 2072 of *LNCS*, Springer-Verlag.

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

Spinczyk, O. (2001), 'The PUMA Project'. `http://ivs.cs.uni-magdeburg.de/~puma/`.

Stroustrup, B. (1997), *The C++ Programming Language*, Addison-Wesley.

Willink, E. D. & Muchnick, V. B. (1999), Weaving a Way Past the C++ One Definition Rule, *in* 'Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'99', Lisbon, Portugal.