The design and implementation of AspectC++

Olaf Spinczyk *, Daniel Lohmann

Friedrich-Alexander University Erlangen-Nuremberg, Computer Science 4, Martensstr. 1, 91058 Erlangen, Germany

Received 28 February 2007; accepted 1 May 2007
Available online 10 May 2007

Abstract

Aspect-Oriented Programming (AOP) is a programming paradigm that supports the modular implementation of crosscutting concerns. Thereby, AOP improves the maintainability, reusability, and configurability of software in general. Although already popular in the Java domain, AOP is still not commonly used in conjunction with C/C++. For a broad adoption of AOP by the software industry, it is crucial to provide solid language and tool support. However, research and tool development for C++ is known to be an extremely hard and tedious task, as the language is overwhelmed with interacting features and hard to analyze. Getting AOP into the C++ domain is not just technical challenge. It is also the question of integrating AOP concepts with the philosophy of the C++ language, which is very different from Java. This paper describes the design and development of the AspectC++ language and weaver, which brings fully-fledged AOP support into the C++ domain.

© 2007 Elsevier B.V. All rights reserved.

Keywords: AOP; C++; AspectC++; Programming languages

1. Motivation

The C/C++ programming language1 is frequently declared dead or at least dying. Actually, it is still the lingua franca in the real world of software industry. There are several reasons for the ongoing success of a language that is often criticized for its complexity. The most important one is the existing code base, which probably is of the dimension of some billion lines of code. Due to its multi-paradigm language design, C++ provides the unique ability to combine modern software development principles with high source-level backward compatibility to this enormous code base. It integrates classical procedural programming, object-oriented programming and, by the means of C++ templates, even generic and generative programming into a single language. Another main reason for the ongoing usage of C/C++ is runtime and memory efficiency of the generated code. For domains, where efficiency is a crucial property, there is still no alternative in sight.

In face of its importance in the real world, C/C++ plays a relatively small role in research activities and tool development related to Aspect-Oriented Programming (AOP) [17]. This is surprising, as the most frequently mentioned examples for aspects (synchronization, monitoring, tracing, caching, remote transparency, etc.), are particularly important concerns especially in the C/C++ dominated domain of (embedded) system software [24].

Probably the biggest problem in research and tool development for C++ is the language itself. On the technical level, one has to deal with the extraordinary complex syntax and semantics. On the conceptual level, one has to be very careful when integrating new concepts like AOP into a language that already supports multiple paradigms. In the following, we discuss some of the peculiarities and difficulties of C++ from the perspective of an aspect weaver.
1.1. Technical level

Developing a standard-compliant parser for the C++ syntax is just a nightmare. It is an extremely hard, tedious, and thankless task. Additionally, to support a substantial set of join point types, an aspect weaver has to perform a full semantic analysis, and the semantics of C++ is even worse. Pretty basic things, like type deduction and overload resolution, are very complicated due to automatic type conversions and namespace/argument dependent name lookup rules. However, all this gets a most significant raise in complexity if it comes to the support of templates. The C++ template language is a Turing-complete language on its own [8]. Template meta-programs are executed by the compiler during the compilation process. To support join points in template code, an aspect weaver has to do the same. It has to perform a full template instantiation.

Another problem is the C/C++ translation model that is built on the concept of single translation units, whereas an aspect weaver typically prefers a global view on the project. This and other oddities as the C preprocessor make it difficult to integrate aspect weaving into the C++ tool chain. Moreover, even if the language is standardized, in the real world one has to face a lot of compiler peculiarities. Proprietary language extensions, implementation bugs and “99% C++ standard conformance” are common characteristics among major compilers.

1.2. Conceptual level

C++ has a powerful but complex static type system, with fundamental and class types, derived types (pointer, array, reference, function), cv-qualifiers (const, volatile) and so on. On the other hand, C++ offers (almost) no runtime type information which facilitates a unified access to instances of any type at runtime. This focus on static typing has to be reflected in an aspect language for C++, e.g. in the way type expressions are matched and join point-specific context is provided. Moreover, C++ integrates multiple programming paradigms into a single language. Matching and weaving has not only to be supported in classes, but also in global functions, operator functions and template code.

1.3. Our contribution

AspectC++ is a general purpose aspect-oriented language extension to C++ designed by the authors. It is aimed to bring fully-fledged AOP support into the C++ domain. Since the first language proposal and prototype weaver implementation [22], AspectC++ has evolved significantly. While being strongly influenced by the AspectJ language model [16,15], AspectC++ supports in version 1.0 all the additional concepts that are unique to the C++ domain. This ranges from global functions, operators, const correctness and multiple inheritance up to weaving in template code and join point-specific instantiation of templates [19].

This paper describes the design and implementation of the AspectC++ language. On the conceptual level it shows, how we integrated AOP concepts into a language as complex as C++. On the technical level, some interesting details about the weaver implementation are presented. On both levels, the paper focuses on the peculiarities and difficulties discussed in the previous section – and how we solved them in AspectC++.

The paper is organized as follows: in the next section, we discuss related work. Section 3 then describes the primary design goals and rationale of AspectC++. Afterwards in Section 4, we concentrate on the conceptual level by describing the AspectC++ language and how it is integrated into C++. This is followed by two real-world examples in Section 5. Section 6 provides an overview on the weaver and thereby discusses, how we addressed the technical level. Based on benchmarks we furthermore discuss the resource consumption of AspectC++ programs. Finally, we draw a conclusion from our work in Section 7.

2. Related work

2.1. Aspect Languages

As already mentioned, AspectC++ adopts the language model introduced by AspectJ [16], which is probably the most mature AOP system for the Java language. The AspectJ language model is also used by AspectC, a proposed AOP language extension for C. Even though there is no weaver for AspectC available, it was successfully used to demonstrate the benefits of AOP for the design and evolution of system software [5,6].

2.2. Early AOP approaches

Most of the approaches considered as “roots of AOP”, like Subject-Oriented Programming [14], Adaptive Programming [18] or Composition Filters [3] provided a C++ language binding in the beginning. However, with the rising of Java, the C++ support was almost discontinued.

2.3. AOP in pure C++

A number of attempts have been suggested to “simulate” AOP concepts in pure C++ using advanced template techniques [7], macro programming [9] or Policy-based Design [1]. In some of these publications it is claimed that, in the case of C++, a dedicated aspect language like AspectC++ is not necessary. However, these approaches have the common drawback that a class has always to be explicitly prepared to be affected by aspects, which makes it hard to use them on existing code. Moreover, aspects have to be explicitly assigned to classes, as a pointcut concept is not available. To our understanding, an AOP language should not make any compromise regarding “obliviousness and quantification” [11]. The non-invasive
and declarative assignment of aspects to classes is at heart of Aspect-Oriented Programming.

2.4. Other C++ language extensions

OpenC++ [4] is a MOP for C++ that allows a compiled C++ meta-program to transform the base-level C++ code. The complete syntax tree is visible on the meta-level and arbitrary transformations are supported. OpenC++ provides no explicit support for AOP-like language extensions. It is a powerful, but somewhat lower-level transformation and MOP toolkit. Other tools based on C++ code transformation like Simplicissimus [21] and CodeBoost [2] are mainly targeted to the field of domain-specific program optimizations for numerical applications. While CodeBoost intentionally supports only those C++ features that are relevant to the domain of program optimization, AspectC++ has to support all language features. It is intended to be a general-purpose aspect language.

3. AspectC++ goals and rationale

3.1. Primary design goals

AspectC++ is being developed with the following goals in mind:

AOP in C++ should be easy.
We want practitioners to use AspectC++ in their daily work. The aspect language has to be general-purpose, applicable to existing projects and needs to be integrated well into the C++ language and tool chain.

AspectC++ should be strong, where C++ is strong.
Even though general-purpose, AspectC++ should specifically be applicable in the C/C++ dominated domains of “very big” and “very small” systems. Hence, it must not lead to a significant overhead at runtime.

3.2. Design rationale

The primary goals of AspectC++, as well as the properties of the C++ language itself, led to some fundamental design decisions:

“AspectJ-style” syntax and semantics,
as it is used and approved. Moreover, AspectJ was designed with very similar goals (e.g. “programmer compatibility” [16]) in mind.

Comply with the C++ philosophy,
as this is crucial for acceptance. As different as C++ is from Java, as different AspectC++ has to be from e.g. AspectJ.

```cpp
aspect TraceService {
  pointcut Methods () = "% Service:: % (...)";
  advice call (Methods ()) before () {
    cout << "Service function invocation" << endl;
  }
};
```

Source-to-source weaving,
as it is the only practicable way to integrate AspectC++ with the high number of existing tools and platforms. The AspectC++ weaver transforms AspectC++ code into C++ code.

Support of C++ templates,
even if it makes things a lot more difficult. The whole C++ standard library is build on templates, they are a vital part of the language.

Avoid using expensive C++ language features
in the generated code, like exception handling or RTTI, as this would lead to a general runtime overhead.

Careful, minimal extension of the C++ grammar,
as the grammar of C++ already is a very fragile building, which should not be shken more than absolutely necessary.

4. The language

The aim of this section is, to provide an overview of the AspectC++ language and to some of its concepts in detail. The AspectC++ syntax and semantics is very similar to AspectJ. The basics are therefore described only briefly here, which leaves more space to focus on C++-specific concerns and those parts of the language that are intentionally different from AspectJ.

4.1. Overview and terminology

AspectC++ is an extension to the C++ language. Every valid C++ program is also a valid AspectC++ program. As in AspectJ, the most relevant terms are join point and advice. A join point denotes a specific position in the static program structure or the program execution graph, where some advice should be given. Supported advice types include code advice (before, after, around), introductions (also known as inter-type declaration in AspectJ) and aspect order definitions. Join points are given in a declarative way by a join point description language. Each set of join points, which is described in this language, is called a pointcut. The sentences of the join point description language are called pointcut expressions. The building blocks of pointcut expressions are match expressions (to some degree comparable to “Generalized Type Names” (GTNs) and “Signature Patterns” in AspectJ), which can be combined using pointcut functions and algebraic operations. Pointcuts can be named and thereby reused in a different context. While named pointcuts can be defined anywhere in the program, advice can only be given by aspects. The aspect
(a) Syntax Extensions
The AspectC++ syntax is an extension to the C++ syntax defined in the ISO/IEC 14882:1998(E) standard.

class-key:
  aspect
declaration / member-declaration:
    pointcut-declaration
advice-declaration
slice-declaration
pointcut-declaration:
  pointcut declaration
pointcut-expression:
  constant-expression
advice-declaration:
  advice pointcut-expression : order-declaration
advice pointcut-expression : declaration
order-declaration:
  order { pointcut-expression-list }
pointcut-expression-list:
  pointcut-expression
pointcut-expression , pointcut-expression-list
slice-declaration:
  slice declaration

(b) Aspects
aspect A { .. ;
  defines the aspect A
aspect A : public B { .. ;
  A inherits from class or aspect B

(c) Advice Declarations
advice pointcut : before(...) (...) the advice code is executed before the join points in the pointcut
advice pointcut : after(...) (...) the advice code is executed after the join points in the pointcut
advice pointcut : around(...) (...) the advice code is executed in place of the join points in the pointcut
advice pointcut : order(high, ... low);
  high and low are pointcuts, which describe sets of aspects. Aspects on the left side of the argument list always have a higher precedence than aspects on the right hand side at the join points, where the order declaration is applied.
advice pointcut : slice class : public Base;
  introduces a new base class Base into the target classes matched by pointcut.
advice pointcut : slice class : public Base (...);
  introduces a new base class Base and new members.
advice pointcut : slice ASlice;
  introduces the slice ASlice into the target classes matched by pointcut. The slice (a class fragment) has to be defined separately in any class or namespace scope.

(d) Match Expressions
Type Matching
"int"
  matches the C++ built-in scalar type int
"\*"  matches any pointer type

Namespace and Class Matching
"Chain"
  matches the class, struct or union Chain
"Memory\*"
  matches any class, struct or union whose name starts with "Memory"

Function Matching
"void reset()"
  matches the function reset having no parameters and returning void
"\* pointf(...)"
  matches the function printf having any number of parameters and returning any type
"\* ... :\* (...)"
  matches any function, operator function, or type conversion function (in any class or namespace)
"\* ... :Service::\*[...].cons:"
  matches any const member-function of the class Service defined in any scope
"\* ... :operator \*(...)"
  matches any type conversion function

Template Matching
"std::set<...>
  matches all template instances of the class std::set
"std::set<int>
  matches only the template instance std::set<int>
"\* ... :\* ... :\* (...)"
  matches any member function from any template class in any scope

(e) Predefined Pointcut Functions

Functions
call(pointcut) N \rightarrow C:
  provides all join points where a named entity in the pointcut is called.
execution(pointcut) N \rightarrow C:
  provides all join points referring to the implementation of a named entity in the pointcut.
construction(pointcut) N \rightarrow C_constr:
  all join points where an instance of the given class(es) is constructed.
destruction(pointcut) N \rightarrow C_destr:
  all join points where an instance of the given class(es) is destructed.

pointcut may contain function names or class names. A class name is equivalent to the names of all functions defined within its scope qualified with the B operator (see below).

Fig. 1. AspectC++ language quick reference.

---

2. AspectJ supports with "++" a third wildcard for subtype matching. In AspectC++ this is realized by the derived() pointcut function.

... examples for match expressions can be found in Fig. 1d. A list of the pointcut functions and algebraic operations currently supported by AspectC++ can be found in Fig. 1e.

4.2. AspectC++ grammar extensions

Fig. 1a shows the AspectC++ extensions to the C++ grammar. Probably the most important design decision for keeping the set of grammar extensions small and simple was to use quoted match expressions. By quoting match expressions, pointcuts can be parsed with the ordinary C++ expression syntax. The real evaluation of the pointcuts itself can be postponed to a separate parser. This clear separation also helps the user to distinguish on the syntax...
level between ordinary code expressions and match expressions, which are quite different concepts. Additionally, it keeps the match expression language extendable.

4.3. The join point model

4.3.1. Join point types

AspectC++ uses a unified join point model to handle all types of advice in the same way. This is different from AspectJ, which distinguishes between pointcuts and advice on the one hand and GTNs and introductions on the other. As shown in the example above, in AspectC++ even match expressions are pointcuts and can be named. While such a coherent language design is a good thing anyway, this is particularly useful in combination with aspect inheritance and (pure) virtual pointcuts. In AspectC++, even the pointcuts used for introductions and baseclass introductions can be (pure) virtual and, thus, be defined or overridden in derived aspects. This is demonstrated in the “Reusable Observer” example in Section 5.1.

Regarding the implementation, the unified model requires join points to be typed in AspectC++. The type of a join point can be specified using a type or a class name. This allows for a more flexible and powerful join point model.

Fig. 1 (continued)
node in the program execution graph, like the call to or execution of a function. Code join points result from applying pointcut functions to name pointcuts. The basic types are additionally separated into more specialized subtypes like Class (N_C) or Function (N_F), Execution (C_E) or Construction (C_Const). The aspect weaver uses the type information to ensure that e.g. code advice (before, after, around) is only given to code join points. Fig. 1f lists all join point types. The subtypes are, however, mostly irrelevant for the user, as most pointcut functions accept the basic types and treat, for instance, a N_C join point as the set of N_F join points describing all member functions.

### 4.3.2. Order advice

Besides code advice (before, after, around) and (base-class-) introductions, AspectC++ supports with order advice a third type of advice. Order advice is used to define a partial order of aspect precedences per pointcut. This makes it possible to have different precedences for different join points. For example, the order declarations

```advice
    advice "Service": order("Locking", !("Locking" ["Tracing"], "Tracing"));
    advice "Client": order("Tracing", !"Tracing");
```

define that in the context of the class or namespace Service all advice given by the aspect Locking should be applied first, followed by advice from any aspect but Locking and Tracing. Advice given by Tracing should have the lowest precedence. However, for Client the order is different. Advice given by Tracing should be applied first. As order declarations are itself advice, they benefit from the unified join point model. They can be given to virtual pointcuts and can be declared in the context of any aspect. Hence, it is possible to separate the precedence rules from the aspects they affect. The AspectC++ weaver collects all partial order declarations for a join point and derives a valid total order. In case of a contradiction, a weave-time error is reported.

### 4.3.3. Matching C++ entities

As already mentioned in Section 1.2, C++ has a rather complex type system, consisting of fundamental types (int, short, etc.) and class types (class, struct, union), derived types (pointer, array, reference, function) and types optionally qualified by cv-qualifiers (const, volatile). Besides ordinary functions and member functions, C++ also supports overloading a predefined set of operator functions. Classes can define type conversion functions to provide implicit casts to other types. And finally, classes or functions can be parameterizable with template arguments.

The AspectC++ match expression language covers all these elements, because in C++ they are an integral part of a type’s or function’s name or signature. Hence, they should be usable as a match criteria. The match expression language is defined by an own grammar, which consists of more than 25 rules. We are therefore not going to describe it in detail, but present match expressions for function signatures as one example for AspectC++’s support for matching “C++ specific” entities.

```cpp
class IntArray{
    public:
        int count() const; // (1)
        void clear(); // (2)
        template <class T> IntArray &operator = (const T *data, int n); // (3)
        operator const Int * () const; // (5)
        operator Int * (); // (6)
    }
```

The Class IntArray consists of members that use cv-qualifiers (1 and 5), templates arguments 3, operator overloading 4, and type conversion functions (5 and 6). Table 1 demonstrates, how these elements are matched by various match expressions. Additional examples, including match expressions for type and scope, can be found in Fig. 1d.

## 4.3.4. Intentionally missing features

AspectC++ intentionally does not implement the get() and set() pointcut functions, known from AspectJ to give advice for field access. Even if desirable, they are not implementable in a language that supports free pointers. Field access through pointers is quite common in C/C++ and implies a danger of “surprising” side effects for such advice.

---

3 Often called type conversion *operators*, too, as they are defined using the `operator` keyword (e.g. “operator int()”).
4.4. Join point API

The join point API (Fig. 1g) is another part of AspectC++ that is heavily influenced by the “C++ philosophy”. Compared to Java, C++ has a less powerful runtime type system, but a more powerful compile-time type system. In Java, basically everything is a Java.lang.Object at runtime, which facilitates the development of generic code, as instances of any type can be treated as Object at runtime. In C++ there is no such common root class. C++, by the means of overloading and templates, facilitates the development of generic code that can be instantiated with any type at runtime. In general, Java promotes genericity at runtime,4 while the C++ philosophy is to use genericity at compile time. For this purpose, we extended the AspectJ idea of a runtime join point API by a compile-time join point API, which provides static type information about the current join point at compilation time.

The compile-time join point API is visible to advice code as class JoinPoint. Provided information includes, besides other type information, the sequence of argument types and the result type of the affected function. JoinPoint::Result is an alias for the function’s result type. The number of function arguments is available as compile-time constant JoinPoint::ARGS. The function argument types are provided through the template class JoinPoint::Arg<i>::Type. We intentionally used an integer template for this purpose, as it makes it possible to iterate at compile time over the sequence of argument types by template meta-programs. Such usage of the compile-time join point API is demonstrated in the “Win32 Errorhandling” example in Section 5.2.

The runtime join point API is visible to advice through the pointer tjp, which refers to an instance of JoinPoint. By using tjp, it is possible to retrieve the dynamic context, like the pointer to the actual result value (tjp->result()). Note that the function to retrieve the value of an argument is overloaded. If the index of the argument is known at compile-time, the template version tjp->arg<i>() can be used. It takes the index as template parameter and (later at runtime) returns a typed pointer to the value. Otherwise, the unsafe version tjp->arg(i) has to be used, which takes the index as a function parameter and returns an untyped void pointer to the value.

The class JoinPoint is not only specific for each join point, it is furthermore tailored down according to the individual requirements of the actual advice. If, for instance, tjp->result() is never called in the advice code, the function is removed from JoinPoint and no memory is occupied by the reference to the result value at runtime. This “pay only what you actually use” is important for facilitating AOP in the domain of embedded systems, where small memory footprints are a crucial concern.

In AspectC++, the join point API also implements the functionality to proceed to the intercepted original code from around advice (tjp->proceed()). It is also possible to store the context information of the intercepted function (returned by tjp->action()) and delegate its execution to another function or thread. This offers a noticeable higher flexibility than in AspectJ, where proceed() can only be called from the advice code itself.

4.5. Language summary

The previous sections presented only a subset of the AspectC++ language features. We left out details about (context binding) pointcut functions, algebraic operations, or the semantics of code advice, as they are very similar to AspectJ. Other elements, like the aspect instantiation, are different from AspectJ, but left out because of space limitations. Nevertheless, these features are available in AspectC++, too.

5. Examples

In the following sections, the expressive power of the AspectC++ language is demonstrated by two real-world examples. The first demonstrates using virtual pointcuts with baseclass introductions for a reusable implementation of the observer pattern. The second example is an aspect that checks the result codes of Win32 API functions and throws an exception in case of an error. It demonstrates how to use the compile-time join point API to exploit the power of C++ template meta-programming in advice code.

5.1. Reusable observer

Reusable implementations of design patterns are a well-known application of AOP [13]. The listing in Fig. 2 shows an AspectC++ implementation of the observer protocol [12]. The abstract aspect ObserverPattern defines interfaces ISubject and IObserver (lines 8–11), which are inserted via baseclass introductions into all classes that take part in the observer protocol (lines 22–23). These roles are represented by pure virtual pointcuts subjects() and observers(). Thus, their definition is delegated to derived concrete aspects. A third virtual pointcut, subjectChange(), describes all methods that potentially change the state of a subject and thereby should lead to a notification of the registered observers (line 17). The pointcut is defined as execution(“%...::%(...)” & & ”%...::%(... const)” & & within(subjects())). It evaluates to the execution of all non-const methods that are defined within a class from subject(). This is a reasonable default. However, it can be overridden in a derived aspect if, for instance, not all state-changing methods should trigger a notification. Finally, the notification of

---

4 This is even true with Java generics introduced in Java 5.0, which are basically a syntactic wrapper around the “treat everything as an object” philosophy.
observers is implemented by giving after execution advice to subjectChange() (lines 25–28).

The ClockObserver aspect is an example for a concrete aspect derived from ObserverPattern. To apply the pattern, the developer only has to define the pure virtual pointcuts subjects() and observers() (lines 41–42) and to write the introduction that inserts update() into the observer classes (lines 45–50).

"Reusable Observer" is a typical application of aspects in the world of object-orientation. The ObserverPattern implementation is even more generic than the AspectJ implementation suggested by Hannemann [13], where the derived aspect has to perform the baseclass introductions for the Observer and Subject interfaces. Purpose and name of these interfaces are, however, implementation details of the protocol and should be hidden. Moreover, the derived aspect has to define the subjectChange() pointcut in any case. In AspectC++ this is not necessary, as it is possible to take advantage from the C++ notion of non-modifying (const) methods in match expressions and thereby find all potentially state-changing methods automatically.
5.2. Win32 Errorhandling

Every program has to deal with the fact that operations might fail at runtime. Today, most developers favor exceptions for the propagation of errors. However, especially in the C/C++ world, there are still thousands of legacy libraries that do not support exception handling, but indicate an error situation via the function’s return value. In the Win32 API, for instance, an error is indicated by returning a special “magic value”. The corresponding error code (reason) can then be retrieved using the GetLastError() function. The actual “magic value” needed to check the result depends on the return type of the function. BOOL functions, for instance, return FALSE, while HANDLE functions return either NULL or INVALID_HANDLE_VALUE. The aim of the ThrowWin32Errors aspect (Fig. 3) is to perform the appropriate check after each call to a Win32 function and thereby transform the Win32 model of error handling into an exception-based model. This is actually very useful for developers that have to work with the Win32 API.

The ThrowWin32Errors aspect gives advice for all calls to functions defined by the win32::Win32API() pointcut (Fig. 3, lines 21–24). The advice code checks for an error condition using the win32::IsErrorResult() helper function. This function performs the check against the type-dependent “magic values”. It is overloaded for each return type used by Win32 functions, (lines 6–16). The compiler’s overload resolution deduces (at compile-time) for each join point the correct helper function to call. Note that this generic implementation of the advice code is only possible, because tjp->result() returns a pointer of the real (static) type of the affected function.

The win32::Exception object thrown in case of an error should include all context information that can be helpful to figure out the reason for the actual failure. The most tricky part to solve here is to build a string representation from the actual parameter values. In AspectJ one would iterate at runtime over all arguments and call Object.toString() on each argument. However, in C++ it is not possible to perform this at runtime, as C++ types do not share a common root class that offers generic services like toString(). The C++ philosophy of genericity is based on static typing. Retrieving a string representation of any object is realized by overloading the stream operator ostream& operator << (ostream&, T) for each type T. Therefore, we have to iterate at compile-time over the join point-specific list of argument types to generate a sequence of stream operator calls, each processing (later at runtime) an argument value of the correct type. This is implemented by a small template meta-program (lines 33–44), which is instantiated at compile-time with the JoinPoint type (line 39) and iterates, by recursive instantiation of the template, over the join point-specific argument type list JoinPoint::Arg<T>.

For each argument type, a stream_params class with a process() method is generated, which later at runtime will stream the typed argument value (retrieved via tjp->arg<I>() and recursively call stream_params::process() for the next argument (lines 35–37, 42–43). Again, the compiler automatically deduces the actual operator to call for a specific argument type during overload resolution.

The “Win32 Errorhandling” example shows, how aspects can be used with the procedural paradigm followed by C-style legacy libraries. It furthermore demonstrates, how advice code can take advantage of the generic and generative programming paradigm offered by C++ templates. A recent paper demonstrates, that this combination of AOP and templates can lead to very generic and efficient aspect implementations [19].

5.3. Examples summary

The “Reusable Observer” and “Win32 Errorhandling” examples show, how AspectC++ can be used with very different “styles” of C/C++ code, that is, with the different programming paradigms integrated into the C++ language. They also illustrate that certain AspectC++ concepts fit well into the C++ philosophy of static typing, which enables developers to write very expressive aspect code.

6. The weaver

The AspectC++ weaver ac++ is a source-to-source front-end that transforms AspectC++ programs into C++ programs. The woven code can then be built with any standard-conforming C++ compiler, like g++ or VisualC++. AspectC++ programs have already been executed on a broad variety of platforms, ranging from the smallest 8 bit micro-controllers to 64 bit servers. The following sections provide some details about the weaver implementation.

6.1. Translation process

A C++ program consists of a set of self-contained7 translation units. The translation process is performed in two steps. First, the compiler transforms each translation unit into an object file, which contains binary code augmented by symbol information that describes all externally visible and unresolved symbols. Then, the linker bundles all object files and creates the executable code by resolving all dependencies.

Various development tools like IDEs or program builders like make strongly rely on this two-step translation process. Thus, for the sake of easy integration, the ac++ command is called for each translation unit and produces C++ code that can be directly fed into the C++ compiler.

---

7 It is, of course, inconvenient to use template meta-programming to build just a string of argument values. However it is the only way of doing this in C++ at all.
6 The ac++ weaver and documentation are available from http://www.aspectc.org/.
7 Anything that is used either has to be defined or declared.
namespace win32 {
  struct Exception {
    Exception(const std::wstring w, DWORD c) { ... }
  };

  // Check for "magic value" indicating an error
  inline bool IsErrorResult(HWND res) {
    return res == NULL || res == INVALID_HANDLE_VALUE;
  }

  inline bool IsErrorResult(HWND res) {
    return res == NULL;
  }

  inline bool IsErrorResult(BOOL res) {
    return res == FALSE;
  }

  // Translates a Win32 error code into a readable text
  std::string GetErrorText(DWORD code) {
  }

  pointcut Win32API() = "% CreateWindow PARAMETERS"
    | "% BeginPaint PARAMETERS"
    | "% CreateFile PARAMETERS"
    | ...

  } // namespace Win32

aspect ThrowWin32Errors {

  // template metaprogram to generate code for
  // streaming a comma-separated sequence of arguments
  template<class TJP, int N>
  struct stream_params {
    static void process(ostream& os, TJP* tjp) {
      os << "tjp"->arg<TJP::ARGS - N>() << ", ";
      stream_params<TJP, N - 1>::process(os, tjp);
    }
  };

  // specialization to terminate the recursion
  template<class TJP>
  struct stream_params<TJP, 1> {
    static void process(ostream& os, TJP* tjp) {
      os << "tjp"->arg<TJP::ARGS - 1>();
    }
  };

  advice call{ win32::Win32API() } : after() {
    if( win32::IsErrorResult("tjp"->result() ) ) {
      ostringstream os;
      DWORD code = GetLastError();
      os << "WIN32 ERROR " << code << ": "
     << win32::GetErrorText(code) << endl;
      os << "WHILE CALLING: "
        << tjp->signature() << endl;
      os << "WITH: " << ";"
    }
  }

  // Generate joinpoint-specific sequence of
  // operations to stream all argument values
  stream_params<JoinPoint,>
    JoinPoint::ARGS::process(os, tjp);
    os << "");
  throw win32::Exception(os.str(), code);
  }
};

Fig. 3. An aspect to throw Win32Errors as exceptions.
6.1. Visibility of aspects

Aspects should be able to affect code in any translation unit. Therefore, a mechanism is needed to include the definition of an aspect in all translation units. Programmers should not be forced to include the definition by hand using the C++ preprocessor directive `#include`. This would violate the “obliviousness” goal. Therefore, we adopted the “forced include” mechanism known from many C++ compilers for that purpose. In practice, this means that aspect definitions are stored in “aspect header files” (*.ah). The location of these files is provided on the command line and the weaver automatically includes all aspect headers in the currently processed translation unit.

6.1.2. Link-once code

Traditionally, a C/C++ linker does not accept two externally visible symbols with the same name to be defined in two different translation units. This is problematic for a C++ aspect weaver, because there are many situations, in which global objects have to be generated. Examples are the instances of singleton aspects and introductions of static attributes or non-inline functions. As the ac++ weaver always processes only a single translation unit, there is no global knowledge, which would help to find the right place for inserting the generated code. To solve this problem, ac++ exploits the so-called COMDAT feature of state-of-the-art C++ compilers. A standard-compliant C++ compiler sometimes has the same problem as ac++. For example, non-inline member functions or static attributes of template classes are allowed to be defined in header files. To avoid the problem of duplicate symbols the compiler and linker use “vague linkage”. Thus, by using certain code generation patterns COMDAT can also be used by the weaver. In most cases, generated global code is transparently wrapped by a template class.

6.1.3. Information sharing

To provide at least a “partial global” knowledge ac++ accesses a global project repository. This is a file, which describes processed translation units by listing join points, aspects and advice. It can be used by one ac++ incarnation to save information for other ac++ incarations running later. For example, it would be impossible to provide a project-wide unique ID (Fig. 1g) for each join point without the project repository. As a side-effect the project repository can also be used by IDEs, like the Eclipse ACDT,8 to visualize join points.

6.2. Architecture and implementation

Fig. 4 illustrates the architecture of ac++ by showing the main building blocks and the data flow during a program transformation from AspectC++ to C++. The weaver implementation is based on PUMA, a framework for C++ code analysis and transformation. PUMA is developed in-line with AspectC++ by our group. It contains a complete C++ front-end, which supports standard ISO C++ 98 as well as many g++ and VisualC++ language extensions. The ac++ weaver has a weight of 85 ksloc, from which are 70 ksloc used solely by the PUMA framework, that is for analyzing and manipulating C++ code.

The weaving process starts in PUMA with scanning, preprocessing, parsing, and a full semantic analysis of the source code. The semantic analysis includes complete function call resolution, necessary to implement call advice, and full template instantiation, needed for matching and weaving in template instances.

The ac++ level first processes the resulting syntax tree. The Tracker Dog is responsible to find all points in the code, which might be affected by advice. The resulting

---

potential join points are passed to the Planner, which also uses the Aspect & Class Database from the Parser and Semantics (PUMA level). The planner internally parses and analyses pointcut expressions (including pointcut type checks) and calculates the sets of matching join points. For each join point the planner sets up a plan that is later used forming static code analysis on the advice and templates instantiated by advice (with JoinPoint as a template parameter) ac++ avoids to generate unneeded elements in this class. The following code fragment shows a part of the JoinPoint structure for a call join point in the "Win32 Errorhandling" example.

```c++
struct TJP_WndProc_1 {
  ...
  template <int I> struct Arg {
    typedef void Type;
    typedef void ReferredType;
  };
  template <> struct Arg<0> {
    typedef HWND Type;
    typedef HWND ReferredType;
  };
  ...
  void **_args;
  inline void *arg(int n) return _args[n];
  template <int I> typename Arg<I>::ReferredType *arg () {
    return (typename Arg<I>::ReferredType *)(_args[I]);
  }
};
```

by the Weaver to generate a sequence of code manipulation commands. The code manipulation is performed again on the PUMA-level by the Manipulator.

The PUMA framework itself is implemented in AspectC++. Aspects are, for instance, used to adapt the system to compiler-specific peculiarities. The PUMA core implements only the standard C++ grammar. All additional compiler-specific grammar extensions are woven in by aspects. Currently such grammar extensions are implemented for VisualC++ (10 aspects, 12 introductions, 13 execution advice) and g++ (1 aspect, 15 introductions, 15 execution advice).

6.3. Code generation

6.3.1. Joinpoint structure

AspectC++ generates a C++ class with a unique name for each join point that is affected by advice code. By per-

6.3.2. Advice transformation

Advice code is transformed into a member function of the aspect, which in turn is transformed to a class. If the advice implementation depends on the JoinPoint type, it is transformed into a template member function and the unique join point class is passed as a template argument to the advice code. Thus, in this case the advice code is generic and can access all type definitions (C++ typedefs) inside the join point class with JoinPoint::Typename, as described in Section 4.4. The following code fragment shows JoinPoint dependent advice code after its transformation into a template function.

```c++
class ThrowWin32Errors {

  // ...
  template <class JoinPoint>
  void __a0_after(JoinPoint *tjp) {
    if(win32::IsErrorResult(*tjp->result())) {
      // ...
    }
  }

6.3.3. Weaving in regular code

Weaving of call or execution advice is based on inlined wrapper functions. For instance, in the “Win32 Errorhandling” example the after call advice for BeginPaint() is implemented by replacing the call
expression  BeginPaint(NULL,&ps) by __call_WndProc_1_0(NULL,&ps). The wrapper function calls BeginPaint() first and invokes the advice afterwards.

inline:: HDC __call_WndProc_1_0 (:: HWND arg0,
:: LPPAINTSTRUCT arg1 {

:: HDC result;
void *args_WndProc_1[] = {(void*)&arg0,(void*)&arg1];
TJP_WndProc_1 tjp_WndProc_1 = {args_WndProc_1,&result};
result = :: BeginPaint(arg0, arg1);
AC:: invoke_ThrowWin32Errors_ThrowWin32Errors_a0_after<
    TJP_WndProc_1> (&tjp_WndProc_1);
return (:: HDC) result;
}

Although generating a wrapper function seems straightforward, call advice weaving is a complex transformation. For example, a call can syntactically be expressed in numerous ways in C++. Specific transformation patterns are needed for unary and binary operator calls. Even invisible calls by implicitly called conversion functions have to be considered.

6.3.4. Weaving in template code
AspectC++ supports advice for join points associated with individual template instances. Therefore, the weaver has to perform a full template instantiation analysis to distinguish template instances and to compare their signatures with match-expressions. To be able to affect only certain instances on the code generation level, our weaver uses the explicit template specialization feature of C++. For example, if advice affects only the instance container<int> the template code of container is copied, manipulated according to the advice and the instantiation, and declared as a specialization of container for int as shown here:

```cpp
template <class ElementType> class container {
public:
    void insert (ElementType elem) {...}
};
```

```cpp
namespace AC{typedef int t_container_0;}
template <> class container<AC:: t_container_0> {
public:
    inline void __exec_old_insert(AC:: t_container_0 elem){...}
    void insert (AC:: t_container_0 arg0) {
        AC:: invoke_MyAspect_a0_before ();
        this->__exec_old_insert(arg0);
    }
};
```

9 C++ does not support the explicit specialization for template functions. However, we can work around this problem by defining a helper template class. Furthermore, some compilers do not support explicit specialization in non-namespace scope. We handle this problem by using partial specialization with an extra dummy argument.

6.4. Overhead
AspectC++ is an AOP-extension for the C++ language, specifically aimed for the application of AOP in resource-constrained environments such as embedded systems. A major goal of AspectC++ is cost efficiency in the generated code. Therefore, the AspectC++ compiler follows a source-to-source weaving approach with generation of code patterns that (1) do not use "expensive" C++ language elements (such as RTTI or exceptions), and (2) can well be optimized by current C++ compilers.

To evaluate the overhead induced by the `ac++` generated code for various AspectC++ language features, we conducted a series of benchmarks. The condensed results are depicted in Table 2. The upper part shows the results of the measurements performed with `g++` as back-end compiler, while we used the even better optimizing Intel C++ compiler `icc` for the measurements in the lower part. For every test we measured the consumed CPU time (clock cycles), dynamic memory consumption (stack, bytes), and static memory consumption (code/data, the size of the whole test program in bytes).
The ground overhead of applying advice to a parameterless function is very low (Table 2a). On a Pentium 3 with g++, advice invocation takes only two cycles, independent from the type of advice (before/after/around for call/execution join points), the join point type (call/execution), and even the number of aspects giving advice to the join point (Table 2b). This is noticeable, as in AspectJ, for instance, around advice induces significantly higher costs than before/after advice [10]. The size of the text segment (code) remains also stable, the increase by 16 bytes in one case was mainly caused by linker alignment of the affected section.

With the icc compiler the variation of the results is higher, but in the most cases they are even better than the g++ results. For example, in the simple incrementer test scenario there are cases in which there is no and sometimes even a “negative overhead”.

While advice for parameterless functions does not lead to additional stack costs, the stack space allocated by the compiler to pass call-by-value parameters is actually doubled. This becomes evident from the plain lines in Table 2c, which represent the costs of a member function call with 0–2 int parameters for that some advice was given (Table 2c numbers are not directly comparable with those from Table 2a and Table 2b, as they include the costs of the method call itself). Each additional 4-byte int parameter increases the absolute stack costs (by 8 bytes with g++ and 8–16 bytes with icc). The reason turned out to be a limitation of the inliner/optimizer: Whenever a function is inlined, the compiler ensures call-by-value semantics by pushing an extra copy of all function parameters on the stack. In most cases, the parameter passing code is later replaced by the optimizer with direct register access, but the (now completely useless) stack reservation remains in the code.

The overhead to retrieve join point specific context is quite low. Compared to plain advice, only 0–6 extra cycles are consumed to provide access to context with g++ and even less with icc. Accessing context which is implicitly available at the join point (such as argument and result values) does furthermore not induce any additional stack costs. The required join point context data generated for this purpose, such as the array of argument references (see Section 6.3.3), is later replaced by the optimizer with direct access to the referenced parameters. Only “extra” context, such as the pointer to the affected instance (tjp->that()) or the pointer to the target of the call (tjp->target()), requires additional stack space (4 bytes).

The results show furthermore the benefits of the context tailoring performed by ac++. The additional 4 stack bytes

---

### Table 2

**AspectC++ µ-benchmark results**

<table>
<thead>
<tr>
<th>Resource consumption with g++ 3.3.5</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>a)</strong> incremener</td>
</tr>
<tr>
<td>advice</td>
</tr>
<tr>
<td>cycles Δ</td>
</tr>
<tr>
<td>stack Δ</td>
</tr>
<tr>
<td>code Δ</td>
</tr>
<tr>
<td>tangled</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>4128</td>
</tr>
<tr>
<td>before execution</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>4128</td>
</tr>
<tr>
<td>after execution</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>4128</td>
</tr>
<tr>
<td>around</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>4128</td>
</tr>
<tr>
<td><strong>b)</strong> multispect</td>
</tr>
<tr>
<td>advice</td>
</tr>
<tr>
<td>cycles Δ</td>
</tr>
<tr>
<td>stack Δ</td>
</tr>
<tr>
<td>code Δ</td>
</tr>
<tr>
<td>before</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>4080</td>
</tr>
<tr>
<td>after</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>4080</td>
</tr>
<tr>
<td>around</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>4096</td>
</tr>
<tr>
<td><strong>c)</strong> parameters, jp-api</td>
</tr>
<tr>
<td>advice</td>
</tr>
<tr>
<td>cycles Δ</td>
</tr>
<tr>
<td>stack Δ</td>
</tr>
<tr>
<td>code Δ</td>
</tr>
<tr>
<td>plain, n=0</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>3968</td>
</tr>
<tr>
<td>that()</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>20</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>3968</td>
</tr>
<tr>
<td>target()</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>20</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>3968</td>
</tr>
<tr>
<td>result()</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>3968</td>
</tr>
<tr>
<td>plain, n=1 arg=0:()</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>24</td>
</tr>
<tr>
<td>3968</td>
</tr>
<tr>
<td>plain, n=2 arg=1:()</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>32</td>
</tr>
<tr>
<td>3984</td>
</tr>
<tr>
<td><strong>Resource consumption with icc 9.0</strong></td>
</tr>
<tr>
<td><strong>a)</strong> incremener</td>
</tr>
<tr>
<td>advice</td>
</tr>
<tr>
<td>cycles Δ</td>
</tr>
<tr>
<td>stack Δ</td>
</tr>
<tr>
<td>code Δ</td>
</tr>
<tr>
<td>tangled</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>6372</td>
</tr>
<tr>
<td>before execution</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>6372</td>
</tr>
<tr>
<td>after execution</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>6372</td>
</tr>
<tr>
<td>around</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>6372</td>
</tr>
<tr>
<td><strong>b)</strong> multispect</td>
</tr>
<tr>
<td>advice</td>
</tr>
<tr>
<td>cycles Δ</td>
</tr>
<tr>
<td>stack Δ</td>
</tr>
<tr>
<td>code Δ</td>
</tr>
<tr>
<td>before</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>6424</td>
</tr>
<tr>
<td>after</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>-1</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>6340</td>
</tr>
<tr>
<td>around</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>6340</td>
</tr>
<tr>
<td><strong>c)</strong> parameters, jp-api</td>
</tr>
<tr>
<td>advice</td>
</tr>
<tr>
<td>cycles Δ</td>
</tr>
<tr>
<td>stack Δ</td>
</tr>
<tr>
<td>code Δ</td>
</tr>
<tr>
<td>plain, n=0</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>6372</td>
</tr>
<tr>
<td>that()</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>-1</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>6372</td>
</tr>
<tr>
<td>target()</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>6372</td>
</tr>
<tr>
<td>result()</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>6372</td>
</tr>
<tr>
<td>plain, n=1 arg=0:()</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>20</td>
</tr>
<tr>
<td>6404</td>
</tr>
<tr>
<td>plain, n=2 arg=1:()</td>
</tr>
<tr>
<td>17</td>
</tr>
<tr>
<td>28</td>
</tr>
<tr>
<td>6420</td>
</tr>
</tbody>
</table>

---

(a) Costs of incrementing a global counter either in the body of a function void f() (tangled) or by giving advice (before/after/around for call/execution join points) to the same function. Δ denotes the difference to tangled.

(b) Scaling of costs if 1–3 aspects give around advice to the same call/execution join point. Δ denotes the difference to the previous line.
to store the pointer returned by $\text{tjp->that()}$, for instance, are only consumed if the advice code actually uses $\text{tjp->that()}$.

Overall, AOP with AspectC++ does not lead to an extra overhead that makes it per se unacceptable even for resource constrained environments such as deeply embedded systems or system software. In combination with an optimizing C++ compiler, the static AspectC++ weaver $\text{ac++}$ generates efficient code. In $\mu$-benchmarks, the overhead of simple before/after/around-advice for call() and execution() join points is practically null, the overhead for accessing advice context is very low, even though in some cases the stack overhead turned out to be higher than necessary. To get such results the compiler needs to provide some basic optimization capabilities. The most important are (1) embedding of functions explicitly marked as inline and (2) performing a local alias analysis to detect and remove unnecessary parameter copies. Without any optimization (especially function embedding), the resulting code would be much worse. This is, however, a quite unrealistic scenario, as such basic optimizations are available and used with almost every C++ compiler.

7. Summary and conclusions

In this paper, we described our work on the design and development of AspectC++, an AOP language extension and weaver for C++. We motivated our work with the ongoing significance of C++ in software industry. Research and tool development for C++ is hard. We examined, from the perspective of an AOP language designer, some of the major peculiarities of C++ and categorized them into a conceptual level (language) and a technical level (tools).

On the conceptual level, we emphasized that an AOP extension for C++ has to fit into the philosophy of C++. Multi-paradigm programming, the focus on static typing and compile-time genericity, as well as backward compatibility to existing code are the fundamental elements of this philosophy. In AspectC++, this is addressed in many places, but particularly by the match expression language, the (static) join point API and the code generated by the weaver. AspectC++ thereby integrates AOP well into the C++ language, which was also demonstrated in the examples.

On the technical level, we discussed some of the major difficulties regarding tool development for C++. We pointed out that an aspect weaver benefits from a fully fledged syntax/semantics analysis, which is, however, a very tedious task to implement. The complexity of the language and the (anachronistic) translation model put a heavy burden on the weaver implementation. We presented some details of the implementation and demonstrated, how the weaver transforms AspectC++ code into C++ code. The overhead of AspectC++ code in comparison with "tangled" implementations is minimal.

Today, AspectC++ is already used by researchers from academia and industry. Currently, 163 people, most of them from companies in the telecommunications or embedded systems area, are subscribed on the $\text{ac++}$ user mailing list. The most prominent academic applications can be found in the domain of tailorable embedded databases [25], namely the Berkeley DB, and operating systems, which is our main field of research. We use AspectC++ in our pure and CiAO operating system product lines [20,23].

Regarding future work, we will continue working on the template support. This has evolved a lot over the last months, however, is still considered "experimental". We also plan to extend weaver in order to support plain C applications. Weaving in macro-generated code is another feature that will be addressed in near future. However, AspectC++ is almost feature-complete. We are convinced that it is now ready for a broad adoption.

References


12 Actually, $\text{ac++}$ is already able to weave in C code, but the generated code has always to be compiled with a C++ compiler.


