The Design and Implementation of AspectC++ ⋆

Olaf Spinczyk * and Daniel Lohmann

Friedrich-Alexander University Erlangen-Nuremberg

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.

Key words: AOP, C++, AspectC++, Programming Languages

1 Motivation

The C/C++ programming language [1] 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

⋆ This work was supported by the German Research Council (DFG) under grant no. SCHR 603/4 and SP 968/2-1.

* Correspondence to: Olaf Spinczyk, University of Erlangen-Nuremberg, Computer Science 4, Martensstr. 1, 91058 Erlangen, Germany, Tel.: +49 9131 8527906

Email addresses: Olaf.Spinczyk@informatik.uni-erlangen.de, Daniel.Lohmann@informatik.uni-erlangen.de (Daniel Lohmann).

[1] Throughout this paper, we use "C/C++" as an abbreviation for "C and C++" and refer to it as a single language. C, as it is used today, is basically a subset of C++ (ignoring some minor differences).

FINAL DRAFT to appear in: Journal on Knowledge-Based Systems,
Special Issue on Creative Software Design, Elsevier

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...), 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++ metaprogram 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.

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 shaken 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

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

gives *before advice* to all calls to functions defined by the pointcut `Methods()`, which is in turn defined as all functions of the class or namespace `Service`, like `void Service::foo()` or `int Service::bar(char*)`. The special `%` and `...` symbols are wildcards, comparable to `*` and `..` in AspectJ. The percent wildcard (`%`) matches any name (or a part of a name) of a C++ entity. The ellipsis (`...`) matches any sequence of argument types (including the C `va_arg` argument type used by

functions like `printf`), namespaces or template parameters. [2] More examples for match expressions can be found in Figure 1-d. A list of the pointcut functions and algebraic operations currently supported by AspectC++ can be found in Figure 1-e.

4.2 AspectC++ Grammar Extensions

Figure 1-a 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 basic join point types are *Name* (**N**) and *Code* (**C**). A name join point represents a named entity from the C++ program, like a class, a function or a namespace. It typically results from a match expression. A code join point represents a 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* ($\mathbf{N}_C$) or *Function* ($\mathbf{N}_F$), *Execution* ($\mathbf{C}_E$) or *Construction* ($\mathbf{C}_{Cons}$). The aspect weaver uses the type information to ensure that e.g. code advice (before, after, around) is only given to code join points. Figure 1-f lists all join point types. The subtypes are, however, mostly irrelevant

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

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`

`"% printf(...)"`
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::%(...) const"`
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*) $\mathbf{N \rightarrow C}_{C}$
provides all join points where a named entity in the *pointcut* is called.

execution(*pointcut*) $\mathbf{N \rightarrow C}_{E}$
provides all join points referring to the implementation of a named entity in the *pointcut*.

construction(*pointcut*) $\mathbf{N \rightarrow C}_{Cons}$
all join points where an instance of the given class(es) is constructed.

destruction(*pointcut*) $\mathbf{N \rightarrow C}_{Des}$
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 combined with the || operator (see below).

Fig. 1. AspectC++ Language Quick Reference

Control Flow

cflow(*pointcut*) **C→C**
captures join points occuring in the dynamic execution context of join points in the *pointcut*. The argument pointcut is forbidden to contain context variables or join points with runtime conditions (currently cflow, that, or target).

Types

base(*pointcut*) **N→$N_{C,F}$**
returns all base classes resp. redefined functions of classes in the *pointcut*

derived(*pointcut*) **N→$N_{C,F}$**
returns all classes in the *pointcut* and all classes derived from them resp. all redefined functions of derived classes

Context

that(*type pattern*) **N→C**
returns all join points where the current C++ `this` pointer refers to an object which is an instance of a type that is compatible to the type described by the *type pattern*

target(*type pattern*) **N→C**
returns all join points where the target object of a call is an instance of a type that is compatible to the type described by the *type pattern*

result(*type pattern*) **N→C**
returns all join points where the result object of a call/execution is an instance of a type described by the *type pattern*

args(*type pattern, ...*) **(N,...)→C**
a list of *type patterns* is used to provide all joinpoints with matching argument signatures

Instead of the *type pattern* it is possible here to pass the name of a **context variable** to which the context information is bound. In this case the type of the variable is used for the type matching.

Scope

within(*pointcut*) **N→C**
filters all join points that are within the functions or classes in the *pointcut*

Algebraic Operators

pointcut **&&** *pointcut* **(N,N)→N, (C,C)→C**
intersection of the join points in the *pointcuts*

pointcut **||** *pointcut* **(N,N)→N, (C,C)→C**
union of the join points in the *pointcuts*

! *pointcut* **N→N, C→C**
exclusion of the join points in the *pointcut*

f) Join Point Types

Code

C, C_C, C_E, C_{Cons}, C_{Des} :
any, *Call*, *Execution*, *Construction*, *Destruction*

Name

N, N_N, N_C, N_F, N_T :
any, *Namespace*, *Class*, *Function*, *Type*

g) Join Point API

The JoinPoint-API is provided within every advice code body by the built-in object ***tjp*** of class ***JoinPoint***.

Compile-time Types and Constants

That [type]
object type (object initiating a call)

Target [type]
target object type (target object of a call)

Result [type]
result type of the affected function

Arg<i>::Type, Arg<i>::ReferredType [type]
type of the i^{th} argument of the affected function (with $0 \leq i < ARGS$)

ARGS [const]
number of arguments

JPID [const]
unique numeric identifier for this join point

JPTYPE [const]
numeric identifier describing the type of this join point (*AC::CALL* or *AC::EXECUTION*)

Runtime Functions and State

*static const char **__signature__()
gives a textual description of the join point (function name, class name, ...)

*That **__that__()
returns a pointer to the object initiating a call or 0 if it is a static method or a global function

*Target **__target__()
returns a pointer to the object that is the target of a call or 0 if it is a static method or a global function

*Result **__result__()
returns a typed pointer to the result value or 0 if the function has no result value

*Arg<i>::ReferredType **__arg__()
returns a typed pointer to the argument value with compile-time index *number*

*void **__arg__(*int number*)
returns a pointer to the memory position holding the argument value with index *number*

void __proceed__()
executes the original code in an around advice

*AC::Action &*__action__()
returns the runtime action object containing the execution environment to execute (*trigger()*) the original code encapsulated by an around advice

Runtime Type Information

static AC::Type __type__()

static AC::Type __resulttype__()

static AC::Type __argtype__(*int* i)
return a C++ ABI V3 conforming string representation of the signature / result type / argument type of the affected function

for the user, as most pointcut functions accept the basic types and treat, for instance, a $\mathbf{N}_C$ join point as the set of $\mathbf{N}_F$ join points describing all member functions.

4.3.2 Order Advice

Besides code advice (before, after, around) and (baseclass-) 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 "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`, ...) 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* [3] 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.

[3] often called type conversion *operators*, too, as they are defined using the `operator` keyword (e.g. "`operator int()`")

Expression	Matches	
`"% IntArray::%(...)"`	**1–6**	Any function, operator function, or type conversion function in IntArray
`"% IntArray::%(...) const"`	**1, 5**	Any *const* -function, -operator function, or -type conversion function in IntArray
`"% IntArray::%<...>(...)"`	**3**	Any instance of any *template* -function, -operator function, or -type conversion function in IntArray
`"% IntArray::%<short>(...)"`	**(3)**	Any *<short> instance* of any *template* -function, -operator function or -type conversion function in IntArray
`"...::operator %(...)"`	**5, 6**	Any type conversion function
`"...::operator %*(...)"`	**5, 6**	Any type conversion function that converts to a *pointer type*
`"% IntArray::operator =(...)"`	**4**	Any assignment operator in IntArray

Table 1
Matching of C++-specific function signatures

```
class IntArray {
public:
  int count() const;                                              (1)
  void clear();                                                   (2)
  template< class T > init( const T* data, int n );               (3)
  IntArray &operator =( const IntArray &src );                    (4)
  operator const int*() const;                                    (5)
  operator int*();                                                (6)
};
```

The Class `IntArray` consists of members that use cv-qualifiers (1, 5), templates arguments (3), operator overloading (4), and type conversion functions (5, 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 Figure 1-d.

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.

```
File: ObserverPattern.ah

aspect ObserverPattern {
  // data structures to manage subjects and observers
  ...
public:
  // Interfaces for each role
  struct ISubject {};
  struct IObserver {
    virtual void update (ISubject *) = 0;
  };

  // To be defined / overridden by the concrete derived aspect
  // subjectChange() matches execution of all non-const methods
  pointcut virtual observers() = 0;
  pointcut virtual subjects() = 0;
  pointcut virtual subjectChange() =  execution( "% ...::%(...)"
            && !"% ...::%(...) const" ) && within( subjects() );

  // Add new baseclass to each subject/observer class
  // and insert code to inform observers after a subject change
  advice observers () : slice class : public ObserverPattern::IObserver;
  advice subjects()   : slice class : public ObserverPattern::ISubject;

  advice subjectChange() : after () {
    ISubject* subject = tjp->that();
    updateObservers( subject );
  }
       void updateObservers( ISubject* subject ) { ... }
  void addObserver( ISubject* subject, IObserver* observer ) { ... }
  void remObserver( ISubject* subject, IObserver* observer ) { ... }
};
_____________________
File: ClockObserver.ah

#include "ObserverPattern.ah"
#include "ClockTimer.h"

aspect ClockObserver : public ObserverPattern {
  // define the pointcuts
  pointcut subjects()   = "ClockTimer";
  pointcut observers()  = "DigitalClock"||"AnalogClock";

public:
  advice observers() : slice class {
  public:
    void update( ObserverPattern::ISubject* sub ) {
      Draw();
    }
  };
};
```

Fig. 2. Reusable Observer-Pattern Aspect

4.4 Join Point API

The join point API (Figure 1-g) 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++

```
namespace win32 {
  struct Exception {
    Exception( const std::string& w, DWORD c ) { ... }
  };

  // Check for "magic value" indicating an error
  inline bool IsErrorResult( HANDLE 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%(...)"
                       || "% BeginPaint(...)"
                       || "% CreateFile%(...)"
                       || ...
} // 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 Win32 Errors as Exceptions

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 compile-time*. 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] 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.

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 Figure 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 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 (Figure 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 after advice for all calls to functions defined by the `win32::Win32API()` pointcut (Figure 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<I>`. 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 [5] . 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 [6] . 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.

[5] 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/`

6.1 Translation Process

A C++ program consists of a set of self-contained [7] 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 binds 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. On the one hand this design decision facilitates the implementation of wrapper programs, which hide `ac++` from the build environment. On the other hand this significantly restricts the knowledge of the aspect weaver to single translation units. To overcome this limitation the following problems had to be solved.

6.1.1 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

[7] anything that is used either has to be defined or declared.

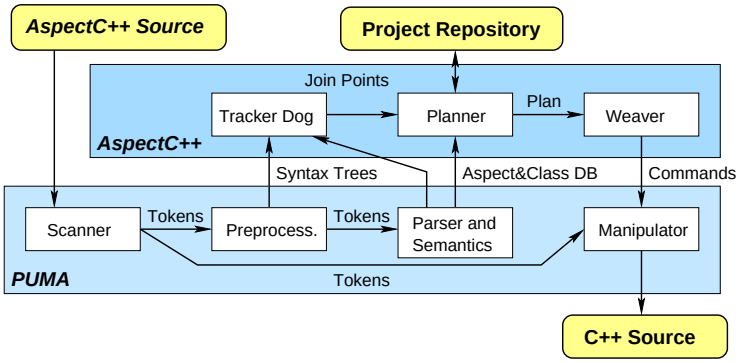

Fig. 4. Architecture of the AspectC++ Weaver

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++` incarnations running later. For example, it would be impossible to provide a project-wide unique ID (Figure1-g) 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

Figure 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 in-

[8] available from `http://acdt.aspectc.org/`

stantiation, 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 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 performing 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.

```
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*)arg (I);   }
};
```

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.

```
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: [9]

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

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 μ-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 [10], while we used the even better optimizing Intel C++ compiler `icc` [11] 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).

[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.

[10] g++ 3.3.5 (-O3 -mpreferred-stack-boundary=2 -fno-align-functions -fno-align-jumps -fno-align-loops -fno-align-labels -fno-reorder-blocks -fno-prefetch-loop-arrays)

[11] `icc` 9.0 (-O3)

Resource consumption with `g++` 3.3.5:

a) incrementer

	advice	cycles abs	cycles Δ	stack abs	stack Δ	code abs	code Δ
	tangled	*4*		*0*		*4128*	
execution	before	*6*	*2*	*0*	*0*	*4128*	*0*
execution	after	*6*	*2*	*0*	*0*	*4128*	*0*
execution	around	*6*	*2*	*0*	*0*	*4128*	*0*
call	before	*6*	*2*	*0*	*0*	*4128*	*0*
call	after	*6*	*2*	*0*	*0*	*4128*	*0*
call	around	*6*	*2*	*0*	*0*	*4128*	*0*

b) multiaspect

	# aspects	cycles abs	cycles Δ	stack abs	stack Δ	code abs	code Δ
execution	1	*6*		*0*		*4080*	
execution	2	*5*	*-1*	*0*	*0*	*4080*	*0*
execution	3	*6*	*1*	*0*	*0*	*4096*	*16*
call	1	*6*		*0*		*4096*	
call	2	*5*	*-1*	*0*	*0*	*4096*	*0*
call	3	*6*	*1*	*0*	*0*	*4096*	*0*

c) parameters, jp-api

	tjp->	cycles abs	cycles Δ	stack abs	stack Δ	code abs	code Δ
int B::bar(...)	*plain*, n=0	*5*		*16*		*3968*	
int B::bar(...)	that()	*7*	*2*	*20*	*4*	*3968*	*0*
int B::bar(...)	target()	*8*	*3*	*20*	*4*	*3968*	*0*
int B::bar(...)	result()	*11*	*6*	*16*	*0*	*3968*	*0*
int B::bar(...)	*plain*, n=1	*13*		*24*		*3968*	
int B::bar(...)	arg<0>()	*13*	*0*	*24*	*0*	*3968*	*0*
int B::bar(...)	*plain*, n=2	*13*		*32*		*3984*	
int B::bar(...)	arg<1>()	*13*	*0*	*32*	*0*	*3984*	*0*

Resource consumption with `icc` 9.0:

a) incrementer

	advice	cycles abs	cycles Δ	stack abs	stack Δ	code abs	code Δ
	tangled	*0*		*0*		*6372*	
execution	before	*0*	*0*	*0*	*0*	*6372*	*0*
execution	after	*0*	*0*	*0*	*0*	*6372*	*0*
execution	around	*0*	*0*	*0*	*0*	*6372*	*0*
call	before	*3*	*3*	*0*	*0*	*6356*	*-16*
call	after	*3*	*3*	*0*	*0*	*6356*	*-16*
call	around	*3*	*3*	*0*	*0*	*6356*	*-16*

b) multiaspect

	# aspects	cycles abs	cycles Δ	stack abs	stack Δ	code abs	code Δ
execution	1	*3*		*4*		*6424*	
execution	2	*2*	*-1*	*4*	*0*	*6340*	*-84*
execution	3	*3*	*0*	*4*	*0*	*6340*	*0*
call	1	*4*		*0*		*6324*	
call	2	*5*	*1*	*0*	*0*	*6340*	*16*
call	3	*9*	*4*	*0*	*0*	*6340*	*0*

c) parameters, jp-api

	tjp->	cycles abs	cycles Δ	stack abs	stack Δ	code abs	code Δ
int B::bar(...)	*plain*, n=0	*9*		*4*		*6372*	
int B::bar(...)	that()	*8*	*-1*	*4*	*0*	*6372*	*0*
int B::bar(...)	target()	*11*	*2*	*4*	*0*	*6372*	*0*
int B::bar(...)	result()	*7*	*-2*	*4*	*0*	*6372*	*0*
int B::bar(...)	*plain*, n=1	*14*		*20*		*6404*	
int B::bar(...)	arg<0>()	*14*	*0*	*20*	*0*	*6404*	*0*
int B::bar(...)	*plain*, n=2	*17*		*28*		*6420*	
int B::bar(...)	arg<1>()	*17*	*0*	*28*	*0*	*6420*	*0*

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.

c) costs of a member function call to int B::bar(*n*) (*n* = 0–2 int parameters) for which some advice is given that either does not use the join-point API (*plain*) or calls a join-point API function (*that()*, ...). Δ denotes the difference to the corresponding *plain* line.

Table 2
AspectC++ μ-benchmark results

The **ground overhead** of applying advice to a parameterless function is very low (Table 2-a). On a Pentium 3 with `g++`, advice invocation takes only 2 cycles, independent from the type of advice (*before/after/around*), the join-point type (*call/execution*), and even the number of aspects giving advice to the join-point (Table 2-b). 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 2-c, which represent the costs of a member function call with 0–2 `int` parameters for that some advice was given (2-c numbers are not directly comparable with those from 2-a and 2-b, 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 to store the pointer returned by `tjp->that()`, for instance, are only consumed if the advice code actually uses `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 `ac++` generates efficient code. In μ-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 capabilites. 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 `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 [12] . 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

[1] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. AW, 2001.

[2] O. S. Bagge, K. T. Kalleberg, M. Haveraaen, and E. Visser. Design of the CodeBoost transformation system for domain-specific optimisation of C++ programs. In D. Binkley and P. Tonella, editors, *Third International Workshop on Source Code Analysis and Manipulation (SCAM 2003)*, pages 65–75, Amsterdam, The Netherlands, Sept. 2003. IEEE.

[3] L. Bergmans. *Composing Concurrent Objects*. PhD thesis, University of Twente, 1994.

[4] S. Chiba. Metaobject Protocol for C++. In *10th ACM Conf. on OOP, Systems, Languages, and Applications (OOPSLA '95)*, pages 285–299, Oct. 1995.

[5] Y. Coady and G. Kiczales. Back to the future: A retroactive study of aspect evolution in operating system code. In M. Akşit, editor, *2nd Int. Conf. on Aspect-Oriented Software Development (AOSD '03)*, pages 50–59, Boston, MA, USA, Mar. 2003. ACM.

[6] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In *ESEC/FSE '01*, 2001.

[7] K. Czarnecki, L. Dominick, and U. W. Eisenecker. Aspektorientierte Programmierung in C++, Teil 1–3. *iX, Magazin für professionelle Informationstechnik*, 8–10, 2001.

[8] K. Czarnecki and U. W. Eisenecker. *Generative Programming. Methods, Tools and Applications*. AW, May 2000.

[9] C. Diggins. Aspect-Oriented Programming & C++. *Dr. Dobb's*, 408(8), Aug. 2004.

[10] B. Dufour, C. Goard, L. Hendren, C. Verbrugge, O. de Moor, and G. Sittampalam. Measuring the dynamic behaviour of AspectJ programs. In *19th ACM Conf. on OOP, Systems, Languages, and Applications (OOPSLA '04)*, pages 150–169, New York, NY, USA, 2004. ACM.

[11] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming. *CACM*, pages 29–32, Oct. 2001.

[12] Actually, `ac++` is already able to weave in C code, but the generated code has always to be compiled with a C++ compiler.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. AW, 1995.

[13] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *17th ACM Conf. on OOP, Systems, Languages, and Applications (OOPSLA '02)*, pages 161–173. ACM, 2002.

[14] W. Harrison and H. Ossher. Subject-oriented programming—a critique of pure objects. In *8th ACM Conf. on OOP, Systems, Languages, and Applications (OOPSLA '93)*, pages 411–428, Sept. 1993.

[15] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting started with AspectJ. *CACM*, pages 59–65, Oct. 2001.

[16] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *15th Eur. Conf. on OOP (ECOOP '01)*, volume 2072 of *LNCS*, pages 327–353. Springer, June 2001.

[17] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *11th Eur. Conf. on OOP (ECOOP '97)*, volume 1241 of *LNCS*, pages 220–242. Springer, June 1997.

[18] K. J. Lieberherr. *Adaptive Object-Oriented Software: the Demeter Method with Propagation Patterns*. PWS, 1996.

[19] D. Lohmann, G. Blaschke, and O. Spinczyk. Generic advice: On the combination of AOP with generative programming in AspectC++. In G. Karsai and E. Visser, editors, *3rd Int. Conf. on Generative Programming and Component Engineering (GPCE '04)*, volume 3286 of *LNCS*, pages 55–74. Springer, Oct. 2004.

[20] D. Lohmann and O. Spinczyk. Architecture-Neutral Operating System Components. *23rd ACM Symp. on OS Principles (SOSP '03)*, Oct. 2003. WiP presentation.

[21] S. Schupp, D. Gregor, D. R. Musser, and S.-M. Liu. Semantic and behavioral library transformations. *Information and Software Technology*, 44(13):797–810, 2002.

[22] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: An aspect-oriented extension to C++. In *40th Int. Conf. on Technology of OO Languages and Systems (TOOLS Pacific '02)*, pages 53–60, Sydney, Australia, Feb. 2002.

[23] O. Spinczyk and D. Lohmann. Using AOP to develop architecture-neutral operating system components. In *11th SIGOPS European W'shop*, pages 188–192, Leuven, Belgium, Sept. 2004. ACM.

[24] A. Tešanović, D. Nyström, J. Hansson, and C. Norström. Aspects and components in real-time system development: Towards reconfigurable and reusable software. *Embedded Computing*, Feb. 2004.

[25] A. Tešanović, K. Sheng, and J. Hansson. Application-tailored database systems: a case of aspects in an embedded database. In *8th Int. Database Engineering and Applications Symp. (IDEAS '04)*, Coimbra, Portugal, July 2004. IEEE.