Generic Advice: On the Combination of AOP with Generative Programming in AspectC++

Daniel Lohmann, Georg Blaschke, and Olaf Spinczyk

Friedrich-Alexander-University Erlangen-Nuremberg, Germany
{dl,gb,os}@cs.fau.de

Abstract. Besides object-orientation, generic types or templates and aspect-oriented programming (AOP) gain increasing popularity as they provide additional dimensions of decomposition. Most modern programming languages like Ada, Eiffel, and C++ already have built-in support for templates. For Java and C# similar extensions will be available in the near future. Even though promising, the combination of aspects with generic and generative programming is still a widely unexplored field. This paper presents our extensions to the AspectC++ language, an aspect-oriented C++ derivate. By these extensions aspects can now affect generic code and exploit the potentials of generic code and template metaprogramming in their implementations. This allows aspects to inject template metaprograms transparently into the component code. A case study demonstrates that this feature enables the development of highly expressive and efficient generic aspect implementations in AspectC++. A discussion whether these concepts are applicable in the context of other aspect-oriented language extensions like AspectJ rounds up our contribution.

1 Introduction

Besides object-orientation, most modern programming languages offer *generic types* or *templates* to provide an additional dimension of decomposition. Languages like Ada, Eiffel and C++ already have built-in support for templates. For the Java language, special extensions for generic types have been proposed [6] and will soon be available in Java 1.5 implementations [4]. Template support has also been proposed for the next version of the C# language and the .NET framework [12]. We are convinced that for a long-term and broad acceptance of AOP it is necessary to provide native support for template constructs in current AOP languages.

As AspectC++ [17] supports AOP in the context of C++, which offers highly sophisticated template features, it is an ideal example to study the relation of these two worlds. To familiarize the reader with the terminology used throughout this paper we start with a very brief introduction of the most important AspectC++ language concepts.

1.1 AspectC++ Concepts and Terminology

AspectC++ is a general purpose aspect-oriented language extension to C++ designed by the authors and others. It is aimed to support the well-known AspectJ programming style in areas with stronger demands on runtime efficiency and code density.

G. Karsai and E. Visser (Eds.): GPCE 2004, LNCS 3286, pp. 55–74, 2004.

The AspectC++ terminology is strongly influenced by the terminology introduced by AspectJ [13]. The most relevant terms are *joinpoint* and *advice*. A *joinpoint* denotes a specific weaving position in the target code (often called component code, too). Joinpoints are usually given in a declarative way by a joinpoint description language. Each set of joinpoints, which is described in this language, is called a *pointcut*. In AspectC++ the sentences of the joinpoint description language are called *pointcut expressions*. For example the pointcut expression

```
call("% Service::%(...)")
```

describes all calls to member functions of the class `Service`[1]. The aspect code that is actually woven into the target code at the joinpoints is called *advice*. Advice is bound to a set of joinpoints (given by a pointcut expression). For example by defining the advice

```
advice call("% Service::%(...)") : before () {
  cout << "Service function invocation" << endl;
}
```

the program will print a message *before* any call to a member function of `Service`. The advice code itself has access to its context, i.e. the joinpoint which it affects, at runtime by a *joinpoint API*. Very similar to the predefined `this`-pointer in C++, AspectC++ provides a pointer called `tjp`, which provides the context information. For example the advice

```
advice call("% Service::%(...)") : before () {
  cout << tjp->signature() << " invocation" << endl;
}
```

prints a message that contains the name of the function that is going to be called.

1.2 Dimensions of Interest

The combination of aspects and templates has two general dimensions of interest. First, there is the dimension of using aspects for the instrumentation of generic code. Second, there is the dimension of using generic code in aspects, that is, to generalize aspect implementations by utilizing generic code. We will see that for the first it is necessary to extend the joinpoint description language while the second leads to specific joinpoint API requirements.

Dimension 1: Using Aspects for the Instrumentation of Generic Code. The most common and popular applications of templates are generic algorithms and container classes, as provided by the C++ standard library STL. To support the modular implementation of crosscutting concerns inside template libraries, a template-aware AOP language should be able to weave aspect code into template classes and functions. This means, for example, that we want to be able to run advice code before and/or after calls to or executions of generic classes or functions. Furthermore, it would be desirable to be able to affect even individual template *instances*. For example, we might want to track element additions to a specific instance of a generic container. Open questions are:

[1] % and ... are wildcard symbols in AspectC++ (as * and .. in AspectJ).

- Which extensions to the joinpoint languages are necessary to describe these new kinds of joinpoints?
- Which implementation issues for aspect weavers and compilers arise from such extensions?

As today's aspect-oriented programming languages offer a wide range of language features, finding a common (language-independent) answer for these questions is difficult. However, this paper will answer them in the context of C++/AspectC++ and discuss them in the context of other aspect languages that support similar language features.

Dimension 2: Using Generic Code in Aspects. Generic programming provides an additional dimension for separation of concerns (SoC). Thus, we also want to *use template code* in aspect implementations. This seems easy at first sight. However, an advanced utilization of generic code in advice implementations incorporates (static) type information from the advice context. For example, consider a reusable caching aspect, implemented by advice code that uses a generic container to store argument/result pairs of function invocations. To instantiate the container template from within the advice code, the affected function *argument types* and *result type* are needed, which in turn depend on the joinpoint the advice code affects. As in this case different advice code versions have to be generated, it is reasonable to understand the advice as being *instantiated* for each joinpoint depending on the types it needs for code generation. We will use a new term for this special kind of advice:

We call advice a *generic advice,* if its implementation depends on joinpoint-specific static type information.

As templates are instantiated at compile time, the type information, which is offered by the run-time joinpoint API of current aspect languages, is not sufficient. Instead, we need a *compile-time* joinpoint API that provides static type information (e.g. the parameter types of the affected function) about the advice instantiation context. The type information provided by the compile-time joinpoint API may then be used to instantiate templates. Open questions concerning dimension 2 are:

- How should aspect languages support the usage of generic code in aspects?
- Which information should be provided by a static joinpoint API?

Besides generic programming, C++ templates are also the base for other advanced and powerful SoC concepts like policy-based design[2] and generative programming[9] that are implemented by template metaprogramming[18]. The ability to use template constructs in advice code promises to forge the link between AOP and these advanced SoC concepts. Aspects may provide great benefit for generative programming and policy-based design by enabling the non-invasive intrusion of template metaprograms. At the same time, generative techniques may be used to implement highly flexible and runtime-efficient aspect implementations.

1.3 Objectives

In this paper we discuss the combination of aspects and templates in AspectC++. From the discussion in this introduction we conclude that weaving in template instances is a consequent and probably useful extension of the target scope of aspects. However, providing advice code with the necessary context information to exploit the power of generic code and metaprograms improves the expressive power of aspect implementations significantly. This promises to be the more interesting research topic[2]. Thus, our work covers both dimensions but concentrates on the second. By presenting a case study we will demonstrate the power of the generic advice feature and the consequences in terms of code quality and performance.

1.4 Outline

The rest is this paper is structured as follows. It starts in section 2 with a description of the template support incorporated into the AspectC++ language and compiler. This is followed by a case study in section 3 that shows how generative techniques can be used for the implementation of a highly flexible and efficient aspect. Furthermore, it evaluates the performance and quality of the solution, which we claim to be unreachable alone with pure template techniques or classic AOP. Section 4 discusses whether similar aspect-oriented language extensions for base languages other than C++ could implement the same concepts. Finally, an overview of related work is given and the results of this paper are summarized.

2 Template Support in AspectC++

This section describes our integration of template support into AspectC++. It is structured according to the two dimensions[3] identified in the introduction. Each of these parts starts with a description of template-related C++ features, which are relevant for the integration, continues with a presentation of our extensions to AspectC++ on the language-level, and ends with a discussion of implementation issues.

2.1 Dimension 1: Using Aspects for the Instrumentation of Generic Code

One possible combination of aspects with templates is the application of advice to template classes and functions as discussed in 1.2.

Relevant Language Features. C++ templates provide a specific kind of polymorphism which is often referred to as *parametric polymorphism*. While *subclass polymorphism* (based on interface inheritance) defines a runtime substitutability between *object instances*, parametric polymorphism defines a compile-time substitutability between *types*. A formal template parameter can be substituted by a type T1 that provides

[2] It is also more appropriate for a conference on generative programming.

[3] In fact, the part on the second dimension is divided into 2a and 2b to handle the very C++-specific template metaprogramming feature separately.

a specific (static) interface. This interface is defined by the set of operations, types and constants used in the template implementation. Note that, in contrast to subclass polymorphism, the parametric substitutability of types T1 and T2 does not lead to any runtime relationship between object instances of T1 and T2. Templates do not break type-safety and induce, in principle, no runtime overhead at all.

In C++, the basic language elements for generic programming are *template classes* and *template functions*. For example, the C++ standard library class

```
template< class T, class U > struct pair {
  T first;
  U second;
  pair() : first(), second() {}
  pair(const T& x, const U& y) : first(x), second(y) {}
  ...
};
```

defines a simple 2-Tuple that can be instantiated with any two types, e.g `pair< int, int >` or `pair< Foo, pair< Bar, int > >` that provide at least a default or copy constructor.

Extensions to the Joinpoint Language. Weaving in generic code requires the programmers to describe the set of relevant template instances. Thus, the first step to integrate template support into AspectC++ was to extend the joinpoint description language.

In AspectC++ joinpoints that should be affected by advice are described by so-called *pointcut expressions*. For example

```
call("% Service::%(...)") && cflow(execution("void error_%(...)"))
```

describes all calls to member functions of the class `Service` . By combining (`&&` operation) these joinpoints with the `cflow` pointcut function these joinpoints become conditional. They are only affected by advice if the flow of control already passed a function with a name beginning with `error_`. Users may define pointcut expressions of arbitrary complexity to describe the crosscutting nature of their aspects. A list of all built-in pointcut functions of AspectC++ is available in the AspectC++ Language Quick Reference Sheet[4].

The core of the joinpoint language are match-expressions. In AspectC++ these expressions are quoted strings where `%` and ... can be used as wildcards. They can be understood as regular expressions matched against the names of known program entities like functions or classes. To support advice for template class and function instances, the signatures of these instances just have to be considered when match expressions are evaluated. In C++ the signatures of template instances are well-defined and can directly be used to extend the set of known program entities for matching. For example, if a function instantiates template classes with the signatures `set<int>` and `set<float>`, the match-expression `"% set<int>::%(...)"` will only match the member functions of `set<int>`. Of course, it is also possible to match the member functions of all instances by using `"% set<...>::%(...)"`.

[4] The AspectC++ compiler and documentation are available from `www.aspectc.org`.

Implementation Issues. While this extension is straightforward on the language level it has severe consequences on the weaver implementation. The weaver has to be able to distinguish template instances during the weaving process. Our AspectC++ compiler transforms AspectC++ source code into C++ code[5]. Thus, the compiler has to perform a full template instantiation analysis of the given source code to distinguish template instances and to compare their signatures with match-expressions. To be able to affect only certain instances our compiler uses the explicit template specialization feature of C++. For example, if advice affects only the instance `set<int>` the template code of `set` is copied, manipulated according to the advice, and declared as a specialization of `set` for `int`[6].

2.2 Dimension 2a: Using Generic Code in Aspects

Templates instances, like `pair<int, int>` in the example above, can also be used in aspect code. However, for a more elaborate and context dependent utilization of templates in advice code, additional support by the aspect language is necessary.

Relevant Language Features. *Generic advice* uses type information from its instantiation context (the joinpoints). For example, it may instantiate a generic container with the result type of the function the advice is applied to. For this, the argument and result types have to be accessible to the advice code at compile time.

The suitable C++ language concept for this purpose is *type aliases*. In C++, a type alias is defined using the `typedef` keyword. Type aliases can be used to transport type information through other types or templates:

```
template< class T, class U > struct pair {
  typedef T first_type;
  typedef U second_type;
  ...
};
template<class PAIR> typename PAIR::first_type& get_first(PAIR& p) {
  return p.first;
}
```

The Static Joinpoint API. To support the implementation of generic advice code the AspectC++ joinpoint API had to be extended. It now provides static type information about all types that are relevant for the joinpoint.

Table 1 gives an overview about the new joinpoint API. The upper part (types and enumerators) provides compile-time type information, which can be used by generic code or metaprograms instantiated by advice.

The methods in the lower part of table 1 can only be called at runtime. However, note that the new function `Arg<i>::ReferredType *arg()` now offers a statically typed

[5] AspectC++ is based on our PUMA C++ transformation framework which is beeing developed in line with AspectC++, but may be used for other purposes as well.

[6] C++ does not support the explicit specialization for template functions. However, we can work around this problem by defining a helper template class.

Table 1. The AspectC++ Joinpoint API

types and enumerators:	
`That`	object type
`Target`	target type (call)
`Arg<i>::Type`	argument type
`ARGS`	number of arguments
`Result`	result type
static methods:	
`const char *signature()`	signature of the function or attribute
`unsigned id()`	identification of the join point
`AC::JPType jptype()`	type of join point
`AC::Type type()`	type of the function or attribute
`AC::Type argtype(int)`	types of the arguments
`int args()`	number of arguments
`AC::Type resulttype()`	result type
non-static methods:	
`void proceed()`	execute join point code
`AC::Action &action()`	Action structure
`That *that()`	object referred to by this
`Target *target()`	target object of a call
`void *arg(int)`	actual argument
`Arg<i>::ReferredType *arg()`	argument with static index
`Result *result()`	result value

interface to access argument values if the argument index is known at compile time. `AC::Type` is a special data type that provides type information at runtime, but cannot be used to instantiate templates.

In the future, additional type information, e.g. types of class members and template arguments, will be added to the joinpoint API, as this might allow further useful applications.

Implementation Issues. AspectC++ generates a C++ class with a unique name for each joinpoint that is affected by advice code. Advice code is transformed into a template member function of the aspect, which in turn is transformed to a class. The unique joinpoint class is passed as a template argument to the advice code. Thus, the advice code is generic and can access all type definitions (C++ typedefs) inside the joinpoint class with `JoinPoint::`*`Typename`*. Indirectly these types can also be used by using the type-safe argument and result accessor function. The following code fragment shows advice code after its transformation into a template function.

```
class ServiceMonitor {
  // ...
  template<class JoinPoint> void __a0_after (JoinPoint *tjp) {
    cout << "Result: " << *tjp->result () << endl;
  }
};
```

As required for generic advice the C++ compiler will instantiate the advice code with `JoinPoint` as a code generation parameter. Depending on the result type of `tjp->result()` for the joinpoint the right output operator will be selected at compile time.

2.3 Dimension 2b: Using Template Metaprograms in Aspects

The C++ template language provides elaborate features that make it a Turing-complete functional language for static metaprogramming on its own. A C++ template metaprogram works on types and constants and is executed by the compiler at compile time. *Generative Advice* is a special form of generic advice that uses joinpoint-specific type information for the instantiation of template metaprograms.

Relevant Language Features. In addition to types, C++ supports *non-type template parameters*, also frequently referred to as value template parameters. Value template parameters allow to instantiate a template according to a compile-time constant. Apart from the definition of compile-time constants, value template parameters are frequently used for arithmetics and looping in template metaprogramming.

A language that provides a *case discrimination* and a *loop* construct is Touring-complete. In the C++ template metalanguage, case discrimination is realized by *template specialization*. Loops are implemented by *recursive instantiation* of templates. These language features, in conjunction with non-type template parameters, are the building blocks of template metaprograms[7]. This is shown in the most popular (and simple) example of a template metaprogram that calculates the factorial of an integer at compile-time:

```
template< int N > struct fac {
  enum{ res = N * fac< N - 1 >::res }; // loop by rec. instantiation
};
// condition to terminate recursion by specialization for case 0
template<> struct fac< 0 >{
  enum{ res = 1 };
};
// using fac
const fac_5 = fac< 5 >::res;
```

The `fac<int>` template depends on value template parameters and calculates the factorial by recursive instantiation of itself. The recursion is terminated by a specialization for the case `fac<0>`.

Type Sequences. With adequate support by the static joinpoint API metaprograms would be able to iterate over type sequences like the argument types of a joinpoint at compile time. However, these sequences have to be provided in a "metaprogram-friendly" way. Just generating `ArgType0`, `ArgType1`, ..., `ArgTypeN` would not allow

[7] Non-type template parameters are, strictly speaking, not a mandatory feature for the Turing-completeness of the C++ template language. They "just" allow one to use the built-in C++ operator set for arithmetic expressions.

metaprograms to iterate over these types. For this purpose the generated joinpoint-specific classes contain a template class `Arg<I>` which contains a the type information for the I^{th} argument as typedefs.

Implementation Issues. Sequences of types can be implemented by recursive template definitions as in the Loki[1] `Typelist`. For the AspectC++ we decided for an implementation with less demands on the back-end compiler based on explicit template specialization. The following code shows the generated type for a call joinpoint in the `main()` function[8].

```
struct TJP_main_0 {
  typedef float Result;
  typedef void That;
  typedef ::Service Target;
  enum { ARGS = 2 };
  template <int I> struct Arg {};
  template <> struct Arg<0> {
    typedef bool Type; typedef bool ReferredType;
  };
  template <> struct Arg<1> {
    typedef int & Type; typedef int ReferredType;
  };
  Result *_result;
  inline Result *result() {return _result;}
};
```

With this type as a template argument for generic advice the advice programmer can now use a metaprogram similar to the factorial calculation to handle each argument in a type-safe way. A further example will be given in the next section.

3 Caching as a Generative Aspect – A Case Study

In this section we demonstrate how the possibility to exploit generic/generative programming techniques for aspect implementations can lead to a very high level of abstraction. Our example is a simple caching policy for function invocations. The implementation idea is straightforward: before executing the function body, a cache is searched for the passed argument values. If an entry is found, the corresponding result is returned immediately, otherwise the function body is executed and the cache is updated.

Caching is a typical example for a crosscutting policy. However, before we extended AspectC++ it was not possible to implement this in a traditional aspect-oriented way. To be generic, the aspect must be able to copy function arguments of arbitrary types into the cache. As C++ lacks good reflection capabilities, it is not possible to do this at runtime.

A caching policy for function invocations is also not easy to implement using sophisticated template techniques like policy-based design[2]. To achieve this, it would

[8] The generated code depends on the compiler used as the `ac++` back-end.

be necessary to have a generic way to expose the signature (parameter types) of a function at compile time and the invocation context (actual parameter values) at runtime, which is not available. Fortunately, exactly this information is provided by the compile-time and run-time joinpoint API in AspectC++. Hence, it is now possible to implement context-dependent policies, like caching, using generative techniques *together* with aspects.

In the following, we present the implementation of such a generative caching aspect. We introduce the example with an in-place "handcrafted" cache implementation for a single function. A typical AOP implementation of this simple in-place cache gives readers, who are not familiar with AOP, an idea about implementing such a policy by an aspect. We then generalize the AOP implementation to make it independent on the number and types of function arguments using template metaprogramming and the static joinpoint API. This generative aspect is applicable to any function with any number of arguments. Finally, we evaluate our solution and provide a detailed performance analysis.

3.1 Scenario

Consider an application that uses the class `Calc` (Figure 1). Our goal is to improve the overall application execution speed. With the help of a tracing aspect we figured out that the application spends most time in the computational intensive functions `Calc::Expensive()`, `Calc::AlsoExpensive()` and `Calc::VeryExpensive()`. Furthermore, we detected that these functions are often called several times in order with exactly the same arguments. Therefore, we decided to improve the execution speed by using a simple one-element cache.

Figure 1 demonstrates the implementation principle of such a cache for the function `Calc::Expensive()`. It uses a local class `Cache` that implements the caching functionality and offers two methods, namely `Cache::Lookup()` (line 28) and `Cache::Update()` (line 32), to access and write the cached data. `Cache` is instantiated as a static object (line 41) so it stays resident between function invocations.

However, as caching is useful only in certain application scenarios and we have to write almost the same code for `Calc::AlsoExpensive()` and `Calc::VeryExpensive()` again, we want to implement this by a reusable aspect.

3.2 Development of a Caching Aspect

The Simple Caching Aspect. Figure 2 shows a typical AOP "translation" of the caching strategy. The class `Cache` has become an inner class of the aspect `Caching`. The aspect gives around advice for the execution of `Calc::Expensive()`, `Cache` is now instantiated in the advice body (line 22). On each invocation, the function parameters are first looked up in the cache and, if found, the cached result is returned (lines 25–27). Otherwise they are calculated by invoking the original function and stored together with the result in the cache (lines 28–34).

This implementation of a caching aspect is quite straightforward. It uses methods of the runtime joinpoint API (`tjp->arg()` and `tjp->result()`) to access the actual parameter and result values of a function invocation. However, it has the major drawback that

```
struct Vector
{
  Vector( double _x = 0, double _y = 0 )
    : x( _x ), y( _y ) {}
  Vector(const Vector& src){operator =(src);}
  Vector& operator =( const Vector& src ) {
    x = src.x; y = src.y; return *this;
  }
  bool operator==(const Vector& with) const{
    return with.x == x && with.y == y;
  }
  double x, y;
};

class Calc {
public:
  Vector Expensive(const Vector& a,
                   const Vector& b) const;
  int AlsoExpensive(double a,double b) const;
  int VeryExpensive(int a,int b,int c) const;
};


Vector Calc::Expensive(const Vector& a,
                       const Vector& b)const{
  struct Cache {
    Cache() : valid( false ) {}
    bool Lookup( const Vector& _a,
                 const Vector& _b ) {
      return valid && _a == a && _b == b;
    }
    void Update( const Vector& _res,
                 const Vector& _a,
                 const Vector& _b ) {
      valid = true;res = _res; a = _a; b = _b;
    }
    Vector a, b, res;
    bool   valid;
  };

  static Cache cc;

  // Lookup value in cache
  if( cc.Lookup( a, b ) ) {
    return cc.res;
  } else {  // Not in cache
    Vector Result;
    // ... do calculations ...

    // Store result in cache
    cc.Update( Result, a, b ) ;
    return cc.res;
  }
}
```

Fig. 1. Elements of the Example Application

it can be applied only to functions with a specific signature, namely functions with two arguments of type `Vector` which return a `Vector`. This limited reusability of the caching aspect comes from the fact that both the class `Cache` and the advice code implementation are built on (implicit) assumptions about the argument and result types.

Generalization. The simple cache class implementation in Figure 2 makes the following implicit assumptions about the advice context it is applied to:

1. The function's result and arguments are of type *Vector*.
2. The function gets *exactly two arguments*. Besides the return value, exactly two values have to be updated by `Cache::Update()` and compared by `Cache::Lookup()`.

Generalizing from 1 (the result/argument types) can be achieved by passing the types as template parameters. However, to generalize from 2 (the number of arguments) is a bit more complicated. Before weaving time, it is not possible to decide how many parameters have to be stored, updated and compared by `Cache`. The memory layout of the class `Cache`, as well as the code of its member functions `Lookup()` and `Update()`, has therefore to be *generated* at compilation time. With AspectC++, it is now possible to do this job by a template metaprogram. The new static joinpoint API provides all the necessary context information (number/types of parameters, result type) in a way that is accessible by the C++ template language.

To simplify the implementation of the generative caching aspect (Figure 3), it is mainly based on existing libraries and well-known idioms for template metaprogramming. We used the Loki library[2] by Alexandrescu, especially `Loki::Typelist` and

```
1  aspect Caching {
2    struct Cache {
3      Cache() : valid( false ) {}
4      bool Lookup( const Vector& _a,
5           const Vector& _b ) const {
6        return valid && _a == a && _b == b;
7      }
8      void Update( const Vector& _res,
9                   const Vector& _a,
10                  const Vector& _b ) {
11       valid = true;
12       res = _res;
13       a = _a; b = _b;
14     }
15     Vector a, b, res;
16     bool   valid;
17   };
18
19   advice execution(
20       "Vector Calc::%( const Vector& %,"
21       "const Vector& % )" ) : around() {
22     static Cache cc;
23
24     // Lookup value in cache
25     if( cc.Lookup( *(Vector*)tjp->arg(0),
26         *(Vector*)tjp->arg(1) ) )
27       *tjp->result() = cc.res;
28     else {
29       // Not in cache. Calculate and store
30       tjp->proceed();
31       cc.Update( *(Vector*)tjp->result(),
32         *(Vector*)tjp->arg(0),
33                 *(Vector*)tjp->arg(1));
34     }
35   }
36 };
```

Fig. 2. Simple Caching Aspect

`Loki::Tuple`[9]. The `Loki::Tuple` construct creates (at compile time) a tuple from a list of types, passed as a `Loki::Typelist`. The resulting tuple is a class that contains one data member for each element in the type-list. The rough implementation idea is, to pass a list of parameter types to the `Cache` template and to store the cache data in a `Loki::Tuple` (instead of distinct data members). The `Lookup()` and `Update()` methods are created at compile time by special generators, because the number of arguments (and therefore the number of necessary comparisons and assignments) is unknown until the point of template instantiation.

In the implementation (Figure 3), the `Cache` class gets only one template parameter (line 32), `TJP`, which is used to pass the actual joinpoint type. The joinpoint type, `JoinPoint`, is created by the AspectC++ compiler and implicitly known in every advice body. It provides the necessary context information. The advice code passes `JoinPoint` as a template argument to `Cache` in the instantiation of the cache object (line 81). Everything else needed for the implementation is retrieved from types and constants defined in `TJP`. The `Cache` class itself is derived from a `Loki::Tuple`, built from the argument types `TJP::Arg<0>::Type`, `TJP::Arg<1>::Type`, ... (line 34). However, the argument types first have to be transformed into a `Loki::Typelist`. This task is performed by the `JP2LokiTL` metaprogram (lines 13-28), which also removes possible const and reference attributes from the types with a `Traits` helper class. By deriving `Cache` from a tuple constructed this way, it already contains one data member of the correct type for each function argument. The remaining data members (`res` and `valid`) are defined explicitly (lines 64, 65).

The code for the methods `Cache::Lookup()` and `Cache::Update()` is generated by metaprograms. The `Comp_N` metaprogram loops over the function arguments and generates a conjunction of comparisons. Each iteration adds one comparison between the

[9] Loki was chosen not only because it provides high level services for our implementation. As a "heavily-templated" library it was also considered being a good real-world test case for the capabilities of our AspectC++ parser.

```
1  #include "loki/TypeTraits.h"
2  #include "loki/Typelist.h"
3  #include "loki/HierarchyGenerators.h"
4
5  template< class T > struct Traits {
6    typedef typename Loki::TypeTraits<typename
7      Loki::TypeTraits< T >::ReferredType
8    >::NonConstType BaseType;
9  };
10
11 namespace AC {
12   // Builds a Loki::Typelist from arg types
13   template< class TJP, int J >
14   struct JP2LokiTL {
15     typedef Loki::Typelist< typename Traits<
16       typename TJP::Arg<
17         TJP::ARGS-J >::Type >::BaseType,
18       typename JP2LokiTL< TJP, J-1 >::Result
19     > Result;
20   };
21   template< class TJP >
22   struct JP2LokiTL< TJP, 1 > {
23     typedef Loki::Typelist< typename Traits<
24       typename TJP::Arg<
25         TJP::ARGS-1 >::Type >::BaseType,
26       Loki::NullType
27     > Result;
28   };
29 }
30
31 aspect Caching {
32   template<class TJP> struct Cache
33     : public Loki::Tuple< typename
34     AC::JP2LokiTL< TJP, TJP::ARGS >::Result >
35   {
36     // Comp. TJP args with a Loki tuple
37     template<class C,int I> struct Comp_N {
38       static bool proc(TJP* tjp,const C& cc){
39         return *tjp->arg< I >() ==
40           Loki::Field< I >( cc ) &&
41           Comp_N<C,I-1>::proc(tjp,cc);
42       }
43     };
44     template<class C> struct Comp_N<C, 0> {
45       static bool proc(TJP* tjp,const C& cc){
46         return *tjp->arg<0>()
47             == Loki::Field<0>(cc);
48       }
49     };
50
51     // Copies TJP args into a Loki tuple
52     template<class C, int I > struct Copy_N {
53       static void proc( TJP* tjp, C& cc ) {
54         Loki::Field<I>(cc) = *tjp->arg< I >();
55         Copy_N< C, I-1 >::proc( tjp, cc );
56       }
57     };
58     template< class C > struct Copy_N< C, 0 > {
59       static void proc( TJP* tjp, C& cc ) {
60         Loki::Field<0>(cc) = *tjp->arg< 0 >();
61       }
62     };
63
64     bool  valid;
65     typename TJP::Result res;
66     Cache() : valid( false ) {}
67
68     bool Lookup( TJP* tjp ) {
69       return valid && Comp_N< Cache,
70           TJP::ARGS - 1 >::proc(tjp, *this);
71     }
72     void Update( TJP* tjp ) {
73       valid = true;
74       res   = *tjp->result();
75       Copy_N< Cache,
76         TJP::ARGS - 1 >::proc(tjp, *this);
77     }
78   };
79
80   advice execution("% Calc::%(...)") : around(){
81     static Cache< JoinPoint > cc;
82
83     if( cc.Lookup( tjp ) )
84       *tjp->result() = cc.res;
85     else {
86       tjp->proceed();
87       cc.Update( tjp );
88     }
89   }
90 };
```

Fig. 3. Generative Caching Aspect

I[th]argument, retrieved with `tjp->arg<I>()`, and the cached value stored in the I[th] tuple element, retrieved with the `Loki::Field<I>()` helper function. The `Copy_N` metaprogram copies in a similar way the argument values into the cache by generating a sequence of assignments. The implementations of `Cache::Lookup()` and `Cache::Update()` just instantiate these metaprograms and call the generated code (lines 68–71, 72–77).

3.3 Evaluation

The final cache implementation (Figure 3) has indeed become a generic and broadly reusable implementation of a caching strategy. It can be applied non-invasively to functions with 1 to n arguments; each argument being of any type that is comparable and assignable. Type-safety is achieved and code redundancy is completely avoided. The

source code complexity, on the other hand, is notably higher. The encapsulation as an aspect may also result in a performance overhead, as the aspect weaver has to create special code to provide the infrastructure for the woven-in advice. In the following, we discuss these specific pros and cons of our approach.

Performance Analysis. Table 2 shows the results of our performance analysis. It displays, from left to right, the runtime overhead induced by a native in-place cache implementation (as in Figure 1), the simple caching aspect (Figure 2), and the generative caching aspect (Figure 3), if applied (as in the example) to the function `Calc::Expensive()`. The numbers are clock cycles on a Pentium III[10]. For the in-place cache, *cache hit* represent the costs of a successful call to `Lookup()`, *cache miss* represents the costs of an unsuccessful call to `Lookup()`, followed by a call to `Update()`. For the aspect implementations, these numbers include the additional overhead introduced by AspectC++. The numbers in square brackets denote the difference between each implementation and its in-place counterpart. They can be understood as the costs of separating out caching as an aspect.

Table 2. Overhead of the Cache Implementations (clock cycles on a Pentium III)

C++	in-place		simple aspect		generative aspect	
cache hit	38.0	[0.0]	55.0	[17.0]	53.0	[15.0]
cache miss	36.0	[0.0]	70.2	[34.2]	67.0	[31.0]

As expected, the in-place implementation performs best[11]. The additional overhead of the aspect implementations is mainly induced by the fact that AspectC++ needs to create an array of references to all function parameters to provide a unified access to them[12]. If `tjp->proceed()` is called from within around advice, a function call (to the original code) has to be performed[13]. In the example, this is the case in a cache-miss situation and thereby explains why the overhead for a cache miss (31/34 cycles) is notably higher as the overhead for a cache hit (15/17 cycles).

[10] The code was compiled with Microsoft Visual C++ 2003 using /O2 /Ot optimizations. All benchmarks measurements were done on a 700 MHz PIII E ("Coppermine") machine running Windows XP SP1. To reduce multitasking and caching effects, each test was performed 25 times and measured the execution cycles for 1000 repetitions (using the `rdtsc` instruction). The presented results are averaged from this 25 series and then divided by 1000. The standard derivation is < 0.1 for all series.

[11] The effect that the overhead of a cache miss is even lower than for a cache hit can be explained by the four necessary (and relatively expensive) floating point comparisons to ensure a cache hit, while a cache miss can ideally be detected after the first comparison. The skipped comparisons seem to out-weigh the costs of `Update()`.

[12] As an optimization, AspectC++ creates this array only if it is actually used in the advice body, which is the case here.

[13] For technical reasons, AspectC++ does this indirectly via a function pointer in a so-called action object, the call can therefore not be inlined. As the detour over an action object is not necessary in many cases, this gives room for future improvements of the code generator.

Although we expected the generative aspect implementation to perform not worse than the simple aspect, we were a bit surprised that it actually performs better (2-3 cycles, around 10-12%). We assume that the structure of the code generated by the template-metaprograms accommodates the compiler's optimization strategies.

The additional 40-86% relative overhead for the generative aspect compared to the in-place implementation seems to be remarkably high. However, this is basically caused by the very low costs of our most simple cache implementation, which performs a successful lookup in only 38 cycles. A more elaborated cache implementation (e.g. one that stores more than one cache line) would consume more cycles and thereby lower the effective relative overhead of the aspect-oriented solutions. And finally, the absolute overhead of the generative caching aspect is still very low. If we assume a cache hit rate of only 30%, applying this aspect would even pay off for a function that performs its calculation in 270 cycles (0.39μs on a PIII-700)!

Discussion. The caching example has shown, how the combination of AOP with template-related techniques can lead to highly generic and efficient policy implementations. As mentioned above, it is hardly possible to reach the same level of genericity in C++ using templates or aspects alone. Both techniques seem to have their very specific strong points that complement each other.

One strong point of aspects is the *instantiation context* which is available through the joinpoint API. By providing access to the parameter and result types of a function at compile-time, as well as to their actual values at runtime, it offers information that is otherwise not available in a generic way. Another point of aspects is their *non-invasive applicability* by a declarative joinpoint language. While a class or function has to be explicitly prepared to be configurable by templates, aspects can be applied easily and in a very flexible manner "on demand". For instance, by using call advice (instead of execution advice) it is possible to do the caching on the caller side. Hence, the same policy can be applied to selected clients only, or to invocations of third-party code that is not available as source.

However, the easy applicability of aspects leads to high requirements on their genericity and reusability. This forges the link to templates and template metaprogramming. Besides the well known possibility to combine a high level of expressiveness with excellent performance [19], carefully developed generic C++ code is typically instantiable in many different contexts. By using generators, it is even possible to *adapt the structure and the code* depending on the instantiation context.

To summarize, aspects are excellent in defining *where* and under *which circumstances* (context) something should happen, while templates provide the necessary flexibility to define in a generic way *what* should happen at these points. The combination works specifically well for policies which depend on the calling context of a function and/or should be flexibly and non-invasively applicable to clients in configurable program families. Other examples where this technique could be applied are constraint checking, program introspection or marshaling.

While the performance analysis made evident that the resulting flexibility is not payed by a remarkable runtime overhead, it still leads to higher source code complexity. This is a well-known problem of template metaprogramming. However, the develop-

ment of broadly reusable policies and aspects is anyway a task for experienced library developers. The application developer, who uses the aspect library, is not confronted with the template code.

A Generic Caching Aspect in Java. As an addition to our case study, we also implemented a generic caching aspect in AspectJ. Our goal was to understand, if and how a similar genericity is reachable in a language that does not support static metaprogramming and follows a more runtime-based philosophy.

The Java implementation uses reflection to store the argument and result values in the cache by dynamically creating instances of the corresponding types and invoking their copy-constructors. However, while in C++ the copy-constructor is part of the canonical class interface (and its absence is a strong indicator for a non-copyable and therefore non-cachable class), in Java for many classes copy-constructors are "just not implemented". This limits the applicability of the aspect, and, even worse, it is not possible to detect this before runtime! Another strong issue of a reflection-based solution is performance. On an 1GHz Athlon machine the additional costs of a cache-hit are 0.44μs, a cache miss costs, because of the expensive reflection-based copying, about 26.47μs. Assuming again a cache hit rate of 30%, the Java cache would not pay off for functions that consume less than 63μs [14]. This "pay-off-number" is significantly higher than the corresponding 0.39μs (270 ticks on a PIII-700) of the C++ solution[15].

To summarize, the reflection-based "generic" Java solution offers only a limited practical applicability and may even lead to unexpected runtime errors.

4 Applicability to Other Languages

Besides C++, other (imperative) languages like Ada or Eiffel support the template concept. For Java and C# generic extensions have been announced or are on the way into the official language standards[12, 4]. Java, furthermore, provides with AspectJ the most-popular aspect weaver. Several active research projects work on incorporating aspects into the C# language [15, 14] and there are chances that the increasing popularity of AOP also leads to the development of aspect weavers for Ada and Eiffel. This gives rise to the question what kind of combination of AOP with generic programming is feasible in these languages. In this section, we discuss if and how our insights on combining AOP with generic and generative programming in C++ are applicable to other languages.

4.1 Relevant Language Features

In the introduction of this paper, we subdivided the possible combinations of aspects with templates into two dimensions of interest, namely *using aspects to instrument*

[14] Java measurements were performed on a 1GHz Athlon 4 ("Thunderbird") running Sun's Java 1.4.2_03 on a Linux 2.6.3 kernel.

[15] We can assume that the 700MHz PIII machine used for the C++ measurements does not perform better than the 1GHz Athlon machine used for the Java measurements.

generic code (dimension 1) and *using generic code in aspects* (dimension 2). We used (in section 2) these dimensions to examine the relevant C++ language elements and derived requirements and implementation ideas for our aspect weaver.

In the following we use the same scheme to examine the other mentioned languages. Table 3 shows the availability of the template-related language features in Generic Java (GJ), Generic C# (GC#), Ada, Eiffel, and C++[16].

Table 3. Language Genericity Features

	GJ	GC#	Ada	Eiffel	C++	Dimension
Template classes/functions	√	√	√	√	√	1
type aliases (typedefs)		(√)			√	2a
Instantiation with native (build-in) types		√	√	√	√	
Non-type template parameters			√		√	2b
template specialization (condition statement)					√	
Recursive instantiation (loop statement)					√	

Dimension 1: Using Aspects to Instrument Generic Code. All of the examined languages provide the feature of template classes and/or functions. A template-aware AOP extension for any of these languages might and should provide the ability to weave aspect code into generic code or certain instantiations of it. While we assume that the necessary extensions to the joinpoint languages are as straightforward as in AspectC++, different consequences on weaver implementations may arise:

- To be able to affect only certain template instances, AspectC++ creates a modified copy of the original template which has been specialized explicitly to promote it to the compiler. Source code weavers for languages that do not support template specialization may run into difficulties here.
- To weave into template code, a weaver has to be able to distinguish between template code and ordinary code. This might become a problem for byte code weavers, if template instances are not visible or distinguishable on this level any more.

Dimension 2: Support for Generic Advice. As explained, we subdivided the relevant language features for dimension 2 into two sets. The language features listed for dimension 2a are helpful to support generic advice code. The language features listed for dimension 2b are additionally necessary to support generative advice code. Up to now, the 2b-features are only available in C++. Only the C++ template language provides elaborate features that make it a Turing-complete functional language on its own which can be used for static metaprogramming. For the other languages we therefore focus on the support of generic advice:

[16] The table contains only those features we considered as relevant from perspective of an aspect weaver. A more elaborated comparison of the genericity support in different programming languages can be found in [11].

- To support generic advice, the argument and result types related to a joinpoint have to be accessible by the advice code at compile time. In AspectC++ it was possible to implement this by an extension of the *joinpoint API* which now provides *type aliases* (typedefs) for all relevant types. However, even though the other examined languages do not offer a similar type alias concept[17], aspect weavers for these languages may provide the type information in another way. For instance, an aspect weaver may offer special keywords for this purpose that are resolved to the fully qualified actual parameter and result types at weaving time.
- Generic Java restricts template instantiations to non-primitive types only. This implicitly narrows the applicability of generic advice to functions that do not use the native build-in data types (e.g. `int`) in their signature. However, aspect weavers for Java may overcome this limitation by providing automatic boxing and unboxing of primitive data types to their corresponding object types (e.g. `java.lang.Integer`)[18].

4.2 Summary

In principle, all of the examined languages are candidates for template-aware aspect-oriented language extensions. Weaving in generic code (dimension 1) as well as generic advice (dimension 2a) should be feasible in these languages, too. However, while in AspectC++ it was possible to use built-in C++ language features for some important parts of the implementation, weavers for other languages may encounter their own difficulties here. Template-metaprogramming is a very powerful, but unfortunately somewhat C++-specific technique. Even if desirable, it is unlikely that the combination of AOP with generative programming (dimension 2b) is applicable to any of the other examined languages.

5 Related Work

No publications deal with the combination of aspects and templates so far. The few existing work focuses on attempts to "simulate" AOP concepts in pure C++ using advanced template techniques or macro programming [8, 3, 10]. In these publications it is frequently claimed that, in the case of C++, a dedicated aspect language like AspectC++ is not necessary. There is a word of truth in it. Technically speaking, the instantiation of advice code (according to a specific joinpoint at weave time) and the instantiation of a template (according to a set of template parameters at compile time) are similar processes. However, this is a too operational view on the ideas of AOP. The important difference is *where and how* instantiations have to be specified. A class has always to be *explicitly* prepared to be parameterizable by templates. The instantiation and context of a parameterized class, has to be described *explicitly* as well, namely at the point it is actually utilized. The instantiation points of an advice, are, in contrast, described *implicitly*

[17] GC# supports type aliases only on local namespace scope. The Ada `type` keyword, which seems to be somewhat similar to C++ `typedef` at first sight, actually introduces a new distinct type that is not implicitly convertible to any other type.

[18] AspectJ, for instance, already uses this technique for the argument/result accessors in its runtime joinpoint API.

by a joinpoint description language in the aspect code, outside the class definition and class utilization. The advice context is *implicitly* available through the joinpoint API. To our understanding, this non-invasive, declarative approach is at heart of aspect-oriented programming.

OpenC++ [7] 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[16] and CodeBoost[5] 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.

6 Summary and Conclusions

The aim of this work was to investigate the combination of aspects with generic and generative programming in AspectC++. We divided this large topic into mainly two dimensions of interest, namely *using aspects to instrument generic code* and *using generic code in aspects*. We examined the relevant features of the C++ template language, used this to derive the requirements to the AspectC++ language, and presented some details about their incorporation into our AspectC++ weaver[19]. We state that weaving in template instances is not more than a consequent extension of the target scope of aspects. However, providing advice code with additional context information to exploit the power of generic code and metaprograms significantly increases the expressive power of aspect implementations as well as the "on-demand" applicability of template metaprograms. The benefits of such *generic advice* were demonstrated by a case study with a generative implementation of a cache policy. Other useful examples would include introspection, constraint checking or marshaling. Furthermore, we showed that the application of such a policy as a generative aspect does not lead to a significant performance overhead.

We also discussed if and how our insights on combining AOP with generic and generative programming in AspectC++ are applicable to languages that offer a similar model for generic types. We concluded that support to weave in generic code and basic support for generic advice is feasible in these languages, too, even though its implementation into aspect weavers might be difficult. However, even if desirable, it is unlikely that other languages will support the powerful combination of AOP with template metaprogramming in near future. This is a unique feature of AspectC++.

References

1. Loki: A C++ library of designs, containing flexible implementations of common design patterns and idioms. http://sourceforge.net/projects/loki-lib/.

[19] The necessary AspectC++ extensions are already implemented and will be published in version 0.9 of the compiler.

2. Andrei Alexandrescu. *Modern C++ Design: Genenric Programming and Design Patterns Applied*. Addison-Wesley, 2001. ISBN 0-20-17043-15.
3. Andrei Alexandrescu. Aspect-Oriented Programming in C++. In *Tutorial held on the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*, November 2002.
4. Calvin Austin. J2SE 1.5 in a Nutshell. Technical report, Sun Microsystems, Inc., February 2004. http://java.sun.com/developer/technicalArticles/releases/j2se15.
5. Otto Skrove Bagge, Karl Trygve Kalleberg, Magne Haveraaen, and Eelco Visser. Design of the CodeBoost transformation system for domain-specific optimisation of C++ programs. In Dave Binkley and Paolo Tonella, editors, *Third International Workshop on Source Code Analysis and Manipulation (SCAM 2003)*, pages 65–75, Amsterdam, The Netherlands, September 2003. IEEE Computer Society Press.
6. G. Bracha and N. Cohen et al. Adding Generics to the Java Programming Language: Participant Draft Specification. Technical Report JSR-000014, Java Community Process, April 2002. http://jcp.org/aboutJava/communityprocess/review/jsr014/index.html.
7. Shigeru Chiba. Metaobject Protocol for C++. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 285–299, October 1995.
8. Krysztof Czarnecki, Lutz Dominick, and Ulrich W. Eisenecker. Aspektorientierte Programmierung in C++, Teil 1–3. *iX, Magazin für professionelle Informationstechnik*, 8–10, 2001.
9. Krysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming. Methods, Tools and Applications.* Addison-Wesley, May 2000. ISBN 0-20-13097-77.
10. Christopher Diggins. Aspect-Oriented Programming & C++. *Dr. Dobb's Journal of Software Tools*, 408(8), August 2004.
11. R. Garcia, J. Järvi, A. Lumsdaine, J. Siek, and J. Willcock. A comparative study of language support for generic programming. In *Proceedings of the 18th Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '03)*, pages 115–134, Anaheim, CA, USA, October 2003.
12. Andrew Kennedy and Don Syme. The design and implementation of generics for the .NET Common Language Runtime. In *Proceedings of the ACM Sigplan Conference on Programming Language Design and Implementation (PLDI'01)*, June 2001.
13. Gregor Kiczales, Erik Hilsdale, Jim Hugonin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In J. Lindskov Knudsen, editor, *ECOOP 2001 – Object-Oriented Programming*, volume 2072 of *LNCS*. Springer-Verlag, June 2001.
14. Howard Kim. AspectC#: An AOSD implementation for C#. Master's thesis, Trinity College, Dublin, Ireland, September 2002.
15. Donal Lafferty and Vinny Cahill. Language-independent aspect-oriented programming. In *Proceedings of the 18th Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '03)*, pages 1–12, Anaheim, CA, USA, October 2003.
16. Sibylle Schupp, Douglas Gregor, David R. Musser, and Shin-Ming Liu. Semantic and behavioral library transformations. *Information and Software Technology*, 44(13):797–810, 2002.
17. Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. AspectC++: An Aspect-Oriented Extension to C++. In *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, pages 53–60, Sydney, Australia, February 2002.
18. Todd Veldhuizen. Template metaprograms. *C++ Report*, May 1995.
19. Todd Veldhuizen and Kumaraswamy Ponnambalam. Linear algebra with C++ template metaprograms. *Dr. Dobb's Journal of Software Tools*, 21(8):38–44, August 1996.