On Typesafe Aspect Implementations in C++

Daniel Lohmann and Olaf Spinczyk
{dl,os}@cs.fau.de

Friedrich-Alexander-University Erlangen-Nuremberg, Germany

Abstract. Compared to other languages, the C++ language offers a less powerful runtime type system, but a very powerful static type system. In AspectC++, this is addressed by an extended join-point API that provides static type information at compile-time and type-safe access to join-point-specific context information. In this paper we show, how the use of static type information leads to the development highly generic, but type-safe aspects that fit well into the C++ language model. This is demonstrated by an example.

1 Introduction

Compared to languages like Java and C#, the C++ language has a less powerful runtime type system, but a more powerful compile-time (static) type system. C#, while still beeing a statically typed language, implements a unified type system where even primitive value types offer the interface of the one and only root class `System.object`. In Java all class types derive from `Java.lang.Object`. Due to autoboxing it is possible in both languages (Java beginning with Java 5) to pass value type instances as object references. Basically, Java and C# allow to treat "everything as an object" at runtime[1]. This facilitates the development of "type generic code", in the sense that such code can deal with objects of any type *at runtime*.

In C++ there is no such common root class and the C++ runtime type information (RTTI) system offers only a very limited set of runtime services. On the other hand, C++ implements a static type system that offers a very high level of expressive power, based on operator and function overloading, argument dependend name lookup and C++ templates. This facilitates the development of highly generic code that can be instantiated *at compile-time* with any type. In general, the C++ philosphy is to use *genericity at compile time*, while Java and C# advise *genericity at runtime*[2]. The C++ model of compile-time genericity has some clear advantages, as it allows a good separation of concerns, typically results in very efficient code, and implicitly ensures type-safety.

Type genericity is particularly important for the development of aspects, as aspects are typically intended to be broadly reusable and applicable. For example, the implementation of a tracing aspect that logs all actual parameter and result values of function invocations, should be independent of the affected function's signature, i.e. on its argument and result types. In AspectJ this is realized by providing a *runtime join-point*

[1] Actually, it is the Smalltalk language that carried the "everything is an object" idea to the extremes. However, Smalltalk offers no static type system.

[2] This is even true with Java generics introduced in the Java 5, which are basically a syntatic wrapper around the "treat everyting as an object" philosophy.

API to advice implementations, which offers a unified interface to access the join-point's context information. This information includes the number of parameters, the argument and return values (as `Object`), and (via the interface of `Object` and Java's reflection capabilities) their runtime types. Our AspectC++ language offers very similar runtime mechanisms. Additionally, the AspectC++ join-point API supports an alternative *type-safe* access to all parameter and result values. For this purpose, we extended the AspectC++ runtime join-point API by a *compile-time join-point API*, which provides static type information about the current join-point at compilation time. We call advice, which depends on static type information from the compile-time join-point API *generic advice* [10].

1.1 Outline

The aim of this paper is to show, how the AspectC++ notion of generic advice can be used to develop reusable, but type-safe aspect implementations that fit well into the C++ philosphy of "doing as much as possible statically". This is demonstrated by an example. The example is an aspect that facilitates exception-based error propagation for legacy third-party C-libraries like the Win32 API.

The rest of this paper is structured as follows. The next section provides a brief introduction into the AspectC++ language and terminology and describes the AspectC++ join-point API. Section 3.1 explains the example project. Afterwards, some details about the weaving process of AspectC++ are given in section 4. This is followed by a discussion of the advantages and limitations of our approach in section 5. Finally, we give an overview of related work and briefly summarize the paper.

2 AspectC++ Concepts and Terminology

AspectC++ [13] 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. While beeing strongly influenced by the AspectJ language model [7, 8], AspectC++ has to support many additional concepts that are unique to the C++ domain. This ranges from operator overloading, const correctness and multiple inheritance up to weaving in template code.
The AspectC++ compiler, plugin for Eclipse, and documentation are available under open source license from the AspectC++ homepage [1].

2.1 Basic Concepts [10]

The AspectC++ terminology is inspired by the terminology introduced by AspectJ. The most relevant terms are *join-point* and *advice*. A *join-point* denotes a specific weaving position in the target code (often called component code, too). Join-points are usually 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*. In AspectC++ the sentences of the join-point description language are called *pointcut expressions*. For example the pointcut expression

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

describes all calls to member functions of the class `Service`. The pointcut expression

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

describes again all calls to member functions of the class `Service`. By combining (`&&` operation) these join-points with the `cflow` pointcut function these join-points 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* [1].

The core of the join-point language are match-expressions. In AspectC++ these expressions are quoted strings where `%` and ... can be used as wildcards[3]. They can be understood as regular expressions matched against the names of known program entities like functions or classes.The aspect code that is actually woven into the target code at the join-points is called *advice*. Advice is bound to a set of join-points (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 join-point which it affects, at runtime by a *join-point 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";
  cout << endl;
}
```

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

2.2 The AspectC++ Join-point API

Table 1 shows an excerpt from the join-point API, describing those parts that are relevant in the context of this paper. The elements that are based on join-point-specific static type information are emphasized. The upper part (types and enumerators) provides compile-time type information, which can be used to instantiate generic code or template metaprograms by advice. The lower part (non-static methods) provides a

[3] In AspectJ `*` is used as a wildcard, but this would result in ambiguities in C++. However, the match mechanism exists in AspectJ, too.

type-safe interface to the join-point context. These methods are bound at compile-time, but called at runtime. For example, the function `Arg<i>::ReferredType *arg()` offers a type-safe way to access argument values if the argument index is known at compile time. Inside the advice body, the static part of the join-point API is provided as a class `JoinPoint`. At runtime, the non-static members are accessed using the `JoinPoint *tjp` pointer.

compile-time types and enumerators:	
`That`	type of the affected class
`Target`	type of the destination class (for call join-points)
`Arg<`*`i`*`>::Type` **`Arg<`*`i`*`>::ReferredType`**	type of the i'th argument with $0 \leq i <$ ARGS
`Result`	result type
`ARGS`	number of arguments
`JPID`	unique numeric identifier for this join-point
`JPTYPE`	type of this join-point (call / execution / construction / destruction)
runtime static methods:	
`const char *signature()`	signature of the affected function
...	
runtime non-static methods:	
`void proceed()`	execute original code (around advice)
`That *that()`	object instance referred to by this
`Target *target()`	target object instance of a call (for call join-points)
`Arg<`*`i`*`>::ReferredType *arg()`	argument value instance of the i'th argument
`Result *result()`	result value instance
...	

Table 1. An Excerpt from the AspectC++ Join-point API

3 Using Static Typing in Aspects - An Example

3.1 Motivation

Every program has to deal with the fact that operations may fail at runtime. Programming language concepts for propagation and handling of runtime errors have evolved over time. Today most developers favor *exceptions* for this purpose. However, especially in the C/C++ world, there are still hundreds of legacy libraries that do not support exception handling and follow a more traditional approach of error handling by reporting an error situation via the function's *return value*. An error is returned either as an error code, a boolean flag or as a special "magic value". In the latter cases, a more descriptive error code can typically be retrieved by calling a special library function or reading a global variable. The C runtime library (CRT), for instance, provides the global variable `errno` for this purpose.

```
#include <windows.h>
HANDLE g_hConfigFile = NULL;

LRESULT WINAPI WndProc( HWND hWnd, UINT nMsg, WPARAM wParam, LPARAM lParam ) {
  HDC     dc = NULL;
  PAINTSTRUCT ps = {0};

  switch( nMsg ) {
    case WM_PAINT:
      dc = BeginPaint( hWnd, &ps );
      ...
      EndPaint(hWnd, &ps);
      break;

    case ...

    default:
      return DefWindowProc(hWnd, nMsg, wParam, lParam);
  }
  return 0;
}

int WINAPI WinMain( ... ) {
  g_hConfigFile = CreateFile( "example.config", GENERIC_READ, 0,
                                          NULL, OPEN_EXISTING, 0, NULL );

  WNDCLASS wc = {0, WndProc, 0, 0, ... , "Example_Class"};
  RegisterClass( &wc );

  HWND hwndMain = CreateWindowEx( 0, "Example_Class", "Example", ... );
  UpdateWindow( hwndMain );

  MSG msg;
  while( GetMessage( &msg, NULL, 0, 0 ) ) {
    TranslateMessage( &msg );
    DispatchMessage( &msg );
  }
  return 0;
}
```

Fig. 1. A Typical Win32 Application

Many libraries overload the indication of a runtime error with the functions regular result. In such libraries, the result value has to be checked against a “magic value” to determine if there was an error. The “magic value” itself often depends on the result type. For instance, functions that perform floating point calculations return a `double` which is either the result of the calculation or the special not-a-number (`NaN`) value in case of an error. The CRT function `fopen()` returns a `FILE*` that is the handle to the opened file or `NULL` in case of an error.

Error checking and handling by validation of function results is cumbersome. Typically, in the client code each function call needs to be surrounded by an if statement with at least two or three additional lines to do the error handling. This results in heavily tangled and almost unreadable code. As a consequence, this kind of error handling is often “forgotten” as C-style languages allow programmers to simply ignore the results of a function call. At runtime this may lead to undefined behavior when the program continues execution with invalid internal state. With error propagation by exceptions this

could not happen, as an exception "can't be ignored" and reported faults are thereby detected as early as possible. Hence, it is a good idea to integrate calls to legacy libraries into the error-handling-by-exception model. As this is a crosscutting concern, we strive for a generic aspect-oriented solution. In the following we describe such a generic aspect for one of most popular (and disliked) legacy libraries: the Win32 API.

3.2 The Example Application

Figure 1 shows the listing of a typical Win32 application. `WinMain()` is the entry point, which performs the usual sequence of Win32 API calls to initialize the application and start the main message loop: First a configuration file is opened (`CreateFile()`) and the window class for the application's main window is registered (`RegisterClass()`). Afterwards the main window is created (`CreateWindowEx()`) and an initial `WM_PAINT` message is sent to it (`UpdateWindow()`). The `WM_PAINT` message is handled by the `WndProc()` window procedure, which acquires a device context (`BeginPaint()`), draws the window's content (not shown here) and then releases the device context (`EndPaint()`). After the main window was created, the application finally enters the message loop to perform all further processing.

Even if the application does not contain any error checking code, any of the above mentioned API functions may fail at runtime. They all follow the Win32 convention by indicating a failure by a "magic" return value and, in case of failure, providing more information about the reason via `GetLastError()`.

3.3 An Aspect to Throw on Win32 Errors

Our goal is now to develop an aspect that implements an exception-based error handling for calls to the Win32 API. The general idea is to give after call advice to all Win32 API functions. In the advice body, the return value of the API function is checked and, in case of an error, an exception is thrown:

```
#include "Win32Error.h"
aspect ThrowWin32Errors {
  advice call( win32::Win32API() ) : after() {
    if( win32::IsError( *tjp->result() ) {
      throw win32::Exception(...);
    }
  }
};
```

The advice affects all calls to API functions that are described by the (externally defined) pointcut `win32::Win32API()`. Its implementation uses a helper function `win32::IsError()` that returns `true` if the passed result value, retrieved via `tjp->result()`, indicates a failure. This is done by checking the result against the "magic value". The actual implementation of the helper function depends on the API function called, or, more precisely, on its return type. In the example the following Win32 API functions are used:

– `EndPaint()` and `UpdateWindow()` are of type `BOOL`.
  `BOOL` functions indicate an error by returning `FALSE`.
– `BeginPaint()` is of type `HDC`.
  `HDC` functions indicate an error by returning `NULL`.
– `CreateWindowEx()` is of type `HWND`.
  `HWND` functions indicate an error by returning `NULL`.
– `RegisterClass()` is of type `ATOM`.
  `ATOM` functions indicate an error by returning `0`.
– `CreateFile()` is of type `HANDLE`.
  `HANDLE` functions indicate an error by either returning `NULL` or `INVALID_HANDLE_VALUE`.

These few API functions already cover a significant part of the different return types and "magic values" used by the Win32 API. The `win32::IsError()` helper function is overloaded for each of these types to perform the check against the type-dependent "magic values" (Figure 3, lines 12–26). The compiler's overload resolution deduces (at compile-time) for each join-point the correct helper function to call (Figure 2). In the case the advice affects calls to a function of a type no compatible helper function is defined for, the overload resolution process fails and results in a compile-time error. Note that this generic implementation of the advice code is only possible, because `tjp->result()` is type-safe, as it returns a pointer of the real static type of the affected function.

3.4 Providing Context Information

Our aspect performs a type-safe validation of the results of Win32 API calls and throws a `win32::Exception` in case of an error. The exception object should include all context information that can be helpful to figure out the reason for the actual failure. Besides the Win32 error code, this should include a human readable string describing the error, the signature of the called function (retrieved with `tjp->signature()`) and the actual parameter values that were passed to the function.

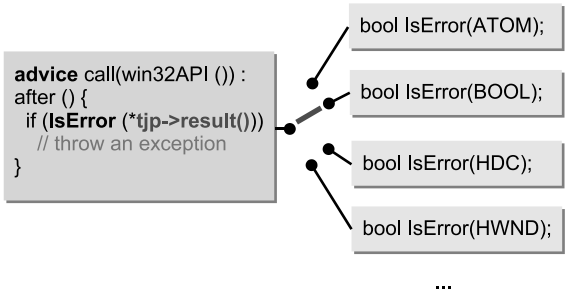

Fig. 2. A Join-Point specific Compile-Time Switch

```
1  namespace win32 {
2
3    struct Exception {
4      Exception( const std::string& w, DWORD c )
5        : where( w ), code( c )
6      {}
7
8      std::string where;
9      DWORD   code;
10   };
11
12   inline bool IsError( HANDLE res ) {
13     return res == NULL || res == INVALID_HANDLE_VALUE;
14   }
15   inline bool IsError( ATOM res ) {
16     return res == 0;
17   }
18   inline bool IsError( HWND res ) {
19     return res == NULL;
20   }
21   inline bool IsError( HDC res ) {
22     return res == NULL;
23   }
24   inline bool IsError( BOOL res ) {
25     return res == FALSE;
26   }
27
28   // Translates a Win32 error code into a
29   // readable text message using the Win32
30   // FormatMessage() function
31   std::string GetErrorText( DWORD code ) {
32     char res[ 256 ];
33     FormatMessage(... , code, 0, res, ...);
34     return res;
35   }
36
37
38   pointcut  Win32API() = "% CreateWindow%(...)"
39                        || "% RegisterClass%(...)"
40                        || "% BeginPaint(...)"
41                        || "% UpdateWindow(...)"
42                        || "% CreateFile%(...)"
43                        || ...;
44 } // namespace Win32
```

Fig. 3. Win32 Errorhandling Helper

The tricky part is to build a string 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 services like `toString()`. The C++ concept to get a string representation of any type is based, once again, on static typing. It 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 metaprogram (Figure 4, lines 4–19), which is instantiated at compile-time with

```
#include "Win32Error.h"

aspect ThrowWin32Errors {
  // template metaprogram to stream a commaseparated sequence of
  // arguments available at a joinpoint
  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::IsError( *tjp->result() ) ) {
      ostringstream os;
      DWORD c = GetLastError();

      os << "WIN32 ERROR " << c << ": "
         << win32::GetErrorText(c) << 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(), c );
    }
} };
```

Fig. 4. An Aspect to Throw Win32 Errors as Exceptions

```
#include "Win32Error.h"

aspect CatchWin32Errors {
  advice execution("% WinMain(...)"
        || "% WndProc(...)" ) : around() {
    try {
      tjp->proceed();
    }
    catch( win32::Exception& e ) {
      MessageBox( NULL, e.where.c_str(), NULL, MB_ICONERROR );
    }
} };
```

Fig. 5. A Simple Aspect to Catch Win32 Errors

the `JoinPoint` type (line 35) 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 (line 10). This implementation is, again, type-safe. The compiler automatically deduces the stream operator to call for a specific type. If no compatible operator is available, a compile-time error is thrown.

3.5 Handling Error Conditions

The exception handling itself is implemented, for demonstration purposes, as another simple aspect (Figure 5). It just displays the context information in a message box. If, for instance, the `CreateFile()` call (Figure 1, line 24) fails, because the configuration file `example.config` does not exist, the following error message pops up:

For real-world applications, the `CatchWin32Errors` aspect can be easily extended to implement advanced error handling concepts. For instance, a detailed error log can be created and the Win32 debugger API might be used to dump the call stack before the application is terminated.

4 Implementation Details

AspectC++ is a source-to-source weaver that transforms AspectC++ programs into C++ programs. The woven code can then be built with any standard-conforming C++ compiler, like g++ or VisualC++. In this section we show some details about this transformation process, focussing on generic advice code and the compile-time join-point API.

4.1 Aspect Transformation

AspectC++ generates a C++ class with a unique name for each join-point 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 join-point 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 join-point class with `JoinPoint::`*`Typename`*. Indirectly these types can also be used by using the type-safe argument and result access function. The following code fragment shows advice code after its transformation into a template function.

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

4.2 Argument Type Sequences

In AspectC++ template-metaprograms can be used to iterate over the argument type sequence of a join-point at compile time, as shown in the example. However, these sequences have to be provided in a "metaprogram-friendly" way. Just generating `ArgType0`, `ArgType1`, ..., `ArgTypeN` would not allow metaprograms to iterate over these types. For this purpose, the generated join-point-specific classes contain a template class `Arg<I>` which provides all the type information for the I'th argument as typedefs.

Sequences of types can be implemented by recursive template definitions as in the Loki[2] `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 a part of the generated type for the call join-point to `RegisterClass()` in the `WinMain()` function (Figure 1, line 28).

```
struct TJP_main_1 {
  typedef ::ATOM Result;
  typedef void That;
  typedef void Target;
  enum { ARGS = 1 };
  template <int I> struct Arg {};
  template <> struct Arg<0> {
    typedef ::WNDCLASSA *Type;
    typedef ::WNDCLASSA *ReferredType;
  };
  void **_args;
  Result *_result;
  inline Result *result() {return _result;}
  inline static const char *signature () {return ...;}
  inline void *arg (int n) {return _args[n];}
  template<int I> typename Arg<I>::ReferredType *arg(){
    return (typename Arg<I>::ReferredType*) arg(I);
  }
};
```

Note that only the `_result` and `_args` attributes consume memory at runtime, as everything else is resolved at compile-time.

5 Discussion

As demonstrated by the `ThrowWin32Errors` aspect, the technique of using static typing for generic advice implementations has some clear advantages regarding genericity and type-safety. On the other hand, the strong focus on static typing and the template-based implementation also implies some potential drawbacks. In this section we discuss the major advantages and limitations of our approach.

5.1 Advantages of Generic Advice

Genericity is achieved, as generic advice can be applied to functions with any signature, even if they use primitive or POD data types. This seamless support of non-class types is particulary important in the C/C++ domain. It is not possible to implement a unified access to instances of such types by extending their interface, e.g. using (baseclass-) introductions or other typical AOP idioms.

Separation of concerns is improved, as type-specific parts of the implementation (like the comparison with a "magic value") are separated out from the advice implementation in own external program entities. Thereby most of the advice code can be reused. Moreover, this makes the aspect code more stable with respect to changes in the component code. If, for instance, the result type of a function is changed to some unknown new type, the missing helper function is detected at compile-time. A non-generic aspect implementation that gives one advice per return type would silently miss to match the function's new signature.

Type-safety is guaranteed, as type errors are detected early at compile-time. Costly and potentially dangerous runtime casts are avoided. In languages that do not offer a type-safe access to the join-point's context information, problems (like a missing helper function) may not be detected before runtime.

5.2 Potential Limitations and Disadvantages

Code bloating is a potential problem, as generic advice is intantiated per join-point which might result in a high number of (similar) template instantiations, each being compiled seperately into the machine code. This is a general and well known issue in the C++ domain. It is difficult to judge its effects on real applications, as they depend on many other properties, especially the optimization capabilities of the compiler. For small template functions, which are inlined anyway, it has no effect at all.

In the example application, around 200 (`BeginPaint()`, 2 arguments) to 1000 (`CreateWindowEx()`, 11 arguments) additional bytes of code are generated for each of the Win32 function matched by the `ErrorException` aspect. Most of this overhead is induced by the generated streaming code. We estimate that a hand-written tangled implementation of the same crosscutting concern would result in similar costs, as it has to contain the same amount of streaming code. However, *additional* calls of the same Win32 function result in (almost) *no* additional overhead. In these cases the build system seems to be able to detect and optimize away the semantically identical template instantiations[4].

[4] All measurements were done using Microsoft Visual C++ 2003 using /O1 optimizations.

Compile-time fixation limits the approach to those types that are known at build-type. Generic advice can not be instantiated for additional types loaded dynamically at runtime. This is an implicit property of static typing. It is no real limitation for languages like C++ or Ada, as these languages do not support runtime type loading anyway. However, for Java or C#, which provide runtime class loading, this might be an issue. A possible solution for such languages is to rely on static type information as much as possible, but fall back to runtime type checking for types that are not known at build time.

Strong type semantics is a prerequisite of taking advantage of the approach. If, for instance, there is no general policy which "magic values" of a return type indicate a failure, it is not possible to bind the required test function solely on the base of the return type. However, if there are only a few exceptions from an otherwise general rule, they can be handled by seperate advice.

The example `ThrowWin32Errors` aspect, for instance, can already be applied to most functions of the Win32 base API in any program. It is easy to achieve complete coverage by implementing the helper functions for the remaining return types. Very few API functions, however, do not follow the general Win32 error handling conventions[5]. Luckily, these few exceptions can easily be handled by giving specific advice for them.

5.3 Conclusions

Overall, the approach fits well into languages like Ada and C++ that have a static typing philosophy. Languages like Java or C#, which support static, but emphazise dynamic typing, would even benefit from generic advice, although it might be necessary to extend the approach for these languages by some runtime mechanism that deals with dynamically loaded classes. Such combination would make the most of both worlds.

6 Related Work

So far, no publications focus on the development of generic, but type-safe aspects in languages with a sophisticated static type system as C++. C#, being currently a language with focus on dynamic typing, will be extended by additional static concepts (templates) in the next version [6]. However, the proposed extensions for integrating AOP into C# (e.g. [12, 11]) do not cover static typing in aspects, yet.

Several extensions have been suggested to increase the static genericity of AspectJ. Instead of an extended join-point API, they are based on an extension of the join-point description language by context-dependend logic variables that are bound at weaving-time. *Sally* [5] focusses on genericity for structural aspects and proposes *parametric introductions* as an extension of the inter-type declaration mechanism in AspectJ and Hyper/J. *LogicAJ* [9] supports a similar mechanism called *generic introductions* and facilitates the development of generic advice code as well as reasoning over non-type

[5] The `TlsGetValue()` function, which is used to retrieve a thread-local storage value, is an example. It does not use a "magic value", but indicates success by clearing the `GetLastError()` value.

program entities like method names. While the use of of a logical language for this purposes has some clear advantages, it also leads to a high level of complexity. As C++ is already a "rich" multi-paradigm language with respect to complexity and expressive power, such an approach implies the risk of introducing redundant language concepts. For instance, a compile-time type deduction engine is already available through templates. For AspectC++, our goal is therefore to *carefully integrate* AOP concepts with the *existing* idioms and the philosophy of the C++ language.

The few existing work in the AOP/C++ domain focusses on using the preprocessor or static type concepts like overloading and advanced template techniques to "simulate" AOP in pure C++ [3, 15, 4].

7 Summary

The aim of this paper was to demonstrate, how static type information and a type-safe join-point API can be used in generic advice for the development of broadly applicable and type-safe aspects. By seperating out the type-specific parts (like the detection of an error result) into own external functions, both a good separation of concerns and a high level of type-safety is achieved.

Generic advice integrates well with the idioms of a language with a strong static type system as C++, where it is common and desirable to "decide things at compile time". As shown in the example, this involves even the utilization of template-metaprogramming techniques in advice code, e.g. to iterate over the function's arguments. Template-metaprogramming is known to be tricky and cumbersome, however, there are promising attempts to reduce its complexity by carefully extending the C++ language [14]. Currently, templates permit to reach a level of genericity and type-safety that is otherwise not feasable. Both, genericity and type-safety are very important properties for any aspect that is intended for reuse, like aspects from an aspect library. Such aspects are potentially applied to functions with any signature, which includes functions that use primitive and POD types. Thus, they should be able to deal with all those types and, even more important, raise potential type errors at compile-time.

References

1. AspectC++ homepage. http://www.aspectc.org/.
2. Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001.
3. Krysztof Czarnecki, Lutz Dominick, and Ulrich W. Eisenecker. Aspektorientierte Programmierung in C++, Teil 1–3. *iX, Magazin für professionelle Informationstechnik*, 8–10, 2001.
4. Christopher Diggins. Aspect-Oriented Programming & C++. *Dr. Dobb's Journal of Software Tools*, 408(8), August 2004.
5. Stefan Hanenberg and Rainer Unland. Parametric introductions. In Mehmet Akşit, editor, *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD '03)*, pages 80–89, Boston, MA, USA, March 2003. ACM Press.
6. 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 '00)*, June 2001.

7. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. Getting started with AspectJ. *Communications of the ACM*, pages 59–65, October 2001.
8. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In J. Lindskov Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP '01)*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer-Verlag, June 2001.
9. Günter Kniesel, Tobias Rho, and Stefan Hanenberg. Evolvable pattern implementations need generic aspects. In *Proceedings of the ECOOP'04 Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'04)*, Oslo, Norway, June 2004.
10. Daniel Lohmann, Georg Blaschke, and Olaf Spinczyk. Generic advice: On the combination of AOP with generative programming in AspectC++. In G. Karsai and E. Visser, editors, *Proceedings of the 3rd International Conference on Generative Programming and Component Engineering (GPCE '04)*, volume 3286 of *Lecture Notes in Computer Science*, pages 55–74. Springer-Verlag, October 2004.
11. Hridesh Rajan and Kevin Sullivan. Eos: Instance-level aspects for integrated system design. In *Proceedings of the 4th Joint European Software Engineering Conference and ACM Symposium on the Foundations of Software Engineering (ESEC/FSE '03)*, September 2003.
12. Mario Schüpany, Christa Schwanninger, and Egon Wuchner. Aspect-oriented programming for .NET. In *Proceedings of the 1st AOSDWorkshop on Aspects, Components and Patterns for Infrastructure Software (AOSD-ACP4IS '02)*, March 2002.
13. 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 '02)*, pages 53–60, Sydney, Australia, February 2002.
14. Daveed Vandevoorde. Reflective metagrogramming. In *Presentation held on the ACCU 2003 Spring Conference*, Oxford, UK, April 2003. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1471.pdf.
15. Detlef Vollmann. Visibility of join-points in AOP and implementation languages. In *Second GI Workshop on Aspect-Oriented Software Development*, Bonn, Germany, February 2002.