Author's private version of article. Please cite as: Tartler, Lohmann, Scheler, Spinczyk: *AspectC++: An Integrated Approach for Static and Dynamic Adaptation of System Software*. Knowledge-Based Systems 23 (2010), pp. 704–720, doi:10.1016/j.knosys.2010.03.002

AspectC++: An Integrated Approach for Static and Dynamic Adaptation of System Software [⋆]

Reinhard Tartler [∗], Daniel Lohmann, Fabian Scheler, and
Olaf Spinczyk [#]

Friedrich-Alexander University Erlangen-Nuremberg, Germany
[#] Technical University Dortmund, Germany

Abstract

Modern computer systems require an enormous amount of flexibility. This is especially the case in low-level system software, from embedded devices to networking services. From literature and practice, various approaches to modularize and integrate adaptations have been investigated. However, most of this work is implemented with dynamic languages that offer extensive run-time support and enable easy integration of such approaches. System software is written in languages like C or C++ in order to minimize utilization of system resources and maximize efficiency. While for these languages highly optimized and reliable compilers are available, the support for static and dynamic adaptation is rather limited. In order to overcome these limitations, we present an adaptation approach that is based on a sophicticated combination of static and dynamic aspect weaving for aspects written in AspectC++. This facilitates the incremental evolution and deployment of system software that has to be "always on". We demonstrate the feasibility of our approach and its applicability to two pieces of system software, namely the Squid web proxy and the eCos operating system, which is used in the domain of resource-constrained deeply embedded systems.

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

---

1 Introduction

Infrastructure software, such as network services or operating systems, is often faced with high availability demands. This poses a real challenge when it comes to

---

[⋆] This work was partly supported by the German Research Council (DFG) under grant no. SCHR 603/4, 603/7-1, and SP 968/2-1.

[∗] Correspondence to: Reinhard Tartler, Friedrich-Alexander-Universität Erlangen-Nüremberg, Informatik 4, Martensstr. 1, 91058 Erlangen, Germany, Tel.: +49 9131 8528731

Preprint submitted to Elsevier *accepted March 3, 2010*

deploying adaptations (such as a feature extension or a bug fix) to the running system. Application-specific approaches (such as plugins) to load adaptation modules at run time often provide an extension interface in order to modify the behavior of the base application. However, this approach inherently limits what the adaptation module can change; in fact, it fails for modules that require more flexibility than what the extension interfaces provides. A good example for such an adaptation module is a bugfix, which potentially affects any part of the application.

One of the hardest problems with implementing adaptation modules is to specify *where* to apply *what* changes in the base program. Aspect-oriented programming (AOP) languages provide mechanisms to solve these challenges by *obliviousness* and *quantification* [11]. Obliviousness means that the application of adaptations, called aspects, can be completely oblivious to the component code, in the sense that neither components nor their developers have to be aware of the aspects. Quantification stands for the property that the same aspect code can easily affect several adaptation points.

Most existing AOP approaches can be categorized as either *dynamic* or *static*, referring to the point in time when the actual aspect weaving process is performed. If the aspect weaver performs *static weaving*, the aspects are woven in at compile time, link time, or load time. With *dynamic weaving*, the aspects are woven into an already running program, which promises to overcome the limits that are imposed by traditional extension interfaces.

Dynamic aspect weavers, which feature invasive modification of run-time behavior, are clearly more complex and therefore seldom used than their static counterparts for system software written in C/C++. However, they allow changing the traditional deployment strategy for corrective changes to an already deployed software from having to restart the whole system to a less intrusive processes. In order to bring these benefits to legacy applications, we are looking for an infrastructure that provides enough flexibility for more intrusive modules like bugfix hot-patches or feature extensions. In this article, we present an approach to deploy such adaptation modules flexibly at compile time or run time in low-level system software.

1.1 Problem Statement

Ideally, an AOP user would be able to select the aspect language and the weaving approach independently, solely based on the problem to solve. However, most existing aspect languages provide weaver support for *either* static *or* dynamic weaving only. What should be independent in theory, is tightly coupled in practice: the decision for a particular aspect language involves the decision for either dynamic or static weaving as well. From a user's viewpoint, we have de facto "static" and "dynamic" aspect *languages*. This is especially true with languages that are directly compiled into binary machine code. In the C/C++ domain there *are* observable differences in the provided AOP features: The available "dynamic" aspect languages for C/C++ (such as Arachne [9], TinyC² [37], TOSKANA [12], or KLASY [36]) offer significantly fewer features than their "static" counterparts (such as AspectC [6], AspectC++ [32], Mirjam/WeaveC [26], *Aspicere* and *Aspicere2* [1,2], and *ACC* [14,15]). Especially language features for generic aspects and static crosscutting are hardly supported. This is unsatisfying; the expressive power of an aspect language (to address the "*what*" part of the problem) should not depend on the intended deployment time (the "*when*") and vice versa. From the viewpoint of weaver implementation, it is, however, understandable: Languages that are strongly based on static typing and compile-time genericity offer hardly any support for run-time reflection, not to speak of means for extension, adaptation, or introduction of new types at runtime. In a sense, Ada, C and C++ are "just not designed" to support many AOP features with runtime weaving. Nevertheless, a uniform, feature-rich, and deployment-time independent aspect language would provide numerous benefits; Section 3 lists some motivating application scenarios.

1.2 Our Contribution

We present results from our efforts to add dynamic weaving support to a statically typed and compiled aspect language, for which only static weaving support had existed before. Our approach is based on a novel combination of static and dynamic weaving, which makes it possible to use AspectC++ features such as *generic advice* (statically typed) and *introductions* even for dynamically woven aspects. We are not aware of any other implementation for the C/C++ language domain that offers both, static and dynamic weaving of aspects written in the same aspect language.

Our targeted application domain is applications that run in a resource-constrained environment. For this reason, we cannot afford invasive modifications of the base application, nor a heavy weighted runtime system. Instead we extend our static aspect weaver to collect type information about the adapted software while preparing it for dynamic weaving. This extra information is then used within the C++ template instantiation mechanisms to generate advice code that is executed at runtime.

We analyze and discuss the combination of static and dynamic weaving with respect to two dimensions: language and tools. On the language level, we provide an in-depth analysis of challenging AOP features from the focus of a statically typed base language. On the tool level, we show how we implemented them in a dynamic weaver for AspectC++. Insights about the relationship between static and dynamic weaving on the tool level and an evaluation of our implementation in the context of the Squid web proxy [33] and the eCos operating system [25,10] round up our contribution.

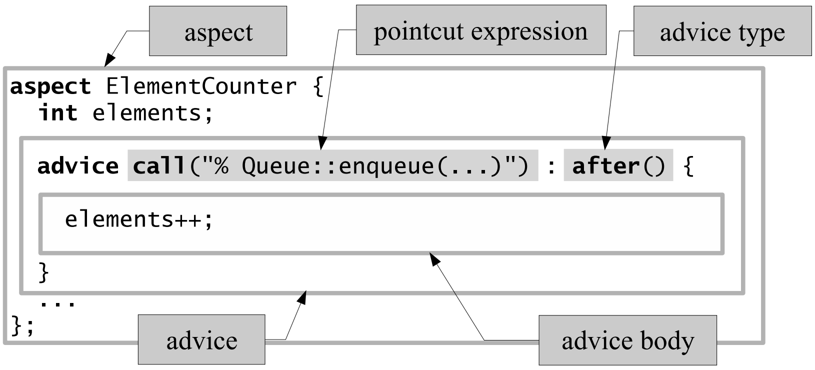


Figure 1. Syntactical Elements of an Aspect in AspectC++

The article extends on previous work, as it provides an actual solution for the problems that have been identified and briefly discussed in [31]. The focus on the implementation challenges of dynamic weaving of static cross-cutting sets it furthermore significantly apart from our previous work on application-tailorable dynamic weaver run-time systems in [13].

1.3 Outline of the Article

We begin with a brief introduction into AOP and the AspectC++ language in Section 2 and the presentation of some motivating application scenarios in Section 3. This is followed by the analysis of the implications with respect to dynamic weaving support in Section 4. Section 5 provides an overview of related work. The concepts and some details of our implementation for AspectC++ are described in Sections 6 and 7, followed by two case studies in Section 8. The first one shows how dynamic weaving can help to dynamically adapt a system service – here the proxy server Squid – both at compile time and at run time. The second study demonstrates how our approach has been successfully deployed in the context of deeply embedded systems. Section 9 discusses the pros and cons of our approach. Finally, our work is summarized and some conclusions are given in Section 10.

2 AOP Concepts at a Glance

Today, most AOP languages use the concepts and terminology that was first introduced by AspectJ[18]. In the remaining parts of this section, we will give a brief overview of the most common AOP language elements in general and the AspectC++ notion in particular, as required for understanding the remaining parts of this article. Even though the introduction is based on AspectC++, it basically holds for any statically woven AOP language.

2.1 Terminology

The most relevant AOP concepts are *join point* and *advice*. An *advice* definition describes a transformation to be performed at specific positions either in the static program structure (*static cross-cutting*) or in the runtime control flow (*dynamic cross-cutting*) of a target program. A *join point* denotes such a specific position in the target program. The target program implicitly exhibits – by its structure and nature – a large set of *potential* join points, which are commonly called *join-point shadows* [17,24]. Advice is given by *aspects* to sets of join points called *pointcuts*. Pointcuts are defined declaratively in a *join-point description language*. The sentences of the join-point description language are called *pointcut expressions*. An *aspect* encapsulates a cross-cutting concern and is otherwise very similar to a class. Besides advice definitions, it may contain class-like elements such as methods or state variables.

As an example, Figure 1 illustrates the syntax of aspects written in AspectC++. The aspect increments the member variable `elements` *after* each call of the function `Queue::enqueue()`. In AspectC++, pointcut expressions are built from *match expressions* and *pointcut functions*. Match expressions are already primitive pointcut expressions and yield a set of *name join points.* Name join-points represent elements of the static program structure such as classes or functions. Technically, match expressions are given as quoted strings that are evaluated against the identifiers of a C++ program. The expression `"% Queue::enqueue(...)"`, for instance, returns a name pointcut containing every (member-) function of the class `Queue` that is called `enqueue`. In the case of overloaded functions with different argument types the expression would match all of them. *Code join points* on the other hand, represent events in the dynamic control flow of a program, such as the execution of a function. Code pointcuts are retrieved by feeding name pointcuts into certain pointcut functions such as `call()` or `execution()`. The pointcut expression `call("% Queue::enqueue(...)")`, for instance, yields all events in the dynamic control flow where a function `Queue::enqueue` is about to be called.

As pointcuts are described declaratively, the target code itself has not to be prepared or instrumented to be affected by aspects. Furthermore, the same aspect can affect various and even unforeseen parts of the target code. These principles of *obliviousness* and *quantification* [11] are considered a major advantage of AOP.

2.2 Static Cross-cutting

An aspect that encapsulates *static cross-cutting* alters the static structure of the program. In most AOP languages, such modifications of the static structure are restricted to the extension of classes by new elements like methods, state variables or base classes.

In AspectC++, the encapsulation of static cross-cutting is supported by a specific type of advice called *slice introduction*. Consider the following aspect, which adds support for thread local storage to a thread control block class:

```
aspect ThreadLocalStorage {
  advice "os::ToC" : slice class {
    int tlsentry;
   public:
    int getTLS() {
      return tlsentry;
    }
    void setTLS(int v) {
       tlsentry = v;
    }
  };
};
```

The above aspect *introduces* a *slice* of class elements, namely the (private) state variable `tlsentry` and some (public) accessor methods `getTLS()` and `setTLS()` into the thread control block class, or, more precisely, into all classes that are matched by the expression "`os::ToC`".

2.3 Dynamic Cross-cutting

An aspect that encapsulates *dynamic cross-cutting* intercepts certain events in the control flow of a running program. Aspects basically provide means to execute some advice code *before*, *after*, or instead of (*around*) the current statement if the event occurs. In the following, this is demonstrated by three different variants of an aspect that intercepts entries into and exits from an operating-system kernel to implement some kernel locking strategy, however, the same pattern can be used to implement locking in any component that needs synchronization. The advice body is identical for all three variants of the `KernelLock_x` aspect: it acquires the lock (which is a member of the aspect), proceeds to the intercepted function (`tjp->proceed()`) and finally releases the lock.

```
aspect KernelLock_1 {
  pointcut kernel() = "% kernel::%(...)";
  os::Lock lock;    // aspect member variable

  advice execution(kernel()) : around() {
    lock.enter();
    tjp->proceed(); // execute the intercepted method
    lock.leave();
 }
};
```

In variant 1, the advice is triggered, whenever any function or method from the class or namespace `kernel` is about to be *executed*. This, however, works only if kernel functions do not invoke each other, as calls to `lock.enter()`/`lock.leave()`

must not be nested. Variant 2 provides a less restrictive solution by intercepting the kernel invocation on the *caller* side:

```
aspect KernelLock_2 {
  ...
  advice call(kernel())
  && !within(kernel()) : around() {
    ...
  }
};
```

The `call()` pointcut function yields all events in the control flow, where a given function is about to be *called*. The `within()` pointcut function simply returns all join points in the given classes, functions or namespaces. By *intersecting* (`&&`) all calls to `kernel()` with the negation (`!`) of all join points inside `kernel()`, the pointcut expression finally evaluates to those calls to a `kernel()` function that are not made from a `kernel()` function itself. This, however, has another potential drawback: as the interception now takes place on the caller side, not only the kernel code, but also the client code has to be woven with the aspect. In many cases, this is not feasible. In variant 3 kernel invocation is therefore again intercepted on the callee side, but further filtered to certain control flows:

```
aspect KernelLock_3 {
  ...
  advice execution(kernel())
  && !cflow(within(kernel())) : around() {
    ...
  }
};
```

The `cflow()` pointcut function yields all code join points that occur while being in a given control flow. The `execution()` pointcut function yields all code join points, where a given function is about to be *executed*. The above pointcut expression therefore evaluates to any nonnested execution of a `kernel()` function. Compared to variant 2, this solution does not require weaving the client code and furthermore reliably detects indirectly nested kernel calls.

2.4 Join-Point Context

In many cases, advice for dynamic cross-cutting needs to read or modify the join-point–specific invocation context such as the actual argument values passed to the intercepted function. To fulfill the goal of quantification, join-point–specific context information has to be provided through a generic interface, as the same advice implementation should be applicable to many different join points, such as functions with different signatures. Most AOP languages provide a *join-point API* for this purpose. In AspectC++, the join-point API is implicitly available in advice bodies through the `JoinPoint *tjp` type and instance pointer:

```
aspect Tracing {
  ...
```

```
  advice execution("% ...::%(...)" && !"void ...::%(...)") : after() {
    JoinPoint::Result res = *tjp->result();
    cout << "leaving " << tjp->signature()
         << " returning" << res;
}};
```

The after-advice implementation of the above `Tracing` aspect is generic. It can be applied to any function with a nonvoid return type, as the join-point API provides the required abstractions from the actual return type.

2.5 Weaving

Aspect weaving is the term used to describe the process of transforming the structure or behavior of a program in order to let aspects “affect” other modules. The AspectC++ compiler weaves by transforming AspectC++ code into ordinary C++ code. It is a preprocessor that mainly generates transparent wrapper functions. This kind of weaving is called “static weaving” as it is performed at compile-time. “Dynamic weaving” is a different weaving approach that supports to weave aspect code into an already running program.

In this article we discuss both, static and dynamic weaving. In fact, our dynamic weaver reuses the tools for static weaving, which enables us to use the very same language for both flavors of weaving.

3 Adaptation Scenarios

Both, static and dynamic weaving, offer their own specific advantages. Supporting both for the same aspect language would increase usefulness and reusability of aspect code, as the same aspect can be used in very different scenarios. As a major advantage, dynamic weaving facilitates *in-vivo adaptation*, that is, the modification of a running program without having to stop it first. Typical application scenarios include (1) hot patching of, (2) policy optimization in, and (3) “on-demand” feature extension for long-running enterprise services [23,9]. Other suggested use cases are introspection and debugging of system software [36]. For “development aspects” significantly shorter compilation times are another major advantage of dynamic weaving. This facilitates short turn-around times for the step-wise refinement of tracing and debugging aspects.

Besides the fact that currently most “static” aspect languages offer significantly more language features than their dynamic counterparts, a major advantage of static weaving is *efficiency*. In a comparative study on Java-based dynamic weavers, Haupt and Mezini observed an advice invocation cost factor of up to 10,000 compared to a plain method call [16]. Even though the runtime overhead of C/C++-based approaches is lower [12,9,13,36] there probably always will be *some* overhead – as well as additional memory costs for the dynamic weaver runtime system. A static weaver, in contrast, can apply most AOP constructs absolutely cost neutral and overhead free [21].

Interestingly, many of the aforementioned use cases for dynamically woven aspects are actually *temporary* solutions. Typically, they have to be applied as dynamic aspects only until the system can be shut down and can then be deployed as static aspects. As we will show in the evaluation section, deploying aspects statically is generally preferable in order to avoid unnecessary overhead. Hence, in such cases, the combination of static and dynamic weaving would offer some noticeable advantages:

- A *hot patch* (1) can be applied as a dynamic aspect to all running instances of a service. Meanwhile, the very same patch can be applied as a (more efficient) static aspect to the service program, resulting in a new software binary that can be used if a new instance of the service is started.
- After the *policy aspect* (2) that performs best in a real-world load situation has been found, it can become the new default and be woven in statically for the next software release.
- If the software itself uses a concept of runtime-loadable modules, a *new feature* (3) can be applied as a dynamic aspect to all currently loaded modules of the service while being woven statically into those modules that are currently not loaded.

As these examples show, a combination of static and dynamic weaving offers the best of both worlds: while the extra flexibility of dynamic weaving is available at any time, its principle overhead would only apply as long as its principle advantages (run-time adaptation) are actually needed.

4 Analysis

In the following sections, we analyze some of the prerequisites and implications for the development of dynamic weavers that arise from a *combination* of static and dynamic weaving in the domain of statically typed and compiled languages. Our application perspective is the scenario of an adaptable software system, which, once deployed, is incrementally extended by aspects.

We understand an *adaptable (software) system* as a *base program* that can be modified or extended *after* its deployment time with *previously unknown* functionality by *adaptation modules*. The Base program and adaptation modules are binary modules, compiled from a set of classes or aspects. Technically, this can be understood as a process running the base program, in whose address space adaptation modules are loaded at runtime. "Previously unknown" means that neither nature nor structure of an actual adaptation module needs to be known when the base program is developed and compiled.

Knows-Relationship without AOP. With traditional modularization concepts, this is, however, not completely true for statically compiled languages that lack extensive support for reflection like C and C++. To provide the intended functional change, the adaptation module has to be explicitly *called* from the base program's control flows. Furthermore, it may have to perform *callbacks* into the base program. For this purpose, the base program typically defines an *adaptation contract* by a set of interfaces that can be used by adaptation modules. As a matter of fact, these interfaces, as well as all points in the control flow where adaptation may occur, had to be known at the compile time of the base program. Conceptually, there is some bi-directional *knows*-relationship between the base program and its adaptation modules (Figure 2.a).

Knows-Relationship with AOP. A frequently made observation (first published by COLYER, RASHID and BLAIR [7]) is that by aspects such bi-directional relationships can become uni-directional. By the AOP concept of *advice*, adaptation modules can integrate "themselves" into the base program's control flow, freeing the base program from the burden of specifying an adaptation interface and explicitly ensuring that potential adaptation modules are invoked from it's control flow. This is often referred to as the *obliviousness principle* of AOP [11] and considered as highly advantageous, as the (potential) adaptation points do not have to be known in advance. The result is an uni-directional *knows*-relationship from adaptation modules to the base system (Figure 2.b).

Knowledge hierarchy of Modules. The uni-directional *knows*-relationship facilitates *incremental adaptation*. By understanding an already extended base system as the new base system, *knows* becomes transitive. Further adaptation can be applied recursively, resulting in a *knowledge hierarchy* of adaptation modules with the base system as root and the latest adaptation module as leaf (Figure 2.c) .

Inter-Module Crosscutting. It is the nature of a crosscutting concern that it does not stop to cut across a system at module boundaries. Therefore, an aspect should

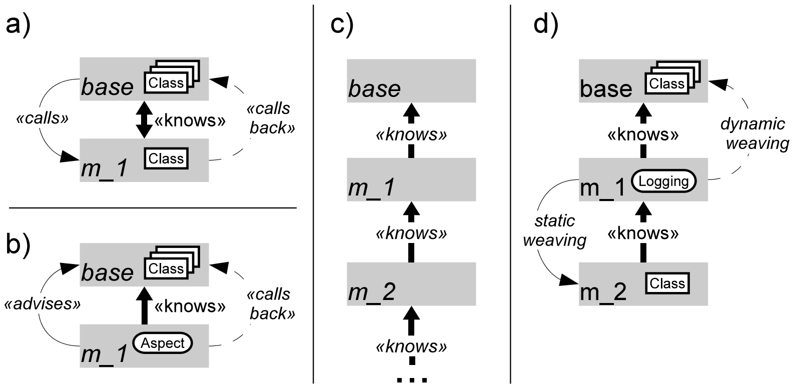


Figure 2. Nature of module relations in adaptable systems: **a)** traditional extensible systems **b)** AOP-based systems **c)** knowledge hierarchy **d)** up- and downward weaving

affect *all* matching join points, regardless whether the base or adaptation modules contain corresponding join-point shadows. For instance, a logging aspect in the base system that logs names and parameters of all performed file operations should not only affect the functions of the base system. It should also affect adaptation modules loaded later at runtime. Otherwise the aspect's output would be incomplete.

It is remarkable that, although they were unknown when the base program was developed and are loaded into the base system at run time, the logging aspect can be woven *statically* into the adaptation modules. The reason is that the adaptation modules are *further down* in the knowledge hierarchy and the logging aspect is *known* when the adaptation modules are being compiled. The only need for dynamic weaving arises when an adaptation module contains an aspect that affects join-point shadows within an already deployed part of the system, that is, join-point shadows within a module *further up* in the knowledge hierarchy. Consider the situation that the logging aspect is not part of the base system, but itself applied as an adaptation module. In this case it has to be woven dynamically into the base system, but can still be woven statically into all further adaptation modules. This is an example of a general rule: *upward weaving* within the knowledge hierarchy of modules has to be done dynamically, while *downward weaving* can be done statically (Figure 2.d).

4.2 Dynamic Weaving in Compiled Code

In this article we focus on aspect weaving in statically typed and compiled code, written in languages such as C, C++, or Ada. Compared to byte-code–based languages (such as Java), which are just-in-time compiled and executed by a virtual machine, these languages and their execution containers offer very poor support for run-time inspection and adaptation. This makes the implementation of dynamic weavers more challenging. Nevertheless, quite some work has already been conducted in this area (see Section 5), we therefore give here just a brief overview on the basic concepts of dynamic weaving in compiled code.

Weaver Binding. *Weaver binding* denotes how the advice code is actually bound to join points at run time and how the join-point shadows are retrieved [13]. The two general approaches used in the domain of compiled languages are *binary code patching* and *code instrumentation*.

Dynamic weavers that are based on *binary code patching* modify the machine code at run time to bind advice to specific join-point shadows. Join-point shadows and the actual weaving positions (e.g., of function calls) in the binary code are retrieved from linker symbol tables or debug information generated by the compiler. Literature shows that binary code patching can be very efficient with respect to runtime overhead of advice invocation. Arachne [9], for instance, binds around-advice to call join points by patching the matching function calls in the machine code, which results in very low overhead. The downside of binary code patching is that it requires structural information of the high-level language to be still present in the machine code. Therefore, compiler optimizations such as inlining have to be disabled, as an inlined function call is no longer available as a call join point in the machine code. While this is less of a problem with existing C code, modern C++ libraries (such as the C++ STL) heavily rely on function inlining to achieve a good performance and a small code size. Naturally, binary code patching is a highly platform-specific approach.

A platform-independent alternative is *code instrumentation*. In order to retain obliviousness this is done transparently, either on the source-code level by a pre-processor, or by the compiler itself. After the instrumentation, each potential join-point shadow provides a *hook* that can be used by dynamic aspects to register advice. These extra hooks, of course, induce some overhead. On the other hand, all join-point shadows from the high-level language are available and all compiler optimizations can be used. This is the approach that is used in our prototype implementation.

Run-time System. In both approaches a *run-time system* is needed to load and unload dynamic aspects at run time and to connect the advice code with the component code. Loading and unloading of aspects is typically realized by means of dynamic link libraries offered by the underlying operating system.

4.3 Challenges

From the viewpoint of dynamic weaving in a statically typed and compiled language, AOP features that either depend on join-point–specific static type information or that change the static structure of the base program are rather challenging. The *generic advice* feature, which is crucial for quantification, falls into the first category, while support for static crosscutting, namely *introductions*, falls into the second.

4.3.1 Generic Advice. To induce similar, but not identical effects on a set of related join points, the aspect language has to provide means to transparently adapt the advice behavior with respect to the actual join point it is invoked for. In AspectC++ *generic advice* [20] is used for this purpose. This type of advice uses static context information provided by the join-point API to instantiate the advice code at compile time with respect to the current join point:

```
aspect TraceResults {
  advice execution("% %(...)" && !"void %(...)") : after() {
    cout << tjp->signature() << "returns: " << *tjp->result() << endl;
} };
```

This simple aspect prints the values returned by all nonvoid functions from the global namespace. Even though simple, it already depends on generic advice. The join-point–API function `tjp->result()` retrieves a typed pointer to the return value with the actual static type *T* of the affected function. The compiler implicitly uses this information to find the best matching version of the stream operator << for type *T* during overload resolution. Developers can provide additional stream operators to support streaming of user-defined data types. Thereby, the advice is generic; it can print result values of *any* type for which a stream operator has been defined. If the compiler cannot find a suitable overload of the stream operator, a compile-time error is thrown.

Advice genericity is an important property of *generic aspect languages* [19]. Compared to run-time genericity based on reflection, which is commonly used in Java-based AOP approaches for similar purposes, generic advice has advantages with respect to type safety [22]. While this is nice, the point is that in a statically typed language, such as C++, there is *no alternative* to compile-time genericity. Reasonable support for run-time–type reflection or a uniform interface (such as `Object` in Java) that offers common functionality, such as `toString()`, is just not available. Note that this even holds with the C++ RTTI (run-time type information), which provides only very limited information and no polymorphic behavior. Even worse: RTTI is available only for class types that define virtual functions, but not for plain class types nor for the (still very common) C-style PODs (structs, arrays) and built-in types (`int`, `char`, `float`).

Hence, generic advice based on *static type information* is a crucial feature for dynamic aspect weavers in this domain. As this means that advice code has to be instantiated for each join point at compile-time, a dynamic weaver that implements generic advice requires access to all relevant type information. For our domain, we consider adding RTTI to all classes containing join-point shadows too expensive. We therefore focus on using type information at dynamic aspect compilation time rather than deployment time.

4.3.2 Introductions. By the concept of *introduction*, an aspect can extend existing types of the base program with additional elements, such as member functions,

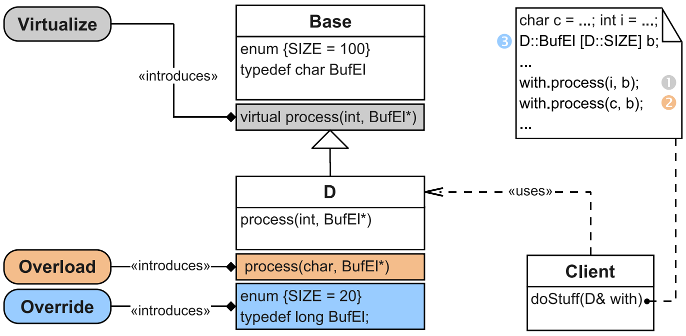


Figure 3. Introductions with language-level side effects.

attributes, inner types, and base classes. A static weaver usually merges the introduced code into all matching classes and thereby ensures that introduced elements become visible *before* the referencing code fragments are compiled. It is, however, impossible to modify a type *ex-post* in the binary code or at runtime, at least not in a feasible way for statically compiled languages, such as C/C++. The assumptions made by the compiler about internal layout and relationships of types are too deeply reflected in the generated machine code. Therefore, for a dynamic weaver, the goal cannot be to manipulate the target types at run time, but to achieve similar *semantics*. This means that *clients* of the affected class, which are aware of the dynamic introduction, shall be able to use the introduced element as if it was introduced statically. At the same time clients, which are not aware of the introduction, must not be broken, which means that the behavior of existing classes, methods, and members, must not change [1] . In the following, we analyze the semantic effects of introductions in AspectC++ and their consequences with respect to dynamic weaving:

Simple introductions. Many introductions have no semantic impact on existing clients of the affected class. On the language level newly introduced *nested classes*, *enums*, *typedefs, attributes*, or *member functions* are normally just ignored by existing clients. Only new clients that are aware of the aspect and the new elements can use them explicitly. Hence, in the vast majority of cases it should be possible to dynamically introduce elements into classes that are defined further up the knowledge hierarchy without affecting the behavior of their clients on the level of language semantics.

Introductions with Language-Level Side Effects. In C++, a method that has once been declared as `virtual` in the inheritance tree, remains virtual if overridden by derived classes, whether they declare it as virtual or not. By this mechanism, a virtual method introduced into some base class can implicitly “virtualize” existing methods of derived classes (Figure 3 case 1, aspect `Virtualize`). Other side effects are induced by the complex C++ name look-up rules. If an introduced element’s

---

[1] As described for generic advice, we consider invasive modifications to the base program like adding a vtable into all classes as not acceptable for our target domain.

identifier is not new, but *covers*, *overrides* or *overloads* an already existing and accessible identifier, the compiler might implicitly prefer the introduced version to the previous one. In Figure 3, the aspect `Overload` introduces a method `process()` for arguments of type `char` into class `D`, which overloads the already existing version for arguments of type `int`. As a side effect, this new version has now to be preferred whenever `process()` is called with a `char` argument (Figure 3 case 2), while previously the `int` version was used. Similar effects can happen, if an introduced element overrides identifiers imported from a base class (Figure 3 case 3, aspect `Override`).

Even though these scenarios can (arguably) be considered as "pathological", they are perfectly legal and part of the C++ language semantics. For dynamic weaving they are critical, because not only the object layout might be affected but also the behavior of the target class and its clients. In the case of a dynamic introduction, this may result in a different behavior of the running program than if the same introduction had been applyed by a statically woven aspect. In order to support semantic equivalence in such an adaptation scenario, the dynamic weaving infrastructure would need to replace previously running (and potentially even inlined) code and transform the internal program state to the new executable code. For our target domain, we have identified this as not feasible and, in fact, practically impossible.

The practical consequence for the development of dynamic aspects would be to *avoid* introductions that cause side effects on modules, which have already been deployed. It is possible to detect this reliably, as these modules are known when the aspect itself is compiled and developed.

Introductions with Code-Level Side Effects. Even if they do not cause semantic side effects on the language level, some *simple introductions* cause side effects on the machine code level. This is the case for all introductions that change the binary representation of objects and classes in memory. Examples are introductions of *nonstatic attributes* or *base class*es. The introduction of a *virtual member function* can also change the object layout, but only if the class does not already contain a virtual function. An additional virtual function may furthermore impact the internal representation of the class itself, specifically the layout of its vtable.

A dynamic weaver can hardly modify the internal binary representations of objects in the address space running modules further up in the knowledge hierarchy. Because of that, the binary representation must also not be modified for new modules further down the knowledge hierarchy, as we allow object instances to be passed between different modules and the binary representation must be identical everywhere. However the introduced element can only be referenced from modules further down the knowledge hierarchy; so it is possible to replace access operations transparently.

5 Related Work

The AOSD community for dynamic weaving has proposed many different approaches for dynamic weaving.Most of them target the domain of byte-code interpreted languages, namely Java. Much fewer have been suggested for compiled languages such as C or C++.

5.1 Dynamic Weaving Approaches for Java

Dynamic weaving approaches for Java can be roughly categorized in based on virtual machine extensions (PROSE [29], Steamloom [4], Axon [3]) and based on load-time or run-time bytecode manipulation, usually by exploiting Java Hotswap or some similar mechanism (JAC [28], Wool [30], JAsCo [34], AspectWerkz [5]). All Java-based approaches provide means to access the current join-point context via the Java reflection mechanism. This facilitates, from a pragmatic point of view, generic aspect implementations. [2] AspectWerkz and Wool furthermore support introductions, which are applied as *mixins* to the classes of the base program. After weaving, the introduced elements can be accessed by explicitly casting an object reference to the mixin interface. By the required explicit cast, AspectWerkz and Wool basically restrict introductions to what we called *simple introductions* in Section 4.3.2 and prevent the problems of side effects on the language level. Mixins have furthermore to conform with the constraints imposed by Java interfaces, which means that only new methods can be introduced. This additionally avoids the discussed side effects regarding the binary representation of object instances. Only AspectWerkz provides support for dynamic as well as static weaving.

5.2 Dynamic Weaving Approaches for C/C++

All approaches to support dynamic weaving in C are based on runtime binary code manipulation. *TinyC²* [37], *TOSKANA* [12], *KLASY* [36], and *Arachne* [9] are built on existing or homegrown code-instrumentation frameworks to rewrite the binary code at run time. The actual weaving positions in the binary code are examined with the help of symbol or debug information, generated by the C compiler during compilation of the targets. Hence, the general restrictions of binary code weaving discussed in Section 4.2 apply, even though KLASY overcomes parts of the information loss by using an extended C compiler. Their `gcc` generates additional symbol information and instruments the code to provide features that are unique in this domain, such as pointcuts on data member access and join-point context

---

[2] According to the definition by KNIESEL and RHO [19], reflection-based approaches do *not* qualify for a *generic aspect language*.

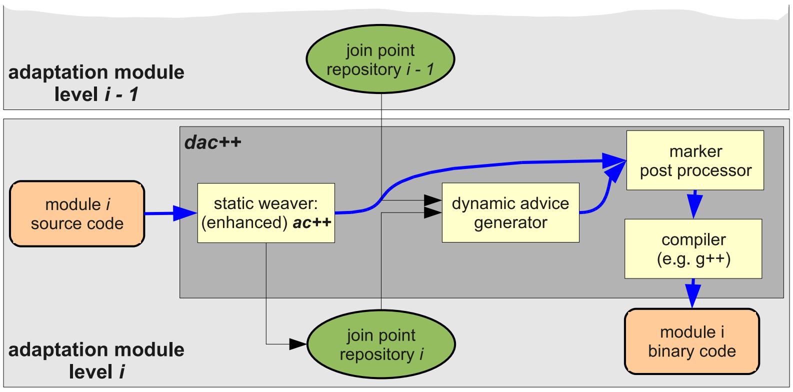


Figure 4. The structure of the dynamic weaving infrastructure

that includes values of local variables. Arachne specifically provides sophisticated means for control flow matching. Means for generic aspect implementations, support for dynamic introductions, as well as support for static *and* dynamic weaving are not provided by any of the existing weavers. Neither is support for inter-module crosscutting with yet to know modules.

6 Dynamic AspectC++

This section introduces the underlying concepts of the dynamic AspectC++ weaving infrastructure `dac++`. While earlier work on `dac++` focused on saving resources by tailoring the weaver's runtime system and exploiting *a-priori* knowledge about dynamic aspects [13], we here concentrate on the design implications of the analysis presented in Section 4.

6.1 Compilation of Adaptation Modules

During compilation of any adaptation module, two kinds of aspects have to be considered: *known* and *unknown aspects*. Known aspects are either defined by the module itself or by a module that was developed earlier. In the first case, aspects can be woven completely statically as in nonadaptable systems. The second case is trickier, as an aspect that is *known* does not necessarily have to be *loaded* already into the system. Hence, these aspects are woven statically, but can be dynamically turned on and off by the run-time system when the module that defines the aspect is dynamically loaded or unloaded.

Unknown aspects, that is, aspects that will be developed in the future, could affect any join-point shadow in the currently compiled module. Hence each module needs to be prepared for dynamic binding of advice by our infrastructure.

The main building block of the *dynamic* weaving infrastructure is a *static* weaver. Figure 4 shows the structure of `dac++` as well as its inputs and outputs while compiling an adaptation module on level i in the knowledge hierarchy. The static weaver is mainly needed to weave all known aspects statically. Besides this, it creates a *join-point repository* that describes the shadows of all potential join points that are located in modules, as well as the known aspects, pointcut definitions, and pieces of advice. The idea behind this repository is to provide sufficient information about join points in order to evaluate pointcut expressions without having to collect all necessary information by parsing the source code (again).

The join-point repository of the current level i as well as the repository of level $i-1$ are needed by the second `dac++` building block, which is the *dynamic advice generator*. It uses the repositories to find out, which piece of advice of the current module affects join points further up in the knowledge hierarchy. For these join points, advice has to be bound dynamically. As generic advice has to be *instantiated* for each target join point, it is the responsibility of the dynamic advice generator to provide the necessary type information about join points located further up in the hierarchy. (Section 6.3 describes the advice instantiation in more detail). It also generates the code that registers the dynamic advice instances with the runtime system at load-time of the module.

All transformed or generated files are then, once again, transformed by a *marker post processor*. Its purpose is to fix the code in cases where language features with side effects on the binary representation of classes were used. As described earlier in Section 4.3.2, critical code has to be replaced transparently. In order to avoid time-consuming reparsing, the static weaver is extended and now marks all critical operations. Based on this information and the join-point repository of the next upper level, the marker post processor can deal with binary code side effects.

6.2 Dynamic Weaving Approach

The two most important approaches for weaving in compiled code were already discussed in section 4.2. For `dac++` the code instrumentation approach is used. The source code is instrumented with an aspect, which uses regular pointcut expressions to define the joint-point shadows, that is, the joint points that are available for pointcuts of dynamic aspects.

The decision to use code instrumentation induces a little overhead in terms of code size. However, here we are exploring the expressiveness of aspect languages for dynamically woven code. Using code instrumentation guarantees that the dynamic weaver can offer the same pointcut expressiveness as the static weaver. Technically, the instrumentation aspect introduces a function pointer for each potential join point. The actual weaving and unweaving of dynamic aspects is implemented by a module loading and unloading mechanism combined with a run-time system that manipulates the function pointers. As the "hooks" are effectively implemented by checking a function pointer, this ensures that the overhead is kept at a minimum.

One issue with this approach is that the number of join-point shadows in "known" modules is finite and fixed; new join-point shadows can only be introduced by new extension modules. This means that the decision on the amount of join-point shadows to be instrumented directly affects the later adaptability of the system. At instrumentation time, a tradeoff must be made between instrumenting all join-point shadows – which would bring maximal flexibility, but also induce high run-time costs – and too few shadows. In our experience, a good compromise is to instrument all execution join-point shadows. Their number is relatively small (it is bounded by the number of functions), so the run-time costs are tolerable. On the other hand, this gives enough control to hook into all run-time control flows. Even if an (uninstrumented) call join-point shadow was necessary, one can always fall back to replacing the complete function that contains the call with an instrumented version – by giving around advice to its execution join point.

6.3 Generic Advice

Advice Instantiation in the Static Case. An example for generic advice has already been presented in Section 4.3.1. In order to instantiate the advice for each join-point shadow, the static weaver for AspectC++ transforms generic advice into a C++ template member function of the aspect, which itself is transformed into an ordinary C++ class [32]. The instantiation is triggered by wrapper code that is inserted at a specific join-point shadow. To provide the necessary type information for the advice, a join-point–specific class is generated that contains the necessary type information as C++ typedefs. This `JoinPoint` class is used as a template parameter in the advice function call [20]. When the C++ compiler translates a template function call, it instantiates the function (the advice in our case) if it has not been instantiated already.

Advice Instantiation in the Dynamic Case. For join points that are located further up in the knowledge hierarchy there is no such wrapper function that could instantiate the advice. Instead of this, the run-time system allows to register a function to be called when a specific dynamic join point is reached. Of course, the run-time system can neither provide static type information nor can it instantiate the template function at run time. Hence, the adaptation module instantiates the advice for these join points itself. This task is performed by the dynamic advice generator (Figure 4). It generates structures with typedefs similar to the `JoinPoint` classes in the static case. The necessary type names are found in the join-point repository of the next upper level. Furthermore, the repository provides the names of all source files that actually define the needed types. If the types are defined in header files, the generator simply includes the definitions in the generated advice instantiation module.

Dynamic `JoinPoint` classes alone do not instantiate advice code. Some additional wrapper function is needed that calls the advice template functions and uses the corresponding dynamic `JoinPoint` class as a template parameter. These wrapper functions are registered with the run-time system when the module is loaded.

By this mechanism, dynamic and static advice code is transformed in an identical manner. Thus, the same static aspect weaver can be employed. In both cases the advice function template is parameterized with a `JoinPoint` type for static type information and compile-time constants (such as the number of arguments). Run-time–context information is passed similarly. In AspectC++, each advice expects a parameter `JoinPoint *tjp` that is used to access the run-time context. In the case of dynamic weaving, the generated wrapper functions not only provide the typedefs in the `JoinPoint` type, but also the requested run-time context via the same interface as in the static case.

6.4 Introductions

Many dynamic introductions can be woven almost "out of the box" with the weaving infrastructure sketched so far. Only introductions that affect the binary compatibility need to be treated with special care.

Simple Introductions. Simple introductions are woven statically into the module that contains the introducing aspect and all modules further down in the knowledge hierarchy. Access to the introduced elements can be performed without overhead. Although the C++ compiler "sees" a different target class definition, when it compiles a module that does not know this aspect, the binary code will still work, because the binary compatibility is not affected.

Introductions with Language-Level Side Effects. The dynamic weaving infrastructure is required to detect introductions with side effects on the language level at compile time. This is not only a matter of detecting used language features in the aspect definition, but also depends on characteristics of the target join points. For example, the introduction of a virtual function does not have any side effects on the language level as long as it does not "virtualize" another function in a derived class, as shown earlier in Figure 3. Although not trivial, this static analysis is feasible, because `dac++` can use the join-point repository, which contains the required structural information about the target component code.

Introductions with Code-Level Side Effects. Introductions that affect the object or class layout are more complicated. Examples are introductions of new non-static attributes, base classes, and virtual member functions. Our approach to cope with these cases is to manipulate all operations that depend on the modified structure. Such operations can only exist in adaptation modules that *know* the introducing aspect.

For dynamic attribute introductions, the run-time system manages the storage for introduced elements. It furthermore provides means to map an object address to the data structure that holds these elements. If the run-time system is asked for that address instead of accessing the object directly, the object layout is modified transparently.

In `dac++` the transformation of the access sites is performed by the marker post processor, based on marks that are inserted by an extended static aspect weaver. The post processor furthermore ensures that the binary compatibility is preserved in all modules. As the aspect weaver simply introduces new attributes as ordinary members into target classes, the post processor has to remove these attribute declarations if the target class was also known by modules further up in the hierarchy. This means that the post processor needs the join-point repository of this layer.

7 Implementation

As a proof of concept, the `dac++` design sketched in the previous section has been implemented and is available at `http://dynamic.aspectc.org`. This section describes the most interesting “aspects” of the implementation.

7.1 Join-Point Repository

The following listing is an excerpt from a join-point repository, as it is generated by our static weaver `ac++`:

```
<files>
  <header id="117" name="HttpHeader.h" len="266" .../> ...
</files>
<namespace id="0" sig="::"> ...
  <class id="166" sig="HttpHeader"> ...
    <function id="572"
     sig="void HttpHeader::append(const HttpHeader *)">
      <src file="401" line="419" len="13" kind="def"/>
      <src file="117" line="201" len="1" kind="decl"/>
      <exec id="73"/>
    </function>
  </class>
</namespace>
```

The repository is an XML document that describes all known join-point shadows. This includes all functions, classes, and namespaces, which are regarded as (name) join points in the AspectC++ join-point model. In this example, a function `HttpHeader::append()` is listed. With `<exec id="73">` `ac++` marks this function as shadow of an execution join point. The pointcut evaluation mechanism of `dac++` is solely based on this information. Moreover, other applications, like editors and development environments, could make use of the information from this repository as well.

The `<src file ="id"...>` tags describe the locations in the source code where the function is defined or declared. By looking up the file ID in the file table at the beginning of the repository, it is possible to identify the file that has to be consulted for the static type information for a particular join point.

7.2 Generic Advice

The key to support generic advice is the instantiation of advice code with the proper static type information as a template parameter. As described earlier, the dynamic advice generator produces code that is responsible for this instantiation. The following listing shows an excerpt of the generated code as an example:

```
#include "HttpHeader.h"
...
#include "DynamicContext.h"
struct StaticContext_73_0 : public DynamicContext {
  typedef void Result;
  static const int JPID = 73;
  static const AC::JPType JPTYPE = (AC::JPType)8;
  enum { ARGS = 1 };
  static unsigned int args() { return ARGS; };
  template <int I, int DUMMY = 0> struct Arg {
    typedef void Type;
    typedef void ReferredType;
  };
  template <int DUMMY> struct Arg<0, DUMMY> {
    typedef const HttpHeader *Type;
    typedef HttpHeader *ReferredType;
  };
  using DynamicContext::arg;
  template <int I> typename Arg<I>::ReferredType *arg () {
    return (typename Arg<I>::ReferredType*)arg (I);
  }
  static const char *signature () {
    return "void HttpHeader::append(const HttpHeader *)";
} };
```

For each dynamically affected join point, a C++ `struct` named `StaticContext_<jpid>_<modid>` is generated. *`jpid`* and *`modid`* are unique numbers that represent the currently compiled module and affected join-point shadow. The base class `DynamicContext` does not depend on the join point. It defines the amount of dynamic context information that is passed from the run-time system to the advice code. In this example the affected join point is, again, the execution of the function `HttpHeader::append()`. The static information about this function contains the result type, the join-point ID, the join-point type (e.g., execution or call), the number and types of arguments, and the function's signature as a string. The template member function `arg<i>()` provides the advice code with a mechanism to access the function's argument value at run time in a type-safe way.

All types used in this generated `struct` would be meaningless without the `#include "HttpHeader.h"` directive at the beginning of the listing. The generator can retrieve this file name by following the file ID in the join-point repository (as described before). The advice instantiation itself is triggered by the following wrapper function, which is also generated by the dynamic advice generator:

```
void __dacwrapper_1_DynamicTracer_a0_before(void *djp) {
  typedef StaticContext_73_0 DJP;
  Tracer::aspectof()->__a0_before<DJP>((DJP*) djp);
}
```

This wrapper function is then registered with the run-time system after the module has been loaded. `Tracer` is the name of the aspect that contains the advice definition. The member function `aspectof` yields a pointer to the aspect instance on which the advice shall be invoked. As `__a0_before` is the internal name of the advice code, which is transformed into a template function, this function call in fact instantiates advice for a particular join point and provides the static information needed by generic implementations.

7.3 Introductions

In order to efficiently map objects to their dynamically introduced members, we decided to statically introduce a single pointer in every class that is supposed to be a target of dynamic introductions. This is done by our static instrumentation aspect (see section 6.2) by a combination of a static introduction and construction advice for the initialization of the pointer. An adaptation module now attaches a data structure that contains all introduced elements, a module ID, and a pointer to further introductions to any target object.

The static weaver `ac++` has been extended to mark[3] all introduced attributes as well as operations that access these attributes, such as `expr.attr`, `obj->attr`, or only `attr`. Based on this information, the marker post processor generates a class definition per module/target class combination, which contains all attribute declarations. Additionally, the marked attribute accesses are replaced by a call to a generic run-time–system function that looks up the object's introduction chain

---

[3] in form of source code annotations

for an entry with the respective module ID. If the object has not been extended yet, an instance of the class with the new attributes will be constructed on demand and appended.

8 Case Studies

In this section we present the results and experiences we made while applying our dynamic weaving infrastructure to third party projects in order to demonstrate the effectiveness and feasibility of our approach. In the first case study, the proxy server Squid is introduced and prepared for static adaptation. This enables us to develop dynamic aspects that support an incremental analysis of the dynamic behavior of Squid, and eventually create a hot patch that can be applied without having to restart the service. The second study illustrates the economical resource consumption of our approach and, thus, its applicability to even embedded systems using the example of eCos. eCos is a well known operating system and widespread in the area of embedded systems. We show how to improve the static and run-time adaptability of eCos by the application of static and dynamic aspect weaving.

8.1 Case Study 1: Weaving in Squid at Any Time

Squid is a widely used web server proxy and well known as an example for dynamic aspect weaving in C code [8]. While earlier versions of Squid were implemented in C, the latest version 3.X has an object-oriented design and is implemented in C++. It is a typical long-running application and, thus, well suited to show that the scenarios envisioned in section 3 can be put into practice with the tool chain presented in this article.

8.1.1 Preparation of Squid. A prerequisite for dynamic weaving into Squid is the code instrumentation, for which we use a configurable static aspect. In this example we decided to instrument all join-point shadows of execution join points: 3099 functions. Due to the selected instrumentation and the run-time system, the code size is increased from 1.73 MB to 1.88 MB.

8.1.2 Generic Tracing. Based on this version of Squid, we can now deploy aspects written in AspectC++ at run time. For example, we implemented a simple tracing aspect for all join points. While weaving the same tracing aspect statically would take as long as completely compiling Squid with `ac++` (about 17 minutes), the compilation of the dynamic version takes only about 5 seconds. This makes it very convenient to modify and recompile tracing aspects, when more join-point context should be printed or only specific functions are relevant.

Now imagine that we use Squid for web page caching in our company. One day a user complains that he has problems to download files. While Squid is still running, we decide to implement a tracing aspect that monitors all functions that deal with the exchange of HTTP messages on a very detailed level:

```
aspect HTTPTracer {
  advice execution("% ...::Http%::%(...)") : before() {
    cout << "trace: " << JoinPoint::signature() << endl;
    ArgPrinter<JP::ARGS>::work (tjp);
} };
```

This aspect matches 165 of the 3099 instrumented dynamic join points. It prints all arguments of the traced functions. For this purpose, a template meta-program `ArgPrinter` has to be used, which iterates over all arguments at compile-time and thereby generates a sequence of calls to the stream operator << with the actual argument types. An example for a similar compile-time loop over all function arguments can be found in [22].

Our run-time system is informed about the new dynamic aspect by sending it a process signal. It then loads the tracing aspect and we can immediately watch the HTTP protocol related control flow. When the user repeats the malfunctioning operation, we can see the following output:

```
trace: void HttpRequest::initHTTP(_method_t,proto...
Arg 1: 1
Arg 2: 1
Arg 3: /releases/edgy/beta/ubuntu-6.10-beta-dvd-i386.iso
trace: int HttpRequest::parseHeader(const char *)
Arg 1: Range: bytes=17904205-
User-Agent: Wget/1.10.2
Accept: */*
Host: cdimages.ubuntu.com
```

The output tells us that the user accesses Squid with the `wget` program, which issues a “range request” for loading a partial file. It turns out that this particular version of `wget` contains a bug in the code that handles our reply on the range request.[4]

8.1.3 A Dynamic Hot Patch. After localizing the problem we can now use a dynamic aspect to fix the problem without having to stop the program. The following aspect does the job (match expressions are truncated):

```
aspect CheckForBrokenWget {
  advice "HttpRequest" : slice class {
    bool _clBroken;
  public:
    bool clientIsBroken() const { return _clBroken; }
    void clientIsBroken(const char *s) {
       _clBroken = strstr(s, "Wget/1.10.2");
  } };

  advice execution("% HttpRequest::parseHeader...") :
  after() {
```

---

[4] In fact wget 1.10.2 works fine. This is a hypothetical scenario.

```
      tjp->that()->clientIsBroken(*tjp->arg<0>());
    }
    advice execution("bool HttpStateData::decide...") :
    after() {
      HttpRequest *request = *tjp->arg<0>();
      if (request->clientIsBroken())
        *tjp->result() = false;
  } };
```

The first part consists of a slice introduction, which was introduced in Section 2.2. Here it contains a boolean attribute, a function that checks for the name and version of the buggy client and sets the attribute accordingly, and a function to read the flag. From our source code and tracing output studies we know that the control reaches `HttpRequest::parseHeader()`, whenever an HTTP message is received. By calling the introduced method `HttpRequest::clientIsBroken()` we check whether this message comes from a buggy client. Later on in the control flow, Squid has to decide whether the range request should be handled. This is done by the function, which is affected by the second piece of advice. It checks if our introduced flag is true and manipulates the result value of the decision function accordingly. This fixes the problem, because client and server then use an ordinary transfer mode.

After testing the patch with a separate instance of Squid, it can be deployed dynamically. During the whole process our production system never had to be stopped. We can now weave the same aspect statically into the Squid source code in order to get an improved version that implicitly contains the fix.

8.1.4 Performance and Code Size. An important question for the applicability of the approach is whether the performance impact of instrumentation is acceptable, that is, how much one has to pay for the ability to apply patches at run time. We retrieved this cost factor by comparing the throughput (requests per second) of the standard version and the fully instrumented version of Squid[5]. The following table lists the results:[6]

module	localhost [req/s]	remote [req/s]
squid	*3044*	*1353*
squid-instrumented	*2834*	*1338*

When measuring the overhead by running the benchmark application on the same host, the actual overhead is shown precisely. However, this is not entirely realistic, as this setting ignores the network induced jitter (e.g., by routers, switches, etc.) for the incoming requests and sent replies. For this reason, this evaluation measures both cases. In a localhost access scenario, the instrumentation causes a performance

---

[5] The latter without any further adaptation modules loaded

[6] Measurements taken on an 2.4 GHz Intel Core2 Quad (Q6600) running Apache Benchmark (ab) under Ubuntu Linux 8.04.1 (kernel 2.6.24) on the same machine (localhost), respectively over switched ethernet (remote). Values are averaged over 500,000 requests. All code (squid-3.0.PRE4, ab, aspects) was compiled with g++ 4.1.2 -O2.

loss of seven percent (3044 versus 2834 connections/s). In the more realistic remote access scenario, however, the difference drops to one percent (1353 versus 1338 connections/s). We consider this overhead as acceptable for the gained flexibility.

As mentioned earlier, the code size of Squid was increased from 1.73 to 1.88 MB (8 percent) due to the instrumentation of 3099 static join points and the run-time system. Besides Squid itself, also the dynamically loaded modules contribute to the overall code size in the instrumented version. The following table shows the static memory requirements:

module	text	data	bss	total [byte]
squid	*1,110,997*	*4828*	*61,1636*	*1,727,461*
squid-instrumented	*1,259,692*	*4860*	*61,6340*	*1,880,892*
HttpTracer	*110,734*	*268*	*736*	*111,738*
CheckForBrokenWGet	*4559*	*276*	*68*	*4903*

The `HttpTracer` module is with a total of 112 KB much bigger than our hot patch, which takes only 5 KB. The reason is that it affects 165 join points: The tracing advice, which contains relatively expensive streaming code, has to be instantiated for each of these points. Additionally, the static context information for each join point contains the executed function's signature as a string. The patch on the other hand has an almost negligible code size. Here only two join points are affected.

8.2 Case Study 2: Static and Dynamic Adaptaion of the eCos Operating System

eCos is a small and highly configurable operating system targeting the market of embedded systems. It is available for a broad variety of 16 and 32 bit microprocessor architectures (PPC, x86, H8/300, ARM7, ARM9, ...) and used in many different application domains (MP3 player, digital cameras, printers, routers, ...). The eCos system itself is provided as a congregation of various components, which are configured *statically* with a configuration tool called *eCosConfig*. The components are implemented in a mixture of C++, C, C-preprocessor macros and assembly code. After the user selects an appropriate eCos configuration within *eCosConfig*, a configuration-specific system of headers and makefiles is generated, which is used to build the *eCos-library*. The final applications are linked against this library.

8.2.1 Analysis. In the context of this case study, we analyzed several parts of the eCos system (kernel, C library, POSIX subsystem, μITRON subsystem, Memory Management, Wallclock Driver, and Watchdog Driver) with respect to their adaptability. In this case study we will exemplarily concentrate on the eCos kernel.

For system software clean encapsulation of the different features is crucial in order to be adaptable. Therefore, our first goal was to figure out the positions and the amount of code that implements highly crosscutting concerns and locally crosscutting optional features. The analysis revealed that 20.54% of the kernel source code is needed to implement four highly crosscutting concerns (CCCs): *Tracing*, *Assertion*, and *Kernel Instrumentation* (profiling) for development support and *Interrupt Synchronization*. The following table (column "original") presents the numbers for each of these concerns individually. Actually, these figures only reflect the number of call sites activating these crosscutting concerns, the functional parts of their implementations were not taken into account here.

	original		**aspectized**	
	LOC	**%**	***LOC***	**%**
CCC Code	*1069*	*20.54 %*	*290*	*6.41 %*
Component Code	*4136*	*79.46 %*	*4237*	*93.59 %*
Total	*5205*	*100.00 %*	*4527*	*100.00 %*

The results of the analysis show that eCos indeed is configurable to a great extent, but certainly lacks adaptability. The high portion of crosscutting concerns and the amount of scattered configuration options in the eCos kernel indicate that complex correlations between different features exist on the level of the implementation. These correlations make it very hard to omit certain features or add new ones. In other words, these correlations hamper the adaptation of the eCos kernel.

8.2.2 Static Adaptability. During the case study, we enhanced the adaptability of eCos by "aspectizing" the highly crosscutting concerns and crosscutting optional features mentioned in the previous section. The necessary refactoring of the source code was straight forward, as the affected code was easy to spot. Highly crosscutting concerns such as *Tracing* are realized as macros to avoid code redundancy. Optional feature implementations are bracketed by preprocessor directives for conditional compilation.

The refactored code was also analyzed and the results are shown in the table below. This analysis is a refinement of the one presented above with respect to the crosscutting concerns, only. These results clearly illustrate, that most of the crosscutting concerns and optional features could be modularized very well by aspects. However, we were not able to modularize assertions, due to their individual semantic, and features implemented in C, as our aspect weaver is not capable of weaving in pure C code.

	original	**aspectized**
Tracing	*336*	*4*
Assertions	*384*	*286*
Kernel Instrumentation	*162*	*0*
Interrupt Synchronization	*187*	*0*
Total	*1069*	*290*

8.2.3 Dynamic Adaptation. The Mars Pathfinder mission launched in 1996 is one of the most well-known space missions of the foregoing decade. On the one side, because it was the first mission to Mars that included a rover (robotic exploration vehicle). On the other side, because of the problems experienced during this mission [35]. After a few days of successful operation the spacecraft experienced total system resets and each of these resets caused a loss of valuable meteorological data.

The absence of the tracing facility on the spacecraft forced the engineers to spend hours running the system on the exact spacecraft replica in their lab with tracing turned on, in an attempt to replicate the precise conditions under which they believed that the reset occurred. The traces finally revealed the priority inversion scenario. The problem was that while a low and a high priority task were competing for the same mutex, a middle priority task preempted the low priority task holding the mutex and, thus, prevented it from unlocking the mutex. The high priority task, thereby, was delayed too long and missed its deadline. This in turn, caused a watchdog to go off and reset the whole system. While such a scenario does not cause too much trouble in normal computing systems it is a serious problem in a real-time computing systems and known as uncontrolled priority inversion. Mutexes in VXWorks (the operating system used for this mission) could either be equipped with the priority inheritance protocol or not. Initially the mutex entailing the priority inversion was configured not to use the priority inheritance protocol. A C-interpreter, embedded into the computing system on the spacecraft, helped to fix the problem by uploading a C-program to the spacecraft with the purpose to enable the priority inheritance protocol for the particular mutex. From this point on, no priority inversion occurred any more. The problem was solved and the mission could be finished successfully.

8.2.4 Motivation. Both the tracing facility and the C-interpreter were absolutely crucial to solve the problem. However, the absence of the tracing facility in the actual system made it extremely hard and time consuming to locate the problem. Additionally, the support for the priority inheritance protocol was statically embedded in the computing system of the spacecraft, but what would have happened if it was not? Or if the C-interpreter was not a part of the computing system due to memory restrictions? The problem would have been unsolvable, the mission would have failed!

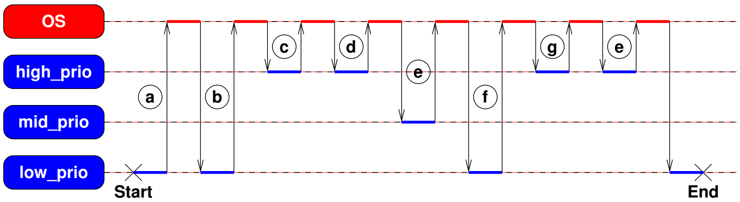
Furthermore, one should keep in mind that the scenario described above can not only be caused by design faults, but also in the context of run-time adaptation. Consider you want to extend the functionality of a running system. Therefore, it might be necessary that additional threads have to be added which also have to lock a specific mutex. In such a scenario the conditions that enable priority inversion can easily be fulfilled by accident.

An alternative solution for such problems is provided by dynamic aspect weaving. Tracing and the priority inheritance protocol, both implemented as dynamic aspects, could then be uploaded to the spacecraft and woven into the running system. There is no need to embed the priority inheritance protocol from the very beginning anymore; it would be loadable on demand. It would not be necessary to have a fully developed C-interpreter, only an infrastructure is needed that allows to weave aspects during runtime. In a former case study [31] we have already shown that tracing could be implemented by a dynamic aspect without suffering significant overhead in comparison to a static tracing aspect. Here we demonstrate that the eCos' priority inheritance protocol could also be implemented as dynamic aspect without having to put up with unacceptable overhead in comparison to static aspects.

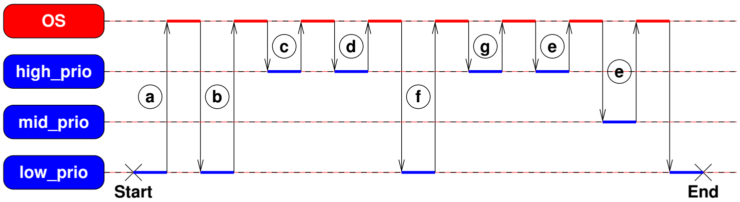
8.2.5 Implementation. We already re-factored eCos' priority inheritance protocol into a static aspect in previous work [21]. In the priority inheritance implementation of eCos the owner of a mutex inherits the priority of a thread trying to lock the same mutex and, thus, blocks. The owner's priority is set back to its original priority when it has unlocked all mutexes it owns, therefore, the count of mutexes locked by one thread has to be tracked. This variant of the priority inheritance protocol induces slightly longer blocking times when a thread holds more than one mutex, but simplifies the implementation a lot. The implementation as static aspect gives advice on the construction of a thread to initialize the number of mutexes locked and to the methods `mutex_lock()`, `mutex_unlock()` and `mutex_trylock()` of the mutex class to update the count of locked mutexes. Call advice on the activation site of the scheduler within method `mutex_lock()` transfers the priority of the blocking thread to the owner of the mutex while execution advice on the method `mutex_unlock()` checks whether all mutexes are unlocked again and the owner's original priority has to be restored.

The conversion from the static aspect to a dynamic version was very straightforward and demanded virtually no manual intervention. The dynamic advice transferring the blocking thread's priority to the owner of the mutex is shown below:

```
advice call("% Cyg_Scheduler::reschedule(...)")
     && within("% Cyg_Mutex::lock_inner(...)")
      : after() {
  Cyg_Thread self = Cyg_Thread::self();
  inherit_priority(tjp->that()->owner,self);
}
```



Without Priority Inheritance Protocol



With Priority Inheritance Protocol

Figure 5. Execution sequence without and with priority inheritance protocol.

	system call	description
a	`mutex_lock(&mutex)`	*successfully lock mutex*
b	`thread_resume(high_prio)`	*activate thread high_prio, a context switch occurs*
c	`thread_resume(mid_prio)`	*activate thread mid_prio, no context switch occurs as mid_prio's priority is lower than high_prio's priority*
d	`mutex_lock(&mutex)`	*try to lock the mutex, as it has already been locked by low_prio, high_prio blocks*
e	`thread_exit()`	*the current thread finishes execution, a context switch occurs*
f	`mutex_unlock(&mutex)`	*thread low_prio unlocks the mutex, a context switch occurs as a thread with a higher priority is already awaiting the allocation of the mutex*
g	`mutex_unlock(&mutex)`	*thread high_prio unlocks the mutex*

Table 1
System calls used in the test application

8.2.6 Evaluation . In order to evaluate our implementation we implemented a small, synthetic eCos test application leading to a priority inversion scenario. At first, this scenario was executed with no priority inheritance protocol present. Then, the dynamic priority inheritance protocol aspect was woven into the system and the same scenario was executed again. The exact execution sequence of both scenarios is depicted in Figure 5, the system calls used at each step of the execution sequence can be obtained from Table 1.

The test application was then linked against four different variants of eCos. Two variants contained support for the weaving of dynamic aspects. In the first of those two variants (variant *dynamic (perfect)*) only these join points needed to weave the dynamic priority inheritance aspect are hooked. This variant illustrates the overhead of the dynamic aspect itself. The second variant (variant *dynamic (flexible)*) hooks all methods of the classes `Cyg_Thread` and `Cyg_Mutex` for dynamic execution join points and all call sites within these classes for dynamic call join points. This variant also would allow to implement other synchronization mechanisms that affect more join points and illustrates the price one has to pay for dynamic adaptation. The other variants use static aspects (variant *static*), only, and either contain the priority inheritance protocol or not.

The test application and the eCos operating system were compiled and linked using the GNU compiler collection and the GNU bintutils[7]. The testcase scenario was executed on a Pentium III (1 GHz) with caches turned on. The binary was downloaded onto the target machine using eCos Redboot[8] and gdb via the serial line and the gdb remote protocol. The memory consumption of the eCos kernel was determined by analysing the memory map file generated by the GNU linker. For run time measurements the test application was executed for 4000 times and the average values of all these measurements obtained by the pentium's `rdtsc` instruction were computed.

The analysis of the memory consumption of the different variants of the test application is mainly restricted to the eCos kernel, the priority inheritance aspect and the dynamic weaver infrastructure. The results of the analysis are shown in Table 2. *Kernel* subsumes the total memory consumption of the eCos kernel, *Priority Inh.* and *Weaver* refer to the memory consumption of the dynamic or the static aspect and the dynamic weaver infrastructure and are already contained in the kernel's memory demand. Column *Total* shows the memory consumption of the complete test application. For a perfect hooking (variant *dynamic (perfect)*) the memory overhead within the eCos kernel is very low, only 144 bytes of RAM and about 1.5 KB of ROM plus 52 bytes of ROM for the dynamic weaver infrastructure are additionally needed in comparison to the variant employing static aspects only (variant *static (priority inh.)*). As soon as more join points are hooked (variant *dynamic (flexible)*), the memory requirements are noticeably increased by the dynamic weaver infrastructure, extra 628 Bytes of RAM and about 8 KB of ROM are needed in comparison to variant *static (priority inh.)*. Keeping in mind that the complete test application consumes about 26 KB of RAM and between 18 KB and 27 KB of ROM, this is still a price that is affordable and should be definitely cheaper than embedding a fully developed C-interpreter. There is no RAM and only very little ROM consumption declared for the dynamic weaver infrastructure, because a direct consequence of our dynamic weaver implementation is that the memory overhead caused by join point monitors is spread over the whole system and is already contained by the RAM and ROM demand of the kernel. The memory demand of the dynamic priority inheritance aspect looks quite large in contrast to the static aspect. This is because the static aspect uses introductions a lot, thus, this memory demand is assigned to the kernel itself, while the memory demand for the introductions of a dynamic

---

[7] gcc version 4.03, binutils version 2.16.1

[8] the boot loader provided along with eCos

	Kernel		**Priority Inh.**		**Weaver**	**Total**	
	RAM	***ROM***	***RAM***	***ROM***	***ROM***	***RAM***	***ROM***
dynamic (flexible)	*2834*	*13478*	*168*	*2562*	*52*	*27177*	*27738*
dynamic (perfect)	*2350*	*6800*	*136*	*1554*	*52*	*26721*	*21130*
static (priority inh.)	*2206*	*5375*	*0*	*77*	*0*	*26495*	*18325*
static (no priority inh.)	*2194*	*4427*	*0*	*0*	*0*	*26445*	*17305*

Table 2
Memory consumption of the different eCos variants measured in bytes.

aspects are fulfilled by the aspect itself.

For the assessment of the run-time overhead imposed by the dynamic aspect and the dynamic weaver infrastructure we measured the execution time of the methods that are affected most by the priority inheritance protocol: these are `mutex_lock()` and `mutex_unlock(),` each with and without a subsequent context switch (refer to a,d,f,g in Table 1 and Figure 5). The results of these measurements are shown in Figure 6. These results confirm the results of the memory measurement. Variant *dynamic (perfect)* only shows minimal decline of run-time performance in contrast to variant *static*, that is, the run-time cost of one hook and the dynamic aspect is quite small in comparison to the static aspect. As soon as more join points are hooked (variant *dynamic (flexible)*) the run-time overhead increases and reaches a factor up to about two (mutex_lock (d), priority inheritance protocol enabled). The only figure not fostering this observation is the execution time of `mutex_unlock()` when no context switch follows and the priority inheritance protocol is enabled. Here the variant hooking more join points (*dynamic (flexible)*, 391 clock cycles) is faster than the variant that only hooks those join points that are really needed (*dynamic (perfect)*, 440 clock cycles). Actually, this system call even executes faster with the dynamic aspect woven (with priority inheritance protocol) than without the dynamic aspect (without priority inheritance protocol, 398 clock cylces). There are some explanations possible: caching effects, code alignment, DRAM refresh cycles, etc., but it is nearly impossible to identify the one of them that really causes the different execution times. The only thing that is almost sure is that there should be no relation to the code of the dynamic weaver infrastructure. In variant *dynamic (perfect)* the dynamic weaver infrastructure is activated twice during this system call, while it is activated for six times in variant *dynamic (flexible)*. The rest of this system call and the code of the dynamic weaver infrastructure are identical for both versions.

9 Discussion

Our case studies show that for many concerns in both system software like networking services and deeply embedded system software, aspect-oriented implementations

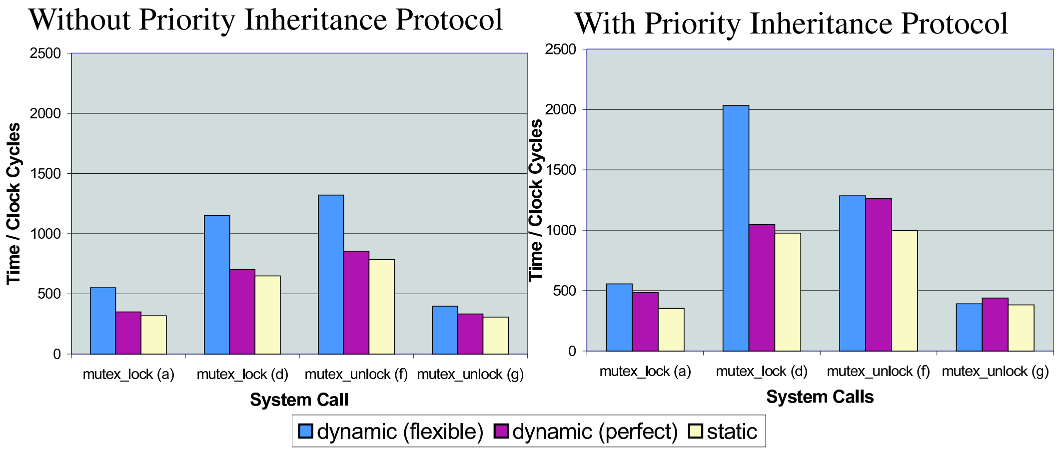


Figure 6. Run-time performance comparison of different eCos variants.

and especially dynamically woven aspects are affordable. By the refactoring and the integration of the dynamic weaver infrastructure into the proxy server Squid and the operating system eCos we have shown that dynamic deployment of adaptation modules is both affordable and feasible. These systems now offer both improved static as well as dynamic run-time adaptability: Better static adaptability, because crosscutting concerns and crosscutting optional features are now cleanly modularized and encapsulated, and better run-time adaptability, because it is now possible to adapt to changing requirements at run time. However, the case studies also illustrate that dynamic adaptation is not for free, especially when many join points have to be instrumented the overhead increases sensibly.

Our approach is conceptually and technically based on two fundamental observations:

- Modules in an AOP-based adaptable system constitute a *knowledge hierarchy*.
- Run time weaving of a dynamic aspect is required *only upwards* the knowledge hierarchy. Downwards the hierarchy, static weaving can be used instead.

Because of these observations statically *and* dynamically woven aspects can use static type information when accessing join-point–specific context. This is the prerequisite for the major advantages: the support for generic aspect implementations by means of *generic advice* and the support of static crosscutting (structural modifications by means of *introductions*).

Generic Advice and Introductions. Important AOP features for generic aspect implementations and static cross-cutting have not been available with dynamic aspect weaving in statically typed and compiled languages before. Due to the combination of static and dynamic weaving this is now possible with our approach.

Generic advice is supported, because the approach makes it possible to *distribute* the instantiations of the context-dependent parts of the advice code. The advice instances for join-point shadows in already deployed modules are generated when the aspect (module) is compiled. This is possible because of observation (1). Advice instantiation for join-point shadows from yet to know modules is postponed until they are known – by generating them with the static weaver when the respective module is compiled. This is possible because of observation (2). As a result, each aspect (module) carries the join-point–specific advice instantiations for all previously deployed modules, while each module carries join-point–specific advice instantiations from all previously deployed aspects.

Introductions are supported as they are only visible downwards the knowledge hierarchy and, hence, can be applied by the static weaver. This is possible because of observation (2). Static weaving provides the necessary means to replace dynamic introductions that induced side effects in the machine code by semantically equivalent proxies. Because of observation (1), it is furthermore possible to detect potential language-level side effects with modules further up the knowledge hierarchy, hence, reach safety.

A Single Language. The availability of introductions and generic advice furthermore closes the expressiveness gap between "static" and "dynamic" aspect languages for this domain. Thereby a real *single language approach* becomes feasible. In our implementation, the same AspectC++ aspect code now can be woven either dynamically or statically. This increases the reusability of aspects and their applicability to different adaptation scenarios.

Implicit Type Safety. The approach provides implicit type safety for dynamic aspects. With respect to known modules, type problems are detected at compile-time of the dynamic aspect. With respect to yet unknown modules from further down the hierarchy, they are detected at compile-time of the respective module. In the first case the issue has to be solved in the aspect, in the second case in the new module.

Resource-Optimal Weaving. In contrast to dynamic weaving, static weaving is, in principle, overhead free [21]. By falling back to static weaving whenever possible and using run-time weaving only when actually required, the approach is resource-optimal with respect to an AOP-induced overhead.

9.1 Remaining Issues

Side-by-Side Development Restrictions. To ensure completeness of aspect application, the knowledge hierarchy has to be in theory a *knowledge chain*; at the time of deployment, unknown sister modules in the knowledge hierarchy may cause problems. The reason is that for join-points in such sister modules, generic advice would not be instantiated at either side; hence, the dynamic aspect could miss some some relevant join points.

It is, however, a matter of design rules if modules from different branches of the knowlegde hierarchy should actually be able to influence each other. Note that even in such cases it is nevertheless possible to *develop* adaptation modules independently – only at the time of final compilation and deployment there has to be a total order between them. The adaptation of the system is performed strictly incrementally: $system = (...(base + m_0) + m_1) + ... + m_n)$

Introduction Side Effects. The (technical) problem that language level side effects of introductions cannot be applied dynamically hampers the goal of a single language approach. It can lead to situations where aspects that could have been applied statically cannot be applied at run time; thus, we have a semantic difference between static and dynamic weaving.

A possible solution would be to introduce new elements generally in a way that they do not "pollute" the namespace of the target class, but have to be looked up via their own namespace. As mentioned in Section 5, several Java-based approaches follow this strategy by applying introductions as mixins. This automatically prevents *accidental* side effects. However, it also hinders *intended* side effects: Especially in combination with generic and generative programming in C++, the possibility to use aspects for noninvasive overloading or overriding of identifiers in the namespace of an existing class is quite handy. Furthermore, placing introduced elements into an extra namespace would significantly change the current semantics of introductions in AspectC++. Therefore we have refrained from such solution.

Advice Ordering. An unsolved problem is the ordering of static and dynamic aspects that affect the same join point. Here AspectC++ provides a sophisticated mechanism: programmers can specify a required partial order of aspects *per join point*. In our current implementation, dynamic aspects can be ordered by the run-time system, but all dynamic advice is executed indirectly by the static module instrumentation aspect and, thus, inherits its precedence.

9.2 Applicability to Other Language Domains

While the approach is specifically suited to level the expressiveness gap between "static" and "dynamic" aspect languages for binary-code languages such as Ada, C, or C++, it is as well applicable for byte-code based languages such as Java or C#. Many dynamic weavers in the Java domain already provide support for generic aspect implementations and introductions; hence the "feature-question" is not that pushing here. However, they generally seem to suffer from significant performance penalties [16]. On the static side of aspect weaving, approaches such as Spoon AOP [27] have demonstrated that generic advice based on static type information is possible and beneficial with Java as well – specifically with respect to performance. Hence, it should be possible to build a dynamic weaving framework similar to our `dac++` on top of their static weaving framework, potentially resulting in a highly efficient approach for static and dynamic weaving in Java.

10 Summary and Conclusions

We have described a novel approach for dynamic weaving based on static weaving in adaptable systems. Our work focuses on statically typed and compiled languages such as Ada, C or C++. The suggested approach makes it possible to use static join-point context even for dynamically applied aspects, which in turn facilitates AOP features for static cross-cutting and generic aspect implementations that had been unavailable with dynamic weaving before. Our results furthermore show that there is no reason for the current de facto distinction between "static" and "dynamic" aspect languages. It is possible to provide the same amount of AOP features independent of the intended aspect deployment time. Thereby, aspects follow a tradition of other modularization entities from the domain of binary-code compiled languages such as linker libraries, which were first available for static linking only. Today, the decision between static or dynamic linking is transparent, merely just another linker switch. Such deployment transparency is now possible with aspects as well. This was demonstrated with the Squid web proxy example.

The aim of this work was also to show the limits of dynamic weaving in this language domain. The most severe problems are caused by introductions with language-level side effects and the lack of side-by-side development support. As our example shows, many useful applications scenarios are possible regardless of these restrictions.

References

[1] B. Adams. Aspicere homepage. `http://users.ugent.be/~badams/aspicere`.

[2] B. Adams. *Co-Evolution of Source Code and the Build System: Impact on the Introduction of AOSD in Legacy Systems.* PhD thesis, Universiteit Gent, Faculteit Ingenierswetenschappen, Sint-Pietersnieuwstraat 41, B-9000 Gent, Belgium, 2008.

[3] S. Aussmann and M. Haupt. Axon - Dynamic AOP through Runtime Inspection and Monitoring. In *2003 Advancing the State-of-the-Art in Run-Time Inspection Workshop (ECOOP-ASARTI '03)*, July 2003.

[4] C. Bockisch, M. Haupt, M. Mezini, K. Ostermann, and G. Kiczales. Virtual machine support for dynamic join points. In *3rd Int. Conf. on Aspect-Oriented Software Development (AOSD '04)*, pages 83–92. ACM, Mar. 2004.

[5] J. Boner. AspectWerkz - Dynamic AOP for Java. Technical report, 2004. http://aspectwerkz.codehaus.org/aw1/papers_talks.html.

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

[7] A. Colyer, A. Rashid, and G. Blair. On the separation of concerns in program families. Technical Report COMP-001-2004, Lancaster University, 2004.

[8] M. Devillechaise, J. Menaud, G. Muller, and J. Lawall. Web cache prefetching as an aspect: Towards a dynamic-weaving based solution. In M. Akşit, editor, *2nd Int. Conf. on Aspect-Oriented Software Development (AOSD '03)*, pages 110–119, Boston, MA, USA, Mar. 2003. ACM.

[9] R. Douence, T. Fritz, N. Loriant, J. M. Menaud, M. S. Devillechaise, and M. Suedholt. An expressive aspect language for system applications with Arachne. In P. Tarr, editor, *4th Int. Conf. on Aspect-Oriented Software Development (AOSD '05)*, pages 27–38, Chicago, Illinois, Mar. 2005. ACM.

[10] eCos homepage. `http://ecos.sourceware.org/`.

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

[12] M. Engel and B. Freisleben. Supporting autonomic computing functionality via dynamic operating system kernel aspects. In P. Tarr, editor, *4th Int. Conf. on Aspect-Oriented Software Development (AOSD '05)*, pages 51–62, Chicago, Illinois, Mar. 2005. ACM.

[13] W. Gilani and O. Spinczyk. Dynamic aspect weaver family for family-based adaptable systems. In *NetObjectDays (NODe '05)*, Lecture Notes in Informatics, pages 94–109, Erfurt, Germany, Sept. 2005. Gesellschaft für Informatik.

[14] M. Gong and H.-A. Jacobsen. AspeCt-oriented C (ACC) homepage. `http://www.aspectc.net`.

[15] W. Gong and H.-A. Jacobsen. *AspeCt-oriented C Language Specification (Version 0.8)*. Middleware Systems Research Group, Department of Computer Science, University of Toronto, Jan. 2008. `http://www.aspectc.net/acc_manual.pdf`.

[16] M. Haupt and M. Mezini. Micro-measurements for dynamic aspect-oriented systems. In *NetObjectDays (NODe '04)*, volume 3263 of *LNCS*, pages 81–96, Erfurt, Germany, Sept. 2004. Springer.

[17] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In *3rd Int. Conf. on Aspect-Oriented Software Development (AOSD '04)*, pages 26–35, New York, NY, USA, Mar. 2004. ACM.

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

[19] G. Kniesel and T. Rho. A definition, overview and taxonomy of generic aspect languages. *L'Objet, Special Issue on Aspect-Oriented Software Development*, 11(2–3):9–39, Sept. 2006.

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

[21] D. Lohmann, F. Scheler, R. Tartler, O. Spinczyk, and W. Schröder-Preikschat. A quantitative analysis of aspects in the eCos kernel. In *ACM SIGOPS/EuroSys Eur. Conf. on Computer Systems 2006 (EuroSys '06)*, pages 191–204, New York, NY, USA, Apr. 2006. ACM.

[22] D. Lohmann and O. Spinczyk. On typesafe aspect implementations in C++. In F. Geschwind, U. Assmann, and O. Nierstrasz, editors, *Software Composition 2005 (SC '05)*, volume 3628 of *LNCS*, pages 135–149, Edinburgh, UK, Apr. 2005. Springer.

[23] N. Loriant, M. Séegura-Devillechaise, and J.-M. Menaud. Software security patches: Audit, deployment and hot update. In *4th AOSD W'shop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-ACP4IS '05)*, Chicago, IL, USA, Mar. 2005. Northeastern University, Boston (NU-CCIS-05-03).

[24] H. Mashuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In *Compiler Construction (CC '04)*, volume 2622 of *LNCS*, pages 46–60. Springer, 2004.

[25] A. Massa. *Embedded Software Development with eCos*. New Riders, 2002.

[26] I. Nagy, R. van Engelen, and D. van der Ploeg. An overview of Mirjam and WeaveC: an industrial-strength aspect-oriented language and weaver for C. In W. Joosen, editor, *7th Int. Conf. on Aspect-Oriented Software Development (AOSD '08)–Industry Track Proceedings*, pages 68–76, Apr. 2008.

[27] R. Pawlak. Spoon: annotation-driven program transformation — the aop case. In *AOMD '05: Proceedings of the 1st workshop on Aspect oriented middleware development*, New York, NY, USA, 2005. ACM.

[28] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: A flexible framework for AOP in Java. In A. Y. amd Satoshi Matsuoka, editor, *Reflection '01*, volume 2192 of *LNCS*, pages 1–24, Kyoto, Japan, 2001. Springer.

[29] A. Popovici, G. Alonso, and T. Gross. Just in Time Aspects: efficient dynamic weaving for java. In M. Akşit, editor, *2nd Int. Conf. on Aspect-Oriented Software Development (AOSD '03)*, pages 100–109, Boston, MA, USA, Mar. 2003. ACM.

[30] Y. Sato, S. Chiba, and M. Tatsubori. A selective, just-in-time aspect weaver. In *2nd Int. Conf. on Generative Programming and Component Engineering (GPCE '03)*, volume 2830 of *LNCS*, pages 189–208, Erfurt, Germany, Oct. 2003. Springer.

[31] W. Schröder-Preikschat, D. Lohmann, W. Gilani, F. Scheler, and O. Spinczyk. Static and dynamic weaving in system software with AspectC++. In Y. Coady, J. Gray, and R. Klefstad, editors, *39th Hawaii International Conference on System Sciences (HICSS '06) - Track 9*. IEEE, 2006.

[32] O. Spinczyk and D. Lohmann. The design and implementation of AspectC++. *Knowledge-Based Systems, Special Issue on Techniques to Produce Intelligent Secure Software*, 20(7):636–651, 2007.

[33] squid-cache homepage. http://www.squid-cache.org.

[34] D. Suvee, W. Vanderperren, and V. Jonckers. JAsCo: An aspect-oriented approach tailored for component based software development. In M. Akşit, editor, *2nd Int. Conf. on Aspect-Oriented Software Development (AOSD '03)*, pages 21–29, Boston, MA, USA, Mar. 2003. ACM.

[35] D. Wilner. Vx-files: What really happened on Mars? Keynote at the 18th IEEE Real-Time Systems Symposium (RTSS '97), Dec. 1997.

[36] Y. Yanagisawa, K. Kourai, S. Chiba, and R. Ishikawa. A dynamic aspect-oriented system for OS kernels. In *6th Int. Conf. on Generative Programming and Component Engineering (GPCE '06)*, pages 69–78, New York, NY, USA, Oct. 2006. ACM.

[37] C. Zhang and H. A. Jacobson. TinyC: Towards building a dynamic weaving aspect language for C. In *2003 Foundations of Aspect-Oriented Languages Workshop (AOSD-FOAL '03)*, Mar. 2003.