Unification of Static and Dynamic AOP for Evolution in Embedded Software Systems

Wasif Gilani, Fabian Scheler, Daniel Lohman,
Olaf Spinczyk, Wolfgang Schröder-Preikschat

Friedrich-Alexander University Erlangen-Nuremberg

{gilani, scheler, lohmann, spinczyk, wosch}@cs.fau.de

Abstract. This paper discusses how evolution in software systems can be supported by a unified application of both static as well as dynamic aspect-oriented technology. The support for evolution is required statically, where the applications could be taken offline and adapted, and dynamically where going offline is not an available option. While this is straightforward in the static case by taking the system offline and statically weaving the aspects, runtime evolution requires an additional dynamic aspect weaving infrastructure.

Our current implementation of the *family-based dynamic aspect weaving infrastructure* supports most of the features known from the static aspect weaving domain, offers a tailored dynamic aspect weaving support, and is able to target a wide range of applications including embedded systems with very small memory footprint. The availability of a *single language* both for static and dynamic aspects means that the decision whether an aspect is static or dynamic is postponed to the later stages of the deployment of aspects into the sytem, and is decided according to the requirements and available resources. As a case study, we will present our experiences with the static and runtime evolution of the embedded operating system eCos.

1 Introduction

Software evolution is the process of keeping the software up-to-date and bug-free by continuous enhancement, corrections, extensions and customizations as per the emerging requirements. This process involves either adapting the core functional behavior, or the insertion of new non-functional behavior. Lehman defined software evolution as the collection of programming activities intended to generate a new version from an older and operational version [10]. Currently, it is estimated that four out of seven software engineers work on repair and enhancement of existing software [26].

Software evolution can be classified into static and runtime evolution. Static evolution corresponds to compile time changes, and involves modification of the code by taking the system offline, reconfiguring, rapairing, and then recompiling as per the new requirements. Runtime evolution means that the system is upgraded and maintained dynamically at runtime, and is vital for long running systems. Traditionally, runtime evolution is handled with approaches like redundant systems, larger memories, increasing processing power, and feature-rich software. Such approaches noticeably bloat applications, reduce reusability, and increase complexity, costs, and further hinder the evolution of the system.

The evolution could be a continuous change, which happens with the maturity of the technology and involves an incremental adoption approach, or it may be radical and forces a system-wide change. When evolution requires changes to multiple modules, it is said to be crosscutting and hence, non-trivial to localize, and results in code tangling. This code tangling limits the offered levels of evolvability, variability, and granularity of the software. Some concerns like security, profiling, tracing, synchronization, etc., are typically reflected in many points of the code, and therefore difficult to implement as independent encapsulated entities. Aspect-oriented programming (AOP) allows encapsulating crosscutting concerns into completely isolated entities called *aspects,* and injection of the additional behavior, encapsulated by aspects, into multiple modules statically or at runtime by *advice*. With AOP, each and every crosscutting concern is well encapsulated in a separate module, thus, allowing evolution in the system in complete isolation without major redesign of the whole system.

This paper provides details and results about some radical improvements carried out in our dynamic aspect weaver family, which have contributed significantly to further bring down the dynamic weaving costs and making it viable even for embedded systems. We further propose the unification of static and dynamic AOP for the C++ domain, by providing a single language, for achieving static and runtime evolution of software systems. The availability of a *single language* means that the decision whether an aspect is to be deployed statically or dynamically is delayed till the deployment stage.

The remaining paper is organized as follows. We start with the motivation by describing an application scenario. This is followed by a discussion of the related work. The sections 4 and 5 describe the improved implementation of the *family-based dynamic weaver,* and the materialization of the *single language approach*. Section 6 presents a case study which was conducted with the embedded operating system eCos. Finally, section 7 concludes the paper.

2 Motivation: Evolvable Software Systems

While the process of static evolution requires the running system to be taken offline and adapted as per newly emerging requirements, runtime evolution requires that the system could be adapted and maintained on the fly. Such a requirement is vital for highly available systems where downtime could be a catastophe in terms of data loss, performance, revenue, etc. Examples of such highly available systems are mission critical space missions, air traffic control, telephone switching systems, business critical applications, etc. The importance of runtime evolution was demonstrated when NASA's Mars Pathfinder robot, which was launched to relay high-resolution pictures and valuable metereological data of the Martian surface back to Earth, experienced serious malfunctioning. A low priority job held a system-wide important resource. This resulted in repeated resets and thereby loss of important data. Fortunately, the limited runtime evolution capability integrated into the system turned out to be vital for the rescue of the multimillion dollar project which otherwise would had been a total failure. A detailed analysis of the problem with the Mars Pathfinder along with the handling of such runtime evolution problems with our unified static and dynamic aspect-based solution is provided in section 6.3.

2.1 Aspects for Evolution

With AOP, the concerns that are prone to evolution, and are crosscutting in nature, are neatly encapsulated in aspects. In static AOP, these aspects are woven at compile time onto the primary functionality in an additive manner without altering the existing architecture. The aspect code is inlined into classes, and therefore, does not induce any significant overhead into the system. Once woven, the static aspects cannot be removed or reconfigured later during runtime. For evolution, the system has to be taken offline to change aspects as per requirements, and the system has to be recompiled for the changes to be made available.

For long running systems, where going offline is not a choice, a runtime mechanism is needed to enable the system to evolve dynamically. Dynamic AOP provides mechanisms to modularize and thereby apply crosscutting policies encapsulated as aspects into the running system in complete isolation. With dynamic AOP, the runtime evolution involves the addition or replacement of aspects or components.

2.2 Unification of Static and Runtime Evolution

Static evolution with static AOP is more efficient as it incurs low overhead, improves start-up time, and reduces memory usage, but at the expense of flexibility. This option is best suited for devices with resource constraints but is limited because of the lack of the knowledge of execution environments. The solutions supporting exclusively dynamic evolution via dynamic weaving might not be acceptable for some domains due to considerable runtime overhead, and low efficiency. We advocate the principle of *static processing where possible and dynamic processing where needed* by a unified application of static and dynamic AOP. Such a unification demands a homogenous support in terms of the AOP features and a single description language for both static and dynamic aspects. This approach would result in the coexistence of both static as well as dynamic aspects in the system. An evolvable concern would be implemented as an aspect if it has a crosscutting behaviour. The decision whether the aspect is static or dynamic could be removed from the aspect implementation and decided purely as per the requirements and available resources.

2.3 Low-Cost Dynamic Weaving Support

For runtime evolution via dynamic AOP, the system has to be equipped with a dynamic aspect weaver. However, many of the available dynamic aspect weaving infrastructures provide fixed runtime support, are either architecture-specific (C-based weavers) or quite expensive (Java-based weaver) to be deployed on the systems with few kilobytes of memory. Another important motivation of our work is to provide a dynamic aspect weaving infrastructure, which should be efficient, low-cost, portable, and could be tailored down to become viable even for resource-constrained systems.

3 Related Work

Many different approaches have been proposed by the research community for runtime evolution. Some advocate using patterns in several features [7,25]. Other approaches suggest the use of reflection and component frameworks [19,8]. We are more interested in the approaches based on employing AOP for software evolution [21,20,18,11,13]. Most of the AOP-based evolution approaches proposed so far are restricted mainly to applying static AOP for static evolution [18,25,11].

For runtime evolution, there are a number of dynamic weavers available, but all of them provide fixed runtime support and suffer from various limitations like portability, memory and runtime overhead, limited AOP feature support, etc. The weavers in Java are based on bytecode manipulation via the JVM debugging interface, customized class loaders, or virtual machine extensions [15,3,16,4,24]. The current memory requirements of Java-based weavers are an order of magnitude too large for many embedded devices. Though the presence of JVM promises a very portable solution, the mere presence of JVM and core libraries require considerable memory. Furthermore, the Java based weavers typically offer slow execution speed as compared to their counterparts in the C or C++ language. This problem is further aggravated by the employment of the debugger interface in some dynamic weavers, which requires the application to be executed in the debug mode. To speed up applications, some weavers employ JIT compilers, but this requires additional resources.

In the C domain, binary code manipulation is generally employed to support dynamic aspect weaving. The availability of mechanisms to perform runtime hooking, precisely at the required join points, means there is no extra overhead due to unnecessary hooks. *Arachne* [14], *TOSKANA* [9] and *TinyC*^{2} [5] follow the binary code manipulation approach. The actual weaving positions in the binary code are determined with the help of symbol tables and/or debug information, generated by the C compiler. Code inlining or stripping of symbol information has to be disabled. All weavers in C provide fixed runtime supports, and their implementations are limited to specific processors and compilers. The platform dependence means they are not appropriate, especially, in the domain of embedded systems which employ a wide spectrum of CPU and hardware platforms. The performance overhead of these weavers [9,14] is significantly lower than the Java-based systems. The offered AOP features are, on the other hand, also limited.

Disabling of code inlining or stripping of symbol information might be acceptable for C, most C++ compilers implicitly perform such optimizations. Therefore, dynamic aspect weaving via binary code manipulation is not a viable option in the C++ domain. There is a very limited research in the C++ domain for supporting dynamic weaving. We are aware of only one approach in the C++ domain, called *DAO C++*[17], which is based on source code instrumentation. Since the instrumentation process does not depend on binary code, DAO C++ is independent of any architecture or compiler-specific restrictions, resulting in a portable solution. However, the absence of any filtration mechanism means that all join points of the target application are hooked. This leads to significant memory and runtime overhead.

4 A Family-based Dynamic Weaving Infrastructure

None of the available weavers offer a tailorable dynamic weaving support. They follow the traditional one-size-fits all approach. For the development of our dynamic weaver infrastructure, we had two objectives. First, to provide a feature-rich dynamic aspect weaver that could be tailored according to specific requirements, and second, to bring down the cost of dynamic weaving and thereby, make dynamic weaving viable even for embedded devices. We applied the software product line (SPL) [2] approach to the dynamic aspect weaving domain and come up with the family-based weaver [22]. The tailored weavers are generated by selecting only the required set of AOP features from the weaver family. Variant management tools simplify and reduce the complexities associated with the configuration and the generation of variants from the software families. They provide graphical support to define application requirements in the form of feature selection in order to generate application-specific variants. We have employed a variant management tool called *pure::variants* to completely automate the generation process [1]. Besides enabling to generate tailored weavers, the availability of a powerful join point filtration mechanism, and additional mechanisms to exploit the "a-priori-knowledge" of the target application restricts the incurred dynamic weaving overhead due to actually affected joinpoints, actually woven aspects, and used AOP features [22]. The optimizations performed by the exploitation of "a-priori-knowledge" about the target application are comparable to the ones offered by static weavers, which basically exploit the same information for this purpose: actually affected joinpoints, aspects, and used AOP features. The main difference is that this information is implicitly available to static weavers, while it has to be explicitly provided for the generation of a tailored dynamic weaver. Overall, the family-based dynamic weaver infrastructure allows a fine-grained adjustment of the trade-off between flexibility and required resources. In conjunction with the single language approach (Section 5), this perfectly fulfills the goal of minimal overhead: For any kind of application, it is now possible to weave as much as possible statically, while providing as much runtime flexibility as necessary. Static versus dynamic weaving of aspects becomes a configurable and tailorable property.

4.1 Improvements in the Implementation

The architecture of the dynamic aspect weaver family consists of three main building blocks, namely, the weaver binding, the runtime monitor, and the build environment for dynamic aspects [22]. Due to significant improvements in the binding mode (AspectC++), and the general dynamic weaving infrastructure, we were able to further bring down the memory and runtime costs of dynamic weaving. The following subsections describe the various improvements carried out in each of these building blocks.

4.2 Weaver Binding

AspectC++ [12] is employed as a binding mode in our family-based weaver as shown in figure 1. Before describing the improvements, we would like to provide a brief overview of how AspectC++ works as a hooking platform in the weaver family.

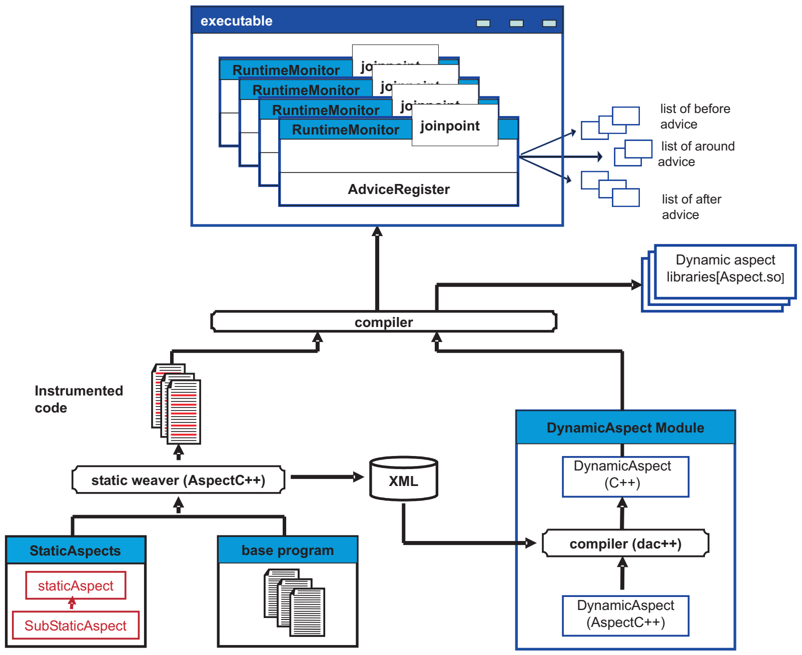

Fig. 1. Architecture of the family-based dynamic aspect weaver

As shown in Figure 2, hooks are encapsulated in the advice code of the static preparation aspect (*instrument*). Since only *before* and *after* advice are defined in this variant of the preparation aspect, the weaving of this aspect would result in a dynamic weaver variant, which supports only before and after advice. If an around advice, or any combination of the three advice types, is to be supported, then the preparation aspect is implemented accordingly. Furthermore, the required amount of context information about the join points is extracted from the static weaver binding mode, and passed via the runtime infrastructure to the dynamic advice code. AspectC++ provides static as well as dynamic context information about the affected join points. The static information includes the join point signature, the argument types, a unique ID, etc., whereas the dynamic information includes current argument values, result value, object instance, etc. It can be noticed that in this particular case, only the join point signature information (*JoinPoint::signature()*) is retrieved by the static advice code and parsed via the inserted hooks to the runtime system. The aspect *instrument* defines a pure virtual pointcut named *dynamicJPS*. The aspect *beforeafterExe* shown in the listing below derives from the *instrument* aspect, and defines exact locations in the source code where hooks should be inserted. The weaving of this aspect would result in the hooking of all execution join points, and all call join points with the exclusion of the functions of the standard library which don't generally contribute to the application's semantics.

```
pointcut std_function_calls() = call("% std::%(...)");
aspect beforeafterExe : public instrument {
  pointcut virtual dynamicJPS() = execution("% ...::%(...)")|| call("% ...::%(...)"
                                  && !std_function_calls();
```

```
aspect instrument {
  pointcut virtual dynamicJPS()=0;
  public:
    advice dynamicJPS():before(){
       ArgsJnPnt<JoinPoint::ARGS> jp;
       jp.jointpointName = JoinPoint::signature();
       monitor<JoinPoint::JPID,MONIT_>::BeforeAdvice(&jp);
    }
    advice dynamicJPS():before(){
       ArgsJnPnt<JoinPoint::ARGS> jp;
       jp.jointpointName = JoinPoint::signature();
       monitor<JoinPoint::JPID,MONIT_>::AfterAdvice(&jp);
    }
};
```

Fig. 2. A static preparation aspect for inserting hooks into the target application

The poincut mechanism in AspectC++, therefore, enables comprehensive filtering of join points for dynamically woven aspects at a fine-grained level, and allows to implement complex hooking policies with ease. During the hooking process, the AspectC++ weaver outputs a project repository, which provides extensive information about the hooked join points, for example their signatures, types, ids, etc. The information is exploited to resolve the pointcuts described in the dynamic aspect code.

We did some significant improvements in the AspectC++ weaver implementation since our last paper [22]. In the previous implementation, the cost of employing around advice for hooking was substantially higher than that of before and after advice. This was particularly problematic in the case of the generation of a variant, from the weaver family, which was required to support both before and after advice. As can be seen from Figure 2, the same context information had to be generated twice at both before and after advice, for each join point. We calculated that in the case of extracting only the signature of the join point, the extra overhead was 13 bytes of memory. In the case of big projects with thousands of join points, this resulted in a significant overhead. In the new version of the AspectC++ weaver, the generation of `tjp->proceed()` function, which is provided in the around advice to invoke the original method, is reimplemented so that `proceed()` can be inlined for small functions. This has resulted in around advice being as efficient as before and after advice in the AspectC++ weaver. Since the cost of the advice types in the weaver family is directly dependent on the cost of the corresponding advice types in AspectC++, this improvement resulted in reducing the cost associated with the dynamic "around advice" in the weaver family. Furthermore, the employment of static around advice helped to avoid extra overhead caused due to the duplicate generation of context information, since the same context information could be shared by different advice types as is shown in figure 3.

4.3 Runtime Monitor

All dynamic aspect weavers follow a centralized model where a single runtime monitor takes care of all interaction between the join points and aspects. Our old version of the dynamic weaver family followed the same design with a single centralized monitor controlling all coordination among the aspects and joinpoints [22]. However, this approach introduces significant runtime overhead as each time when the thread of control

```
advice dynamicJPS():around(){
        ArgsJnPnt<JoinPoint::ARGS> jp;
        jp.jointpointName = JoinPoint::signature();
        jp._that = tjp->that();
        ...
        monitor<JoinPoint::JPID,MONIT_>::BeforeAdvice(&jp);
        tjp->proceed();
        monitor<JoinPoint::JPID,MONIT_>::AfterAdvice(&jp);
}
```

Fig. 3. Modified hooking mechanism employing around advice and templates.

reaches a hooked join point, the list of join points registered with the runtime monitor is traversed to find out the matching join point. The associated complexity with this join point look-up operation is *O(log N)*, where *N* is the number of join points registered with the monitor. Once the right join point is located, the advice stored in the advice containers associated with the join point are executed. Even if there are just empty hooks with no advice registered, this model causes significant runtime overhead.

As a solution, we implemented a new version where each potential join point is provided with a unique runtime monitor. The allocation of unique monitor objects means that the involved complexity for join point look-up is effectively reduced to *O(1)* in contrast to the *O(log N)* complexity of the centralized model. Figure 1 shows the architecture of the weaver family with decentralized runtime monitors. It could have been quite a cumbersome and expensive process to assign each join point with a unique runtime monitor, but templates in C++ come to the rescue, as shown in Figure 3. AspectC++ weaver assigns unique numeric ids to all hooked join points, which are exploited to generate a unique monitor for each join point. It can be seen that the template takes an additional parameter (*MONIT_*), which is used for module identification. This parameter is necessary in the case of "*Extensible Systems*" to be able to weave dynamic aspects even into the modules loaded later into the running system. The components employed in the old implementation that had the sole responsibility of registering and later identifying each of the module's monitor objects for the weaving and unweaving of aspects are no longer needed. This helped to save 5078 bytes of memory which was consumed by the *Extensible Systems* feature in the old implementation.

Furthermore, the memory cost of different AOP features, and hooking is brought down remarkably. This is due to significant optimizations and improvements carried out in the implementation of our static and family-based dynamic weaver. A comparison between the cost of some of the variants of our dynamic aspect weaver family with the old and new implementation are shown in figure 4. It can be seen that the variants with the new implementation consume significantly less memory as compared to the old implementation[22] while providing the same level of AOP feature support. In the new implementation, the variant with minimal AOP feature support consumes exactly 5707 bytes of memory, which is almost half to what it costed in our old implementation (12079 bytes). The variant with maximum AOP feature support (all types of advice, ordering, context, etc.) consumes 10020 bytes of memory which is also significantly lower as compared to previous implementation (23315 bytes). Additionally, the memory cost of each hook has been reduced to just 12 bytes as shown in figure 5. We cannot

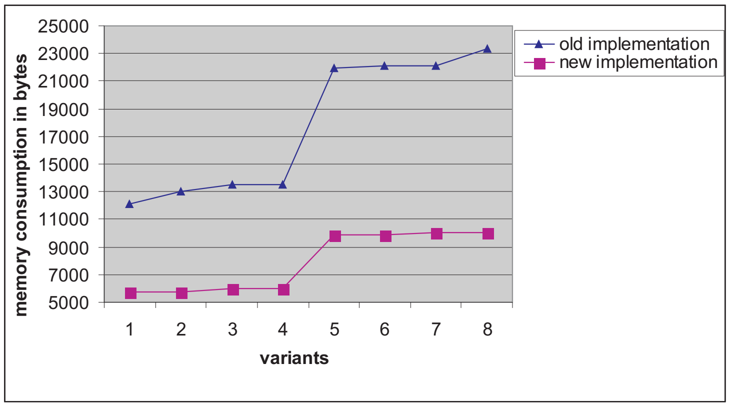

Fig. 4. Different variants of the family-based weaver as per their memory consumption

imagine any further reduction in this cost except moving to binary code manipulation approach which restricts our weaver to specific architectures.

Without Instrumentation

```
void Foo::g( ) {
  puts ("g( )\n");
}

0000003c <Foo::g()>:

3c: push  %ebp
3d: mov   %esp,%ebp
3f: movl .rodata.str1.1,0x8(%ebp)
46: pop   %ebp
47: jmp   puts
```

With Instrumentation

```
aspect instrumentExe : public instrument {
 pointcut virtual dynamicJPS() =
   execution("void Foo1::g()");
};

00000098 <Foo1::g()>:

98: push  %ebp
99: mov   %esp,%ebp
9b: pushl monitor<1,0>::advicebefore
a1: call Cont::trigAdvice(adviceCont_list*)
a6: pop   %eax
a7: movl  .rodata.str1.1,0x8(%ebp)
ae: leave
af: jmp   puts
```

Fig. 5. Cost of hook = 12 bytes

4.4 Build Environment for Dynamic Aspects

In the old implementation, the static aspects were implemented in AspectC++, whereas the dynamic aspects were implemented in C++. In the new implementation, the AspectC++ language has been adopted for the description of dynamic aspects as well (see section 5). Before the dynamic aspects could be loaded into the target application, they have to be transformed into the standard C++ code. A dynamic aspect compiler *dac++* has been developed that transforms the dynamic aspects defined in AspectC++ to standard C++ code. Once transformed, a standard C++ compiler is employed to compile the aspects into shared libraries. The dynamic aspect code itself can be linked either statically with the component code, or loaded at runtime by means of a dynamic aspect loader (Loader). As soon as a dynamic aspect is loaded into the target application, whenever a join point matched by the pointcut definition is reached, the unique monitor for the join point activates the advice code, and returns the control to the application.

5 The Single Language Approach

AspectC++ was designed primarily for the description of static aspects. The adoption of AspectC++ for dynamic aspects required the same level of AOP feature by the weaver family as is available in the static AspectC++ weaver. In our previous paper [23], we analyzed the possibility of supporting a single language, and discussed reasons for the absence of some of the AOP features in the dynamic aspect weavers in the C/C++ domain. We further suggested solutions that have been realized for the dynamic aspect weaver family.

Table 1. Status of the availability of various AOP features in our static and dynamic aspect weaving infrastructures

AOP Features	Static Weaving	Dynamic Weaving
before advice	√	√
after advice	√	√
around advice	√	√
exec join points	√	√
call join points	√	√
object construction	√	√
object destruction	√	√
get/set field	–	–
multiple aspects	√	√
context information	√	√
aspect ordering	√	√
introductions	√	√*

* Introductions of base classes and virtual functions are not yet supported

Table 1 gives an overview of the various AOP features currently supported by both our static weaver (AspectC++), and dynamic weaver family. Our dynamic weaver supports more AOP features than any of its counterpart in the C/C++ domain. The features not supported in the current implementation are get/set fields. This can be considered as challenging to impossible in languages that support C-style pointers[1].

The transformation process of dynamic aspects from AspectC++ to C++ is straightforward. The following listing shows an aspect *Hello* written with AspectC++:

```
aspect Hello {
   advice somePointCut() : before() {
      std::cout << "hello from dynamic aspect! " << std::endl;
   }
```

[1] The support for get/set join points in existing weavers is quite limited, as it is restricted to direct access of global variables.

```
    advice somePointCut() : before() {
       std::cout << "hello from dynamic aspect! " << std::endl;
       std::cout << "signature " << tjp->signature() << std::endl;
    }
 };

 class Hello {
   static void advice1_a0_before() {
     std::cout << "hello from dynamic aspect! " << std::endl;
   }
 template<class ThisJoinPoint>
    static void advice1_a1_before(ThisJoinPoint *tjp) {
      std::cout << "hello from dynamic aspect! " << std::endl;
      std::cout << "signature " << tjp->signature() << std::endl;
    }
 };

 #include "monitor.h" // runtime monitor
 void invoke_a0_before() {
   Hello::advice1_a0_before();
 }
 void invoke_a1_before(void *djp) {
   typedef DynamicJoinPoint<0> DJP;
   Hello::advice1_a1_before<DJP>((DJP*)djp);
 }
 /* module initialisation code */

 __attribute__ ((constructor))
 void __init_dynamic_aspects() {
   monitor<invoke_a0_before,0,0>::registerBeforeAdvice();
   monitor<invoke_a1_before,0,0>::registerBeforeAdvice();
 }
 __attribute__ ((destructor))
 void __fini_dynamic_aspects()  {
   monitor<invoke_a0_before,0,0>::unregisterBeforeAdvice();
   monitor<invoke_a1_before,0,0>::unregisterBeforeAdvice();
 }
```

As seen from the above listing, *dac++* extracts ids of the join points matched by the pointcut from the project repository to translate pointcut descriptions into a sequence of template-based C++ statements, which use join point ids as parameters, to register the advice code. This template-based pointcut matching mechanism provides a very efficient solution in comparison to any mechanism based on signature matching at runtime.

The adoption of AspectC++ both for static and dynamic aspects has resulted in the merger of the static and dynamic AOP for C++, where the decision whether an aspect is static or dynamic is delayed till the deployment stages, and is purely driven by the available resources and the requirements. This type of flexibility is particularly crucial for resource-constrained systems, which follow the principle of static evolution where possible and runtime evolution where necessary.

6 Static and Runtime Evolution in the eCos Operating System

eCos is a small and highly configurable operating system targeted for 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

Table 2. The left table shows the amount of CCCs in the source code of the kernel before and after refactoring, the right table shows the distribution of the cross cutting code over the different CCCs.

	original		aspectized	
	LOC	%	LOC	%
CCC Code	1069	20.54 %	290	6.41 %
Component Code	4136	79.46 %	4237	93.59 %
Total	5205	100 %	4527	100 %

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

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*. Against this library the final applications will be linked.

6.1 Analysis

In the context of a 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 evolvability. For the following discussion we will exemplarily concentrate on the eCos kernel.

For system software clean encapsulation of the different features is crucial in order to be evolvable. 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: *Tracing*, *Assertion*, and *Kernel Instrumentation* (profiling) for development support and *Interrupt Synchronization*. Table 2 (column "original") presents the numbers for each of these concerns individually. Actually, these figures only reflect the number of call sites activating these CCCs, the functional parts of their implementations were not taken into account here.

The results of the analysis show that eCos indeed is configurable to a great extent, but certainly lacks evolvability. 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 evolution of the eCos kernel.

6.2 Static Evolution

During the case study, we enhanced the evolvability 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 right columns of Table 2. 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.

6.3 Runtime Evolution

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 [27]. After a few days of successful operation the spacecraft experienced total system resets and each of these resets caused a loss of valuable metereological 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.

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 had 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 runtime evolution. 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.

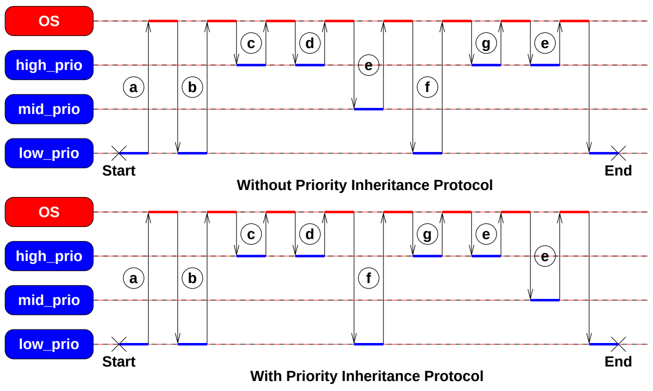

Fig. 6. Execution sequence without and with priority inheritance protocol.

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 [23] 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 in-acceptable overhead in comparison to static aspects.

Implementation We already re-factored eCos' priority inheritance protocol into a static aspect in previous work [6]. 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 straight forward 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);
}
```

Evaluation Setup 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 6, the system calls used at each step of the execution sequence can be obtained from table 3.

Table 3. System calls used in the test application

	System Call	Description
a	mutex_lock(&mutex)	lock mutex, as it has not been locked before it can be successfully locked
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	cyg_mutex_unlock(&mutex)	thread high_prio unlocks the mutex

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 evolution. 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[2]. 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[3] 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.

Evaluation Results 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 4. 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 (prio. 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 (prio. 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 delcared 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 (see section 4) 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 aspects are fulfilled by the aspect itself.

For the assessment of the runtime 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 3 and Figure 6). The results of these measurements are shown in Figure 4. These results confirm the results of the memory measurement. Variant *dynamic (perfect)* only shows minimal decline of runtime performance in contrast to variant *static*, i.e. the runtime 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 runtime 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 (*dy-*

[2] gcc version 4.03, binutils version 2.16.1
[3] the boot loader provided along with eCos

Table 4. Memory consumption of the different eCos variants measured in bytes. *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.

	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 (prio. inh.)	2206	5375	0	77	0	26495	18325
static (no. prio. inh.)	2194	4427	0	0	0	26445	17305

namic (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.

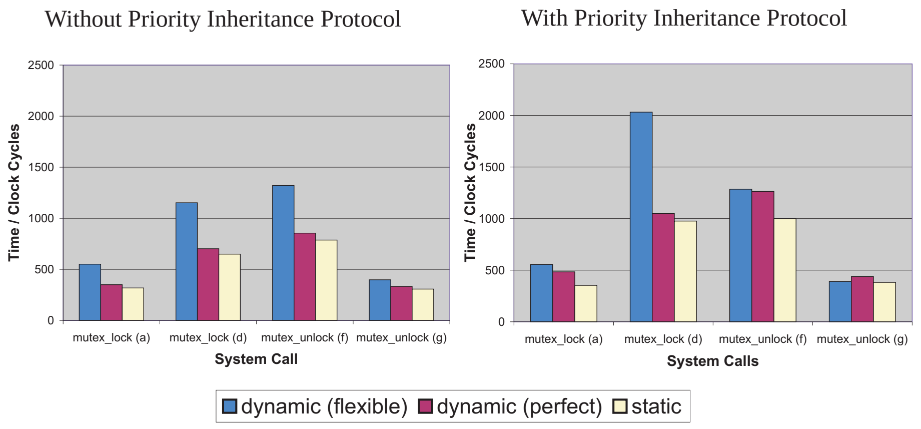

Fig. 7. Runtime performance comparison of different eCos variants. The left diagram shows the execution of the analysed system calls with the priority inheritance protocol, the right diagram the execution times without priority inheritance protocol.

6.4 Discussion

In general, this case study shows that for many concerns in embedded system software, aspect-oriented implementations and especially dynamically woven aspects are affordable. After the refactoring and the integration of the dynamic weaver infrastructure into eCos, the system now offers an even better static as well as runtime evolvability. Better static evolvability because crosscutting concerns and crosscutting optional features are now cleanly modularized and encapsulated. Better runtime evolvability because it is now possible to adapt to changing requirements at runtime. However, this case study also illustrates, that dynamic evolution is not for free, especially when many join points have to be instrumented the overhead increases sensibly.

7 Summary

In this paper, we have presented our improved version of the dynamic aspect weaver family, which has significantly reduced the memory and runtime overhead associated with the dynamic aspect weaving. Additionally, the availability of a single language for both static and dynamic aspects allowed to provide a unified mechanism for both static and runtime evolution. Such a unified mechanism results in an increased levels of flexibility and evolvability of software systems as the decision whether an aspect is a static or a dynamic one, is postponed to the later stages of deployement, and is decided as per the requirements and available resources. By virtue of our family-based dynamic weaver, even systems with very small memory footprint are able to afford some degree of dynamism to deal gracefully with the runtime evolution requirements they are subjected to.

References

1. Danilo Beuche. Variant management with pure::variants. Technical report, pure-systems GmbH, 2003. http://www.pure-systems.com/.
2. Krysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming. Methods, Tools and Applications.* May 2000.
3. Andrei Popovici et al. Just in Time Aspects: efficient dynamic weaving for java. In *AOSD '03*, pages 100–109, March 2003.
4. C. Bockisch et al. Virtual machine support for dynamic join points. In *AOSD '04*, pages 83–92, March 2004.
5. C. Zhang et al. TinyC: Towards building a dynamic weaving aspect language for C. In *AOSD-FOAL '03*, March 2003.
6. Daniel Lohmann et al. A quantitative analysis of aspects in the eCos kernel. In *EuroSys '06*, pages 191–204, April 2006.
7. Douglas C. Schmidt et al. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. 2000.
8. Fabio Kon et al. Monitoring, Security, and Dynamic Configuration with the DynamicTAO Reflective ORB. In *IFIP/ACM Distributed Systems Platforms and Open Distributed Processing (Middleware '00)*, April 2000.
9. M. Engel et al. Supporting Autonomic Computing Functionality via Dynamic Operating System Kernel Aspects. In *AOSD '05*, pages 51–62, March 2005.
10. M Lehman et al. Towards a theory of software evolution - and its practical impact. In *ISPSE '00*, pages 2–11, November 2000.
11. Neil Loughran et al. Supporting Product Line Evolution With Framed Aspects. In *3rd AOSD (AOSD-ACP4IS '04)*, March 2004.

12. Olaf Spinczyk et al. AspectC++: An aspect-oriented extension to C++. In *TOOLS '02*, pages 53–60, February 2002.
13. Philip Greenwood et al. Dynamic framed aspects for dynamic software evolution. In *ECOOP-RAM-SE '04*, June 2004.
14. R. Douence et al. An expressive aspect language for system applications with Arachne. In *AOSD '05*, pages 27–38, March 2005.
15. R. Pawlak et al. JAC: A flexible framework for AOP in Java. volume 2192, pages 1–24, 2001.
16. S. Aussmann et al. Axon - Dynamic AOP through Runtime Inspection and Monitoring. In *ECOOP-ASARTI '03*, July 2003.
17. Sufyan Almajali et al. Dynamic Aspect Oriented C++ for Upgrading without Restarting. In *AITA '04*, July 2004.
18. Sven Apel et al. Combining Feature-Oriented and Aspect-Oriented Programming to Support Software Evolution. In *ECOOP-RAM-SE '05*, pages 3–16, July 2005.
19. T. Ledoux et al. OpenCorba: A reflective open broker. volume 1616, pages 197–214, 1999.
20. Takashi Ishio et al. Program Slicing Tool for Effective Software Evolution Using Aspect-Oriented Technique. In *PSE '03*, pages 3–12, November 2003.
21. Walter Cazzola et al. AOP for software evolution: a design oriented approach. In *SAC '05*, pages 1346–1350, November 2005.
22. Wasif Gilani et al. Dynamic aspect weaver family for family-based adaptable systems. In *NODE '05*, pages 94–109, September 2005.
23. Wolfgang Schröder-Preikschat et al. Static and dynamic weaving in system software with AspectC++. In *HICSS '06*, 2006.
24. Y. Sato et al. A selective, just-in-time aspect weaver. In *GPCE '03*, volume 2830, pages 189–208, October 2003.
25. Hassan Gomaa. Architecture-centric evolution in software product lines. In *ECOOP-ACE '05*, July 2005.
26. C. Verhoef. Towards automated modification of legacy assets. *Annals of Software Engineering*, 9(1-4):315–336, May 2000.
27. David Wilner. Vx-files: What really happened on mars? Keynote at the 18th IEEE Real-Time Systems Symposium (RTSS '97), December 1997.