A Dynamic AOP-Engine for .NET [*]

Andreas Frei, Patrick Grawehr, and Gustavo Alonso
Department of Computer Science
Swiss Federal Institute of Technology Zürich
CH-8092 Zürich, Switzerland

{frei,alonso}@inf.ethz.ch
pgrawehr@student.ethz.ch

ABSTRACT

AOP technology is being used in many areas where separation of concern is important. Many systems available today have been developed with Java in mind and are currently being improved on all layers from the interception mechanism and aspect interpreter to the language and component model.

The application of AOP paradigms to languages other than Java is still not widespread although there are advantages in doing so. For instance by adapting AOP techniques for .NET we can benefit from the multi-language dimension. An application written in managed C++ could be easily crosscut with an aspect written in C# or any other managed language. This leads to a new abstraction of aspects where we are not bound to any language.

In this paper we present a system that provides a basic dynamic AOP-Engine. It allows weaving and unweaving at runtime of advices in applications running in a .NET environment. This process takes place without having to stop applications, thereby opening up the opportunity for many different forms of adaptation and dynamic evolution.

1. INTRODUCTION

AOP [18] has been introduced as a technique for separation of concerns that copes with scattering and tangling of non-functional code over multiple classes. For example, logging behavior is widely accepted as a non-functional behavior of an application which can be added or removed as needed. Typically, there are three basic forms of introducing new behavior in an application (a process called *weaving*). The first one is statically and it is the most widespread. In this form of weaving, the application is modified by recompiling it and adding the new functionality in the form of *aspects* [3, 38, 35]. The second form of weaving is load-time weaving. In this approach, *pointcuts*, which are method hooks used to monitor when to trigger the execution of aspects, are inserted into the code before the class gets loaded [23, 5, 4, 22]. The last form of weaving is runtime weaving and will be the one we are dealing with in this paper. As the name implies, these systems allow to weave aspects at runtime while the application is operational and without interrupting its operations [28, 34]. Both runtime and load-time weaving are regarded as dynamic AOP solutions.

Dynamic AOP is currently mainly pursued in the context of Java. For instance, solutions are available on Java platforms as varied as JBoss [14] or mobile infrastructures [29]. There is, however, only limited work on dynamic AOP in environments other than Java. For instance, systems like JAsCo.NET [39] and Weave.NET [19] insert the pointcut hooks not later than at load-time. Thus, new aspects can only operate on the already inserted hooks, thereby limiting the applicability of the approach. To allow fully dynamic AOP in these systems, one would need to insert hooks on all possible method entries and exits. This would lead to an unacceptable slowdown as every method invocation would also involve checking for aspects to be executed.

In this paper we tackle the problem of providing runtime weaving in a .NET environment focusing on the basic behavior needed to enable *before, after* and *around* weaving (i.e., the aspects are executed before, after, and around a given method). The contribution of the paper is a complete architecture for runtime weaving in .NET. The paper also provides an in depth discussion on the differences between implementing dynamic AOP in a JVM and in .NET. Our approach is based on developing a mechanism that allows to insert pointcuts at runtime. Moreover, we do not separate the definition of pointcuts and aspects. Our dynamic AOP engine works with *dynamic link libraries* (DLL) that contain both the aspect and the definition of the pointcut. The engines accepts such DLLs (through a process we call aspect insertion) and generates stubs for the required pointcuts, which are then inserted into the code to capture method entries and exits. This takes place by using a combination of the profiler and debugger interface which allows us to query the runtime metadata available in .NET and inserting stubs into the methods using this information. For methods that have been native compiled, the engine triggers a recompilation that produces new native code to include the stubs. Even though the application has to be run in debugging and profiling mode, the overhead compared to the pre-hook insertion technique is negligible. In addition, by weaving stubs at the Intermediate Language level we can modify applications written in one language with aspects built using

[*]The work presented in this paper was supported (in part) by the National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS), a center supported by the Swiss National Science Foundation under grant number 5005-67322.

a different language. With this, the engine we present in this paper can be used in a wide variety of ways. For instance, it can be used to maintain, upgrade and patch highly available applications without interrupting their operations. Functionality can be extended and monitored by adding *before* and *after* pointcuts. With the *around* pointcut buggy methods can be circumvented and replaced by new methods.

The paper is structured as follows. Section 2 introduces our dynamic AOP-Engine solution. In section 3 we evaluate our solution with benchmarks and look at its applicability to middleware architectures. Section 4 gives an overview of dynamic AOP solutions in Java and .NET. Finally, we state our conclusion and describe some future work.

2. DYNAMIC AOP-ENGINE IMPLEMENTATION

In the following we give an overview of the system architecture and explain in detail the steps involved on insertion of an aspect and the processing of an aspect advice.

2.1 Overview

The .NET environment has similar concepts as Java. The Common Language Runtime (CLR) is Microsoft's virtual execution environment which has been standardized as the Common Language Infrastructure (CLI). The CLI is responsible for executing the Common Intermediate Language (CIL) which corresponds to the bytecode in Java. With the CLI a new language dimension has been added. Combined with the Common Type System (CTS), the CLI allows to use different languages which are intermixed. The CTS defines the types that can be expressed in metadata and CIL and the possible operations that can be performed on those types. It supports different languages such as C#, VB.NET, VC.NET, Eiffel, Oberon, Perl, Python, Smalltalk, etc.

In order to extend or change the functionality of a .NET application we implemented the core functionality of a dynamic AOP-Engine [15]. Thus, the engine allows us to exchange methods and support the insertion of stubs at the beginning and/or at the end of a method. These requirements are part of the basic functionalities of an AOP system and are called *around, before,* and *after* crosscuts.

2.2 System Architecture

All the components of our dynamic AOP-Engine are shown in Figure 1. The program which will be crosscut at runtime is shown on the lower left side with the CLR in the middle as shadowed box. The CLR is divided by the process boundary indicating that a switch from the running program to the debugger would stop all running threads in the program and switch to the debugger.

The top part shows the *AOPDebugger* and its subcomponents. These are the common parts of a debugger for managed applications, as suggested by the CLR documentation. The AOPDebugger controls the execution of all other components, controls the *AOPProfiler* and contains the large data structures that are needed to manage the system. Finally, it also provides the *AOPController* as a user interface to control the aspect insertion and removal. The AOPDebugger and AOPProfiler use the corresponding CLR debugger interface [11] and CLR profiler interface [30].

The approach taken by PROSE [27] and Woole [34] suggest to implement dynamic AOP using the Java Platform Debugger Interface (JPDA). Whereas the two projects use two different layers of the JPDA, they have in common that the two intercept code by setting breakpoints on places where a joinpoint is needed. When a breakpoint is reached the thread continues its execution in the engine to execute the aspect advice. In .NET however reaching a breakpoint suspends all threads before continuing its execution. The debugger approach is therefore not applicable to .NET as on reaching a joinpoint the whole application would be stopped.

To avoid this problem, we use a combination of the debugger and profiler API. Instead of setting a breakpoint, a stub is inserted into the method bodies by the AOPProfiler.

The AOPProfiler is a module that is loaded into the target process by the CLR. It is written in unmanaged C++, since the use of any managed code is not allowed, as it would cause deadlocks inside the runtime. For example, a deadlock might occur in a callback that occurred because of a garbage collection event. The CLR locks its heap for this purpose. If then some memory from the managed heap would be allocated a deadlock would be the result.

The AOPProfiler consists of two almost independent parts: the first contains the callback interface, whose methods are called by the CLR when certain events happen. These events include [30], loading and unloading of modules, loading and unloading of classes, suspending and resuming of threads, Just in Time Compilation, entering and leaving of methods, memory allocation, and garbage collection.

As we will see later, the Just-in-Time compilation will be the hook we intercept to replace one method for another.

The second part of the AOPProfiler contains the communication classes which are used to communicate with the AOPDebugger and the AOPController. To communicate with the AOPDebugger, the AOPProfiler uses a specialized shared memory interface, and to communicate with the AOPController, it uses COM interop. COM Interop [21] is a feature provided by the CLR that allows the use of managed classes over COM. The AOPController is written in C#, so that it is possible to use the managed reflection API to handle the aspects. The main task of this component is to manage the aspect libraries and to provide the runtime environment for them.

Finally, we have the user providing aspects in the bottom right corner of the picture. An aspect can be written in any language as long as it is both supported by the CLR and compiled to a standalone DLL before execution. Unlike other AOP implementations, we do not need external files to describe the aspect. It is completely self-contained and contains the information on where and how methods should be crosscut. An aspect DLL must at least contain one class that implements the *Aspect* interface. The AOPController will use this interface to ask the aspect which functions it wants to have crosscut and what kind of crosscuts should be made. At a later stage, when the crosscut locations are hit, the aspect advice is executed.

The dynamic AOP-Engine consists therefore of the *AOPDebugger, AOPProfiler,* and *AOPController* which get loaded into the process when the program is started.

2.3 The AOP-Debugger

The AOPDebugger is by far the most complex part of the dynamic AOP-Engine. To debug a managed application, an Object of type *CLSID_CorDebug* has first to be created by using the COM *CoCreateInstance* function. If successful, the caller gets an interface of type *ICorDebug* which will be the anchor point for all the operations performed within the

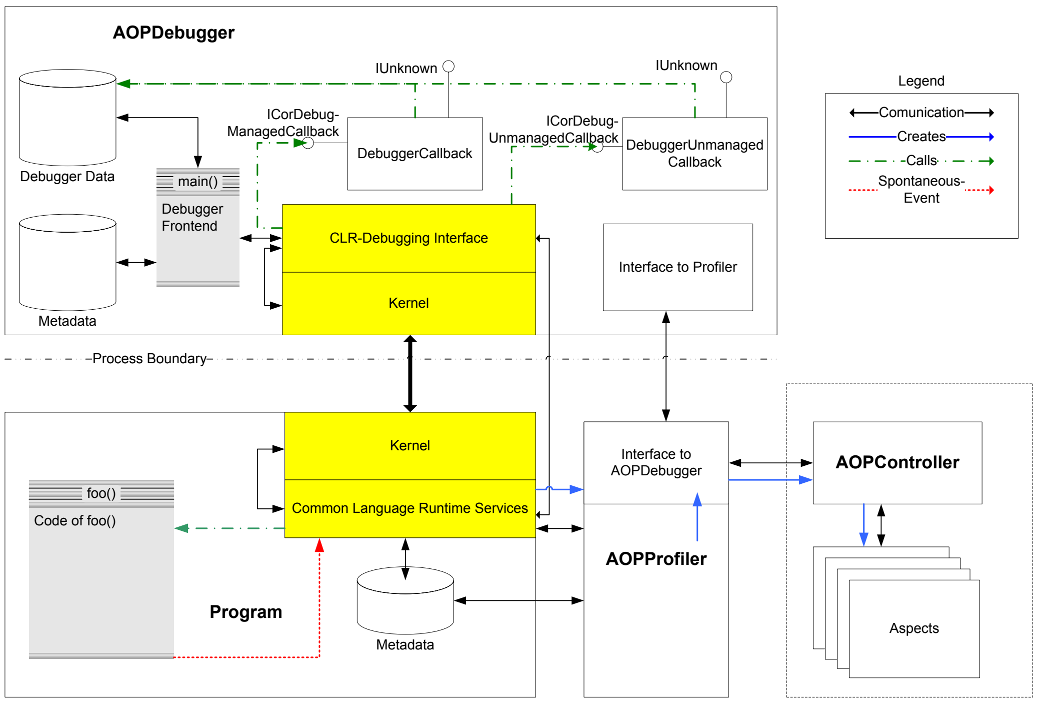

Figure 1: AOP System Overview

AOPDebugger. Our primary focus is on passing Callback-Interfaces to this newly created object. That means we need to have classes that implement *ICorDebugManagedCallback* and *ICorDebugUnmanagedCallback*. As the names suggest, the CLR will call the methods of these interfaces whenever a managed or unmanaged debug event occurs in the process which is supervised. A debug event is anything inside that process that might be worth knowing, ranging from very obvious events like breakpoints and access violation exceptions to more informative ones like newly created threads or loaded libraries.

Some of the events are mainly intended for the debugger itself, because the debugger needs to keep its internal data structures in sync with the process, others are meant to be passed on to the user. Internally, the CLR debugging interface communicates with the real CLR (meaning the execution engine) running in the process being tested. To start debugging, we use *ICorDebug::CreateProcess()*, which creates and attaches the execution engine (shadowed box in the bottom part). Two distinct callback interfaces are used as managed or native code generate different events. In case of managed code the event is processed by the CLR in the target process and information is sent to the AOPDebugger about it. In case of native code, the event is processed by the Windows kernel debugging capabilities and handled exactly in the same manner as for a native Windows debugger. In many situations, the unmanaged events are much more powerful because they can also be applied to finding problems in the CLR itself or in other unmanaged parts of the target process, like the AOPProfiler module. The picture also shows the major data structures involved: The AOPDebugger's internal data contains information about the process which runs under its control: symbol information, function names, modules, and source code. A second database contains all the metadata attached to a managed assembly at compile time. The metadata describes all the namespaces, classes, methods, and parameters and types used in an assembly. The CLR refuses to execute code without proper metadata.

The lower part of the image shows the contents of the target process address space. To the left, we see the controlled process itself, which has the CLR loaded and contains some functions (foo, in the Figure 1). It also contains the metadata database used by the CLR to verify the code. Unfortunately, this database is not synchronized with the debugger side copy, so a change on one side will not be reflected on the other.

The AOPDebugger is built on the CLR debugger CORDBG, which Microsoft provides in full source code, and is based on the CLR debugging API. This API provides functionality to access the metadata and the state of the process. The state consists of: Threads, state of the stacks, functions on the stacks, values of local and global variables, loaded functions, their names, addresses, code, signature, loaded additional modules, processor registers, and exceptions.

Most of these state sets can be changed with the proper command. In our extended debugger we added the following features to control the aspects:

- Load, unloading and inspection of aspects and their behavior.

- Debugging of aspects

- Communication with the AOPProfiler

- Automatic AOPProfiler startup
- Disassembling, decoding and assembling of instruction streams.

2.4 The AOP-Profiler

The AOPProfiler is a module that will be loaded into the target process by the CLR. We use it to provide fast in-process callbacks from the process to the aspects. The name is a bit misleading in our context, but comes from the API [30] it uses, which was originally meant to support writing of code profilers for the CLR. Our Profiler supports this profiling as an extra feature, but the profiling functionality is actually not needed in our application.

Our AOPProfiler consists, as already noted above, of a COM-Class that provides a callback interface for the CLR. The most important function contained therein is *ICorProfilerCallback::JitCompilationStarted()*. It is called by the CLR when it is about to JIT-compile a managed function to native code. This is the point where our AOP System gets notified: It will look up whether a replacement exists for the method in question and if necessary, performs the replacing, using *ICorProfilerInfo::SetILFunctionBody()*.

The AOPProfiler has four main tasks in supporting dynamic AOP:

1. When the event *JitCompilationStarted* is generated, replace the current code with new one, if necessary.
2. Load the AOPController into the process and communicate with it.
3. Communicate with the AOPDebugger.
4. Provide the stubs for the crosscutting itself, as described later.

Since it is not allowed to call managed code from within any of the profiler callback functions, we must make sure that any calls from the AOPProfiler module to the AOPController happen in a separate thread, completely unrelated to the one in which the CLR delivers its events. This is a very hard restriction, since no synchronization can be used between the threads of the two components. Any attempt to wait until one thread has entered a certain state or performed some other operation is guaranteed to deadlock the CLR sooner or later. The cause of the problem is the aspect engine which runs inside the same CLR as the process and therefore also uses the same critical sections (and the same profiler callbacks). Since many of the internal functions and data structures of the CLR are not re-entrant, it heavily relies on critical sections and mutexes to secure them. In the CLR the calling function (i.e., the JIT compiler) already owns its mutex when the corresponding profiler callback (*JitCompilationStarted*) is executed. If the profiler called itself into managed code which would need JIT compilation or performed any other operation that needs the same mutex, the deadlock would be unavoidable. Since it is not documented which operations require which mutexes, we chose to secure the two components of the profiler as independently as possible. The real implementation contains many more threads which are often only used to guarantee that some calls can be handled asynchronously. This obviously also causes problems because it is never exactly known when which thread has completed what operation. Sometimes, all we could do was to let some time elapse and hope the data or the result of the other thread would be available then. If not, we had to assume something had failed and would need to abort with a timeout, because another thread might be waiting for the same mutex our thread currently owned.

2.5 Insertion of an Aspect

We will now concentrate on how all modules work together to provide the AOP functionality. Figure 2 shows schematically all the calls and messages involved in inserting a crosscut into a method. Lets call the method we are going to crosscut *foo* and the aspect which will be introduced *FooAspect*. It is contained as a class in Module *Aspect.dll*.

The process starts with the user entering the command to load the aspect in the AOPController. This request will be placed in a queue for the AOPProfiler and will be delivered as soon as the profiler communication thread is ready to handle the request. The message handler thread of the profiler will receive the request and forward it to the AOPController (1). The AOP-Engine will open the dll using the reflection API and look for any classes that implement the *IAspect* interface and whose name ends in *Aspect*. Having found one, it will instantiate the class (in our example *FooAspect*) and call its *Init* method, providing it with an interface to perform crosscut operations (2). The normal case is now that, from within *Init*, the aspect requests joinpoints. For our example, we assume the aspect wants to be notified whenever the method *foo* is entered or left. The AOP-Engine notifies the AOPProfiler (3), which forwards the request through the interface to the debugger (4,5). The AOPDebugger will then grab the metadata corresponding to the function under examination from the database and combine it with the current MSIL code (gray), as found in the target process (6,7,8). This information is bound together and passed on to the disassembly module, where the entire function is decoded (9).

Now the function generator (10) needs to add code to the beginning and the end of the function. The code should use the callback function inside our *FooAspect*. To call a function, one needs the metadata token for it, since in IL code everything uses these tokens instead of pointers, these being portable and independent of the true memory layout of the system. As we are about to perform a cross-module call, we need a MethodRef token [9]. The straight-forward approach would be to register the Module *Aspect.dll* as being required by our program module and then create a MethodRef token using the *IMetaDataEmit* interface. Unfortunately, this does not work as expected: For reasons unknown to us, the CLR would later, when our modified method was JIT compiled, complain that it could not find the requested method. This caused a code verification failure, which terminated the process. Copying the code from the aspect into the program did not work either, because tokens are module-relative and cannot be used outside of where they were defined. Nor does *IMetaDataEmit::DefineMethod()* correctly update the metadata of the process. As noted above, the metadata is not synchronized between the debugger and the profiler, so only the AOPProfiler can reasonably use *IMetaDataEmit*. Our solution is based on using a special metadata table we are able to do modifications at runtime. This metadata has a very similar organization as a rational database where every kind of information (Types, Methods, Parameters, Variables) has its own table. The corresponding token points to the index of a table. This table lists a description of the method head with the number and type of parameters.

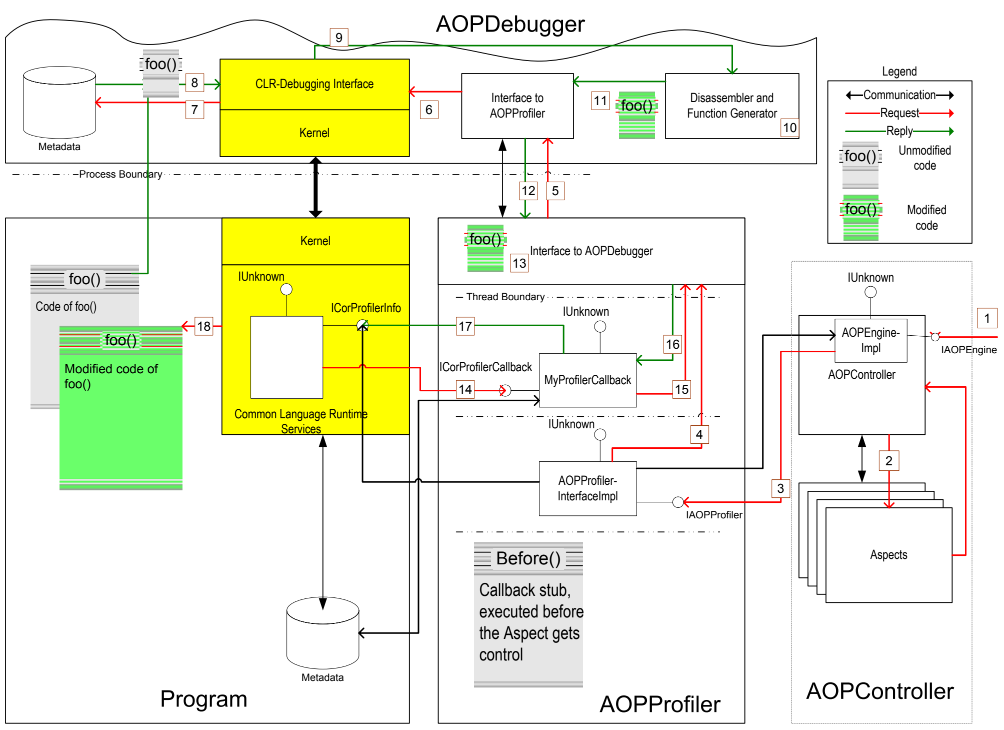

Figure 2: Inserting an Aspect

The CIL *caller* instruction needs just that to perform a call into unmanaged code, given by its linear address. The instruction is unverifiable since it is impossible to guess from an address whether it really points to a valid native code function. The function generator therefore inserts a *caller* instruction into the instruction stream, with the address of a callback function provided specially for this purpose by the profiler. We call this a stub, since it is just a very short piece of code that forwards the call to the aspect engine. As the debugger cannot create the required metadata tokens itself, it creates an extra data structure with information about what additional tokens are required to compile the function (11). The new function with the additional token information is now sent back to the profiler (12), where it is cached for the moment (13).

When the function *foo* is about to be called for the first time, the CLR calls the profiler's *JitCompilationStarted* callback (14). The callback function checks whether there is a matching function ready for insertion (15). If yes, the function body is removed from the cache and prepared for insertion (16). Before the real insertion can happen, the token information data structure is examined and the corresponding tokens are created using the *IMetaDataEmit* interface. The code block is also patched once again to insert the real tokens at the appropriate locations since the debugger had only put place holders there. When all tokens have been created, the AOPProfiler calls *IProfilerInfo::SetILFunctionBody()* with the new method body (17) and exits from the callback function. The CLR now performs the JIT compilation using the new code and starts executing it (18).

Once an aspect is no longer needed, the stub needs to be removed. This can be done by using the *ICorProfilerInfo:: SetFunctionReJIT()* the next time the method is called to re-JIT the original code.

2.6 Execution of an advice

A short time after the new function starts executing, our callback is hit for the first time. The CLR will perform a managed-to-native transition, pack some parameters to the stack and call our stub for the *before* callback (1) (see Figure 3). The parameters include a module token, a method token, the number of parameters and array of all the parameters passed to *foo*, so that it will not only be possible to uniquely identify the calling method, but also to inspect the parameters it was called with. The stub (2) performs some maintenance operations and then calls the aspect engine with the same parameters it was been called initially (4), optionally with an additional pointer to an interface that allows parameter modification (3). From the module and method token, the aspect engine finds the aspect that is responsible for this method and calls its *BeforeCallback* method. The aspect may use *IVarCallback* to modify the variables (5) or perform any other appropriate operation. If the aspect has completed its work, it returns control to the aspect engine and the stack is subsequently unwound, until *foo()* can continue with its execution. The next callback will happen just before *foo()* returns. This time, it will use another stub but, apart from that, the operation is identical.

3. EVALUATION

In order to evaluate our solution we compare it to another dynamic AOP system, JAsCo.NET [39]. Further on we discuss our solution in its applicability in two middleware

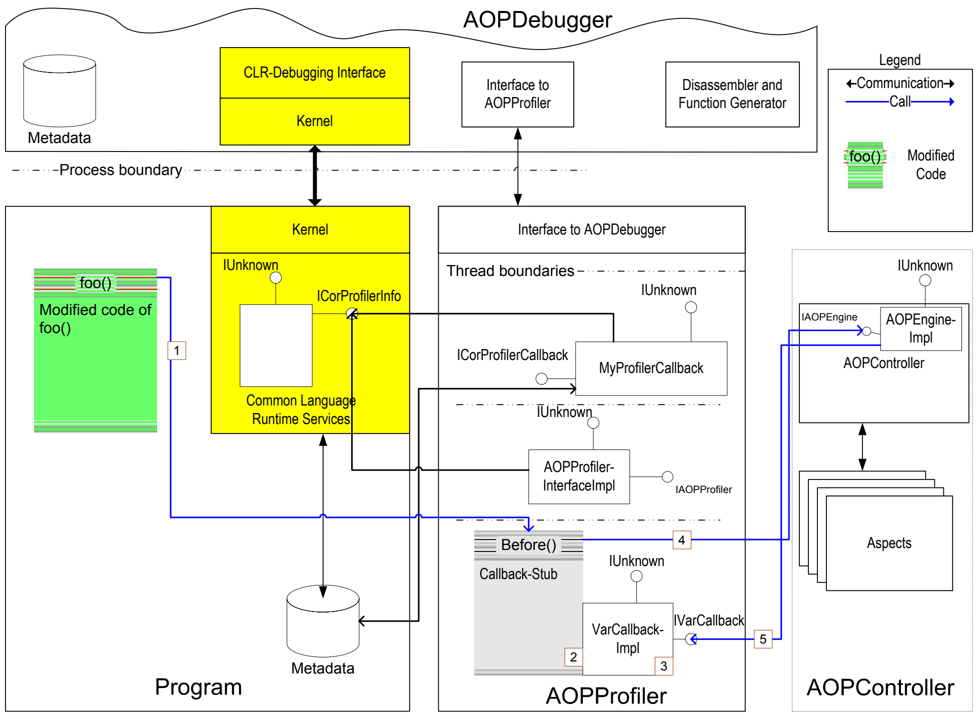

Figure 3: Processing a Callback

scenarios. At the end we give two concrete aspect examples in C# and VB.NET reflecting the language independence of our solution.

3.1 Benchmarks

The performance analysis of our dynamic AOP-Engine was done with a benchmark introduced by JAC [23]. JAsCo.NET allows to insert joinpoints in all method entries and exits. The joinpoints have to be inserted before the application is started. Therefore, JAsCo.NET is a dynamic AOP solution as it allows to add and remove aspects at runtime which uses the pre-hooked joinpoints.

We adapted the JAC benchmark for .NET containing nine empty functions with varying parameters and return types. Each run looped 100'000 times over these functions. We let them run several times so that the standard deviation was less than 1%. The benchmarks were performed on a Pentium 4, 3GHz, with 1 GB of RAM, and on the .NET v1.1.4322 in workstation flavor.

The results show the average time of one loop over these nine empty functions. In the first test, we measured the execution time of the functions without AOP overhead. The second test runs the functions with AOP support where we attached the dynamic AOP-Engine for our solution and weaved the functions with all joinpoints for JAsCo.NET.

From the second measurement we observe a three times increase with the dynamic AOP-Engine (note the difference in units). Several reasons lead to such a big overhead even though there were no aspects inserted. We have to run our engine with the debugger and the profiler. The debugger requires the code to be run in non optimized mode and the profiler produces events which are handled in the dynamic

Test	dAOP-Engine	JAsCo.NET
normal Execution (release build)	5 ns	
with Debugger/Adapted (release)	16 ns	46.208 μs
before()	61.489 μs	52.130 μs
after()	1.969 μs	48.123 μs
around()	64.543 μs	47.797 μs

Table 1: Benchmarks

AOP-Engine. JAsCo.NET performed several orders of magnitude worse, as every hook has to be checked for a registered aspect.

For the *before* and *around* crosscut we are 18% and 35% slower than JAsCo.NET. The reason lies in the memory allocation for the function arguments. On any method execution, the arguments have to be locked up and a structure needs to be built up in order to pass it on to the aspect advice. In case of an *after* crosscut, the benchmark results show a significant improvement as no memory has to be allocated. If an *after* crosscut needs the target function arguments the *before* crosscut is needed which would slow down an *after* crosscut again.

In a last test we measured the insertion time of an aspect. An insertion in the dynamic AOP-Engine can take in our examples up to 0.8s, due to the generation of the stub which has to be inserted on each method. In JAsCo.NET an insertion is done immediately at the activation by enabling the connectors.

Both solutions are still in a first prototype stage and performance improvements can still change these results. Nevertheless, it shows already the characteristics of each solution. The results show that pre-hook insertion might lead to a performance degradation even no aspect is inserted. Our approach does not change the application as long as no aspects are inserted thereby minimizing overhead.

3.2 Possible Middleware Scenarios

We evaluate our solution for its applicability in middleware scenarios. Figure 4 shows two typical scenarios borrowed from [33] where we evaluate the applicability of our solution. The scenarios show a web service infrastructure Figure 4a) is a three-tier *WebServer, BusinessLogic, and Database*, architecture and Figure 4b) a two-tier WebServer and Database architecture. In the three-tier architecture an update at the web server side would involve placing new webpages in the web server without restarting the web server. The same holds for databases where we are able to migrate a database to a new machine so that the *Clients* do not notice any interruption. The *Business Logic Process* is therefore a key factor in settings where any restart would cause an interruption for the client.

As web services are getting more powerful more business logic is getting enclosed in the web server and the typical multiple process architecture of Figure 4a) is getting replaced by a web server and database architecture in Figure 4b). In such an architecture the scripts of web-pages and the business logic are executed in the same process. This imposes new requirements on the dynamic AOP-Engine as it has to be attached to the web service process.

Three-Tier Infrastructure

The first scenario is described in [33]. It uses web services as frontend for a separated business logic process with a database backend. It is a typical 3-tier infrastructure: each tier is running in its own process which can be either on the same machine or, for performance reasons, distributed. The communication between the processes is done by using an RPC style or a messaging infrastructure.

In such a scenario the web service layer would be very thin and presumably contain no important data as that would be queued in the business logic process or the database. Changes on the web service side could therefore be done by providing new web pages or Active Server Pages (ASP) which can contain web service descriptions or scripts.

The backend database is in many cases replicated and fail-overs are handled by the database. But the problem exists for the business logic process. In most cases a bug in a library would lead to a restart of the whole process making sure in advance that all states have been stored to the database.

With our solution we start the business process together with the dynamic AOP-Engine.

To enable AOP functionality at a later time the application has not to be pre-hooked with all possible joinpoints. At runtime we are then able to add new behavior into the business logic process for example to log certain transactions. As buggy methods may be recognized at runtime a patch using an *around* crosscut could fix the problem temporarily.

The *engine* would then allow to replace the buggy code with an *around* aspect.

Web Service Tier

In many new applications the business logic and web service layer are merged into the web service process and leaving the database on a remote machine. A classical example is described as the PetShop store [25].

In this case we cannot just simply start the dynamic AOP-Engine together with the web service process as this process first needs to be generated upon the first web request. To allow dynamic AOP behavior anyway, we have to place a first web request ourselves, which would allow us to attach the dynamic AOP-Engine to the created process. The delay between the first web request and successful attachment of the engine is important as any method invocation in this time would prohibit the notification of the engine with the required method identifications which are needed for a later stub insertion. Currently it is therefore not possible to attache the dynamic AOP-Engine to an already running process.

In case the profiler supports in the future a way to gather the method identifications the process could be attached anytime. It would even allow to turn off most of the expensive events generated for the profiler, thereby increasing the performance of the whole application.

3.3 Aspect Examples

We have introduced two scenarios where the dynamic AOP-Engine can be used and give now a short overview of two aspects and their pointcuts. As our dynamic AOP-Engine only relays on MSIL code and therefore on its Common Type System (CTS) we are able to write aspects in any managed language. We give two examples one of which is in C# and the other in VB.NET, see code examples below.

A pointcut can be specified directly in the *Init* method (line 11) of the Aspect by using the *IAspectEngineCallback* class. The method *IAspectEngineCallback.AddJoinPoint(..)* (line 15) takes four arguments. The first specifies the methods which are crosscut in a regular expression like form. That is tailored by three booleans: crosscut *before*, crosscut *after* and whether the passed *arguments* need to be changed. The return value of the *BeforeCallback* (line 19) method specifies if an around, (*return 0;*) or a before (*return 1;*) has to be executed.

C# Aspect

The C# examples shows how a buggy method can be replaced with an aspect. Assume a programmer hard coded by mistake the calculation of an account interest. We have therefore to replace that method code by a dynamically set interest rate and return the new value.

This is achieved by initializing a pointcut (line 11) with a joinpoint for the specific class and method *BusinessLogic. Interest::getInterest*. The *BeforeCallback* method calculates the new interest by looking up first the actual interest rate and then calculates the interest from the amount in the first parameter (line 25). The new interest will be stored temporally in the new result field and return with the argument 0 specifying that the running code can be discarded. The *AfterCallback* is then used to return the new interest (line 36).

```
1 using System;
2 using AspectEngine;
3
4 namespace NsAspect {
5   public class CSharpAspect:Aspect  {
```

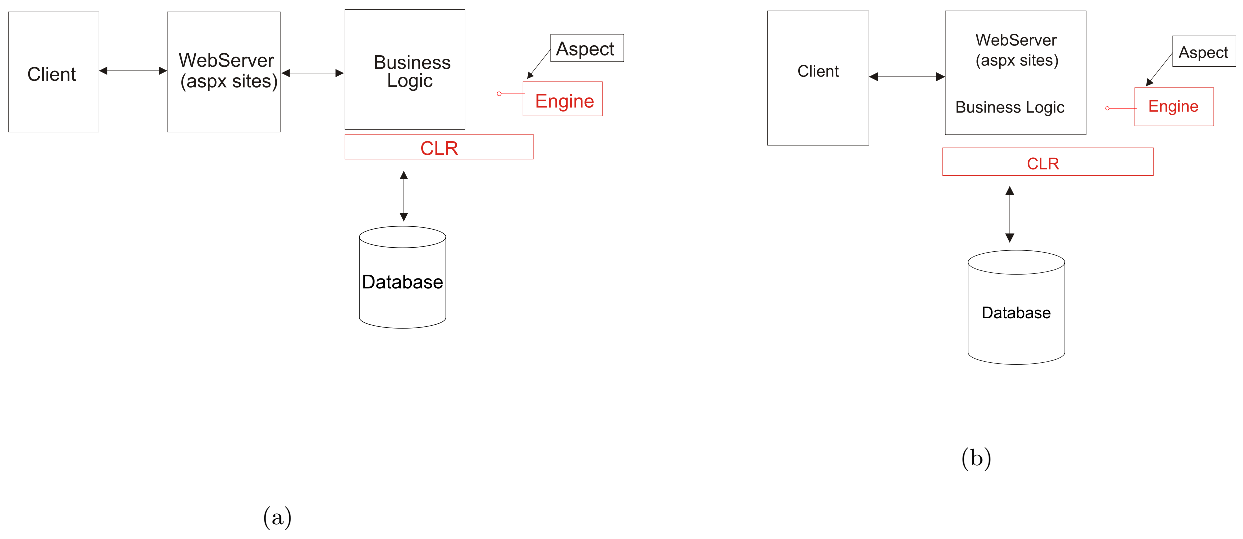

Figure 4: .NET Architecture Scenario

```

  public CSharpAspect() {       }

  private int retresult;

  public override int Init(IAspectEngineCallback callback) {
    base.Init(callback);
    //regexp for joinpoints, crosscut before, after, parameters modifiable
    callback.AddJoinPoint("BusinessLogic.Interest::getInterest",
             true,true,true);
    return 1;
  }

  public override int BeforeCallback(
    int thread, int instanceno, int moduleno,
    IDT type, Array numargs, IVarCallback setvars)
  {
    Console.WriteLine("-CSharpAspect::BeforeAndAroundCallback()-");
    // calculate the new rate with the current interest
    retresult = currentInterest * (int)numargs(0);

    // around advice
    return 0;
  }

  public override void AfterCallback(
    int thread, int instanceno, int moduleno,
    IDT type, object oldret, ref object newret)
  {
    Console.WriteLine("-CSharpAspect::AfterCallback()-");
    newret = retresult;
  }
}
```

VB.NET Aspect

In this example we write a VB.NET aspect which crosscuts all methods in the *BusinessLogic.DBConnection* class to trigger all remote calls and remote returns. The *BusinessLogic* can be written in any other .NET language. The writer of the aspect does not have to care about the language.

In the *Init* method a crosscut is specified which adds a joinpoint to all methods in the *BusinessLogic.DBConnection* class. The arguments of the crosscut (line 17) specify a *before* and *after* joinpoint and as no *around* is requested, the return value is set to 1 (line 18). With the next call for a method in the *DBConnection* class, a log message will be printed before and after the execution to trigger the remote invocations.

```
 1 Imports System
 2 Imports AspectEngine
 3
 4 Namespace NsAspect
 5
 6   Public Class VBLogAspect
 7     Inherits Aspect
 8
 9     Public Sub VBLogAspect()
10     End Sub
11
12     Public Overrides Function Init( _
13       ByVal callback As IAspectEngineCallback) As Integer
14       MyBase.Init(callback)
15
16       callback.AddJoinPoint("BusinessLogic.DBConnection::*",
17            True, True, False)
18       Return 1
19     End Function
20
21     Public Overrides Function BeforeCallback( _
22       ByVal thread As Integer, ByVal instanceno As Integer, _
23       ByVal moduleno As Integer, ByVal type As IDT, _
24       ByVal numargs As Array, ByVal setvars As IVarCallback) _
25       As Integer
26
27       Console.WriteLine("-VBLogAspect::BeforeCallback()-")
28       'before crosscut
29       Return 1
30     End Function
31
32
33     Public Overrides Sub AfterCallback( _
34       ByVal thread As Integer, ByVal instanceno As Integer, _
35       ByVal moduleno As Integer, ByVal type As IDT, _
36       ByVal oldret As Object, ByRef newret As Object)
37
38       Console.WriteLine("-VBLogAspect::AfterCallback()-")
39     End Sub
40
41   End Class
42 End Namespace
```

4. DYNAMIC AOP COMPARISON

In this section we first give an overview of dynamic AOP implementations in Java and compare them to our solution. Later on, we show the differences to .NET solutions.

Java dynamic AOP Solutions

Apart from static weaving solutions like AspectJ [3] or Javassist [5] several projects are working on introducing AOP concepts at runtime.

JAsCo [37, 7] is an aspect-oriented programming language originally tailored for the component-based field, in particular the Java Beans component model. The language introduces two concepts: aspect beans and connectors. The aspect bean defines the behavior and some specifications of the hook. This makes the hook context independent and re-usable. The connector is used for deploying one or more hooks within a specific context, the so called traps. On occurrence of such a trap, the appropriate aspect bean is executed. These traps have to be inserted before the execution is started and a overhead results for all methods, even if they are not crosscut. To overcome this problem, the JAsCo community introduced in [10] a HotSwap solution to insert and remove traps at runtime. A .NET implementation is explained later in the .NET solutions part.

AspectWerkz [4] uses its own classloader to weave joinpoints at loadtime of the class. Once these joinpoints are weaved they can be disabled and enabled again and re-used by new aspects inserted at runtime. To allow insertion of aspects at runtime, joinpoints have to be inserted on any possible used place when the class is loaded. No .NET implementation is supported so far.

JAC [23, 24] is a Java Aspect Component framework to build aspect-oriented distributed applications. Two levels of aspect-oriented programming are distinguished:

- The programming level is similar to AspectJ where new aspects, new pointcuts, and new wrappers are implemented.
- The configuration level is used to customize existing aspects. By using a configuration language it is possible to reconfigure the system with predefined aspects.

The joinpoints used by JAC have to be established at compile-time or load-time of the class. Any dynamic changes of aspects can be done at the pointcut level of the already established joinpoints.

The RtJAC [13] project overcomes the problem of introducing new joinpoints at runtime, where they had to change the loading mechanism of JAC. No .NET implementation is supported so far.

PROSE [27, 26, 28] is a just-in-time weaver of aspects which allows to weave an application at runtime. Pointcuts and aspect advice are defined in an aspect class. Such an aspect can be woven at runtime without previously woven joinpoints.

PROSE uses in one version the JVMDI which is the lowest layer of the Java Platform Debugger Architecture (JPDA). By using the debugger interface, the joinpoints can be set as breakpoints. When a breakpoint is reached, the PROSE engine will execute the appropriate aspect advice. In a second version of PROSE, joinpoints are inserted for every method entry, exit and field modification. The so called joinpoint stubs are evaluated in the PROSE engine to call an aspect advice, if the respective joinpoint is defined. The insertion of the joinpoint stubs is done at JIT compilation time by using the IBM Jikes Research Virtual Machine.

In our dynamic AOP-Engine we do not have to change the CLR JIT mechanism but we trigger each JIT compilation. This allows us on a later time to insert a joinpoint stub. It is comparable with setting a breakpoint at the specified location as introduced in the JVMDI version of PROSE except that the breakpoint consists of a joinpoint stub. This approach is necessary since the profiler of .NET is not allowed to execute managed code directly.

Woole [34] is a just-in-time aspect weaver by Sato et al.. A previous implementation [6] of Woole was only based on the the *HotSwap* [12, 17] mechanism introduced in the JVM 1.4. The HotSwap mechanism in the JVM allows to replace binary compatible classes (see chapter 13) at runtime, which replaces a method body with a woven message body. The woven message body may therefore contain a *before* or *after* aspect advice. This mechanism even allows to replace a method body with the aspect advice alone which would be analogue to the the *around* advice in AspectJ.

Woole takes advantage of breakpoint execution in the Java Debugger Interface. A breakpoint which represents a joinpoint, can be set in the running application. As soon as a breakpoint occurs, Woole executes the aspect advice. By having breakpoint execution and method replacement in one system, it can choose the right behavior depending on performance issues or system decisions. Currently, the decision for one or the other solution is done by hand but could be automated onto the system itself.

As Woole relays on the Java Debugger Interface the concept is, as in PROSE, not fully applicable in .NET.

.NET AOP Solutions

CLAW [16, 8], Cross Language Aspect Weaver, is a runtime extension to the CLR which allows to weave aspects at JIT compile time. CLAW explores the usage of the profiler interface to generate dynamic proxies at JIT compile time. The pointcuts are defined in an external XML file which is parsed on insertion of the aspect. The dynamic proxies is then used to dispatch to the aspect advice which is similar to our approach. As we take advantage of the debugger and profiler interface, we are able to take the one which is appropriate for a specific task to insert our stub and remove it again, which is not yet possible in CLAW. So far no implementation is available.

JAsCo.NET [39] The JAsCo community is also working on the JAsCo.NET implementation. Additional to the two concepts, the .NET solution provides support for specifying aspects on specific Web Service interactions. This approach requires a new Web Service component model. The JAsCo.NET web service has built-in traps, which enable run-time aspect application and removal. As not all third party web service vendors have JAsCo.NET enabled services, regular web services are transformed into JAsCo.NET web services at startup. To allow web services to be crosscut with our approach, we are currently investigating the possibility to attach the engine at runtime to the web service process. Even though JAsCo.NET has been introduced for web services it can also be used for standalone applications.

Weave.NET [19] describes a solution to weave aspects at load-time in a language-neutral way. As .NET is based on the Common Type System (CTS), any language which maps to the CTS can be woven with other such languages. The pointcuts are defined in separate XML files which are combined with the aspect code to weave with the target assembly. Because weaving is done on the assembly level, no source code is needed for cross-language weaving. As Weave.Net weaves the aspects at load-time directly into the code without a trap mechanism, the aspects cannot be removed again at runtime. It is therefore not a truly dynamic solution.

Loom.NET presents an AOP solution for weaving aspect-code and component-code which uses the mechanisms of the Common Type System in .NET. Their approach is therefore also not restricted to one language. The first implementation [35] weaves aspect statically whereas they allow in a second approach dynamic weaving [36]. The pointcuts are defined as attributes in the aspect and woven at instantiation time of the target class. As long as the instances are used, the aspects in that instance are executed. Loom.NET does not support unweaving or disabling of aspects. Even if Loom.NET supported the unweaving or disabling of aspects, dynamic AOP would be very difficult to achieve. Every object creation would need to be changed to call the aspect weaver to create an instance together with a generic aspect supporting before, after and around. This way, pointcuts are only applicable to interface methods.

The following other AOP solutions are explained only briefly as they are either static or in a pre-prototype stage.

CAMEO [20] is a static aspect weaver which hooks into the C# compilation phase. The aspect weaver uses the abstract graph of the Shared Source Common Language Infrastructure (SSCLI) [1] to weave the aspect code. Eos [32, 31] is a static AOP approach similar to AspectJ but features added support first-class aspect instances and instance-level advising. AopDotNetAddIn [1] is an AspectJ like language which can be plugged into the .NET infrastructure. AOP.NET [2] describes a similar dynamic AOP approach. However, they propose to use the profiler to replace the relative virtual (RVA) at runtime to call a proxy method. No implementation is available so far.

5. CONCLUSION

In this paper we introduce and benchmark a dynamic AOP-Engine for .NET. New aspects can be introduced at runtime without previous pointcut weaving. Since performance and uptime are vital for middleware applications, they can take advantage of weaving pointcuts at runtime. The application does not need to be restarted nor do classes need to be reloaded to enable the insertion of new pointcuts.

The dynamic AOP-Engine takes over ideas from Java AOP systems like PROSE or Woole which use the debugger interface. In the .NET implementation we had to take a combination of the debugger and profiler interface as the debugger alone would suspend the whole application. A stub is prepared and inserted on the next method invocation which in case of already JIT-ed code requires a re-JIT of the method.

[1]The Shared Source Common Language Infrastructure (SSCLI) is also known as the open source implementation of the CLI called Rotor

Our approach takes advantage of the multi-language possibilities in .NET. Programs written in one language can be crosscut with any CTS compatible language. A middleware application written in different .NET languages can therefore be crosscut with the same aspect implementation. The dynamic AOP-Engine has to be bound to the process at startup. As the around advice is part of the possible pointcuts, the engine can also be used to replace buggy methods. Although it is only a momentary bug-fix, it can be handled at runtime and does not require a restart of a vital process.

Future work will involve attaching the engine to a web service process. Even though we have put our attention to be able to extend and replace managed code, it would be possible to extend the dynamic AOP-Engine to support unmanaged native applications and thereby support legacy applications.

6. REFERENCES

[1] AopDotNetAddIn. http://www.geocities.com/m_mesalem/aop.html.

[2] AOP.NET. http://wwwse.fhs-hagenberg.ac.at/se/berufspraktika/2002/se99047/contents.

[3] AspectJ. Online Documentation, 2003. http://eclipse.org/aspectj/.

[4] Jonas Bonér and Alexandre Vasseur. AspectWerkz. http://aspectwerkz.codehaus.org.

[5] Shigeru Chiba and Muga Nishizawa. An Easy-to-Use Toolkit for Efficient Java Bytecode Translators. In *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering (GPCE '03)*, pages 364–376. Springer-Verlag, September 2003.

[6] Shigeru Chiba, Yoshiki Sato, and Michiaki Tatsubori. Using HotSwap for Implementing Dynamic AOP Systems. In *Workshop on Advancing the State of the Art in Runtime Inspection (ASARTI), ECOOP'03*, July 2003.

[7] Cibrn, M.A., D'Hondt, M., Suve, D., and Vanderperren. Jasco for linking business rules to object-oriented software. In *Proceedings of the International Conference on Computer Science, Software Engineering, CSITeA03*, June 2003.

[8] John Lam's Software Development Weblog. http://www.iunknown.com/000092.html.

[9] CLI, 2002. Common Language Infrastructure, Partition I-V, ECMA-335, second revision.

[10] JAsCo Community. JAsCo HotSwap for AOSD. http://ssel.vub.ac.be/jasco/running.php.

[11] Debugger API, 2002. The Debugging API of the Microsoft CLR.

[12] Mikhail Dmitriev. *Safe Class and Data Evolution in Large and Long-Lived Java Applications.* PhD thesis, University of Glasgow, May 2001.

[13] Miklos Espak. Improving Efficiency by Weaving at Run-time. In *5th GPCE Young Researchers Workshop 2003*, September 2003.

[14] Marc Fleury and Francisco Reverbel. The JBoss Extension Server. In *Proceedings of the 4th ACM/IFIP/USENIX International Middleware Conference*, pages 343–373. Springer-Verlag, June 2003.

[15] Patrick Grawehr. AOP Implementation in .NET (german). Master's thesis, Swiss Federal Institute of Technology (ETHZ), October 2003. http://www.iks.inf.ethz.ch/publications/dsa/.

[16] John Lam. Cross-Language Load-Time Aspect Weaving on Microsoft's Common Language Runtime. Demo at AOSD 2002, April 2002.

[17] JVM 1.4 HotSwap. http://java.sun.com/j2se/1.4.2/docs/-guide/jpda/enhancements.html.

[18] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and

John Irwin. Aspect-Oriented Programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[19] Donal Lafferty and Vinny Cahill. Language Independent Aspect-Oriented Programming. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 1–12. ACM Press, 2003.

[20] B. D. Chaudhary M. Devi Prasad. AOP Support for C#. In *Second AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, mar 2003.

[21] MSDN Help library.

[22] Nanning. http://nanning.codehaus.org/.

[23] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: A Flexible Solution for Aspect-Oriented Programming in Java. In *Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns*, pages 1–24, Kyoto, Japan, September 2001. Springer Verlag.

[24] Renaud Pawlak, Laurence Duchien, Gerard Florin, and Lionel Seinturier. Dynamic Wrappers: Handling the Composition Issue with JAC . In *39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39)*, Santa Barbara, California, July 2001.

[25] Microsoft .NET Pet Shop 3.x: Design Patterns and Architecture of the .NET Pet Shop. http://msdn.microsoft.com.

[26] A. Popovici, G. Alonso, and T. Gross. Just In Time Aspects: Efficient Dynamic Weaving for Java. In *2nd Intl. Conf. on Aspect-Oriented Software Development, Boston, USA*, March 2003.

[27] A. Popovici, T. Gross, and G. Alonso. Dynamic Weaving for Aspect Oriented Programming. In *1st Intl. Conf. on Aspect-Oriented Software Development, Enschede, The Netherlands*, April 2002.

[28] Andrei Popovici. *PROSE, a study on Dynamic AOP*. PhD thesis, Swiss Federal Institute of Technology (ETHZ), 2003.

[29] Andrei Popovici, Andreas Frei, and Gustavo Alonso. A proactive middleware platform for mobile computing. In *Proceedings of the 4th ACM/IFIP/USENIX International Middleware Conference*, pages 455–473. Springer-Verlag, June 2003.

[30] Profiler API, 2002. The Profiler API of the Microsoft CLR.

[31] Hridesh Rajan and Kevin Sullivan. Eos: Instance-Level Aspects for Integrated System Design. In *Proceedings of the ESEC/FSE 2003*, September 2003.

[32] Hridesh Rajan and Kevin Sullivan. Need for Instance Level Aspects with Rich Pointcut Language. In *Proceedings of the Workshop on Software Engineering Properties of Languages for Aspect Technologies (SPLAT)/ AOSD 2003*, March 2003.

[33] Performance Comparison: .NET Remoting vs. ASP.NET Web Services. http://msdn.microsoft.com/library/.

[34] Yoshiki Sato, Shigeru Chiba, and Michiaki Tatsubori. A Selective, Just-in-Time Aspect Weaver. In *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering (GPCE '03)*. Springer-Verlag, September 2003.

[35] Wolfgang Schult and Andreas Polze. Aspect-Oriented Programming with C# and .NET . In *Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, april 2002.

[36] Wolfgang Schult and Andreas Polze. Dynamic Aspect-Weaving with .NET. In *Proceedings of International Symposium on Object-oriented Real-time distributed Computing (ISORC) 2002*, November 2002.

[37] Davy Suve, Wim Vanderperren, and Viviane Jonckers. Jasco: an aspect-oriented approach tailored for component based software development. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 21–29. ACM Press, 2003.

[38] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proceedings of the 21st international conference on Software engineering*, pages 107–119. IEEE Computer Society Press, 1999.

[39] D. Verspecht, W. Vanderperren, D. Suve, and V. Jonckers. JAsCo.NET: Unraveling Crosscutting Concerns in .NET Web Services. In *Proceedings of the Second Nordic Conference on Web Services NCWS'03*, November 2003.