Aspect Weaving with C# and .NET

Michael A. Blackstock
Department of Computer Science
University of British Columbia
201-2366 Main Mall
Vancouver, B.C. Canada V6T 1Z4
michael@cs.ubc.ca

ABSTRACT

Since current object oriented programming languages don't have existing support for aspects, aspects are often supported through language extensions [1, 2]. Another approach is to use the existing language to encapsulate aspect behaviors, and provide an additional language to express cross cutting statements [3-5]. Finally, other systems [6]including the one described in this paper use features of the existing language to specify aspect behavior and cross cutting.

This paper presents a prototype weaver called AOP.NET that demonstrates the feasibility of supporting aspect oriented programming in C# without the need for language extensions, or a cross cutting statement file. All of the information related to supporting AOP including the cross cutting statements is contained in the aspect declaration. The cross cutting statements are expressed using a language feature called attributes which are used to annotate methods, fields and classes with meta data in languages targeting the Common Language Runtime (CLR) such as C#. Since attributes are supported in all CLR languages it should be possible to maintain .NET language independence with this approach [3, 5].

AOP.NET demonstrates the feasibility of static and transparent dynamic weaving in .NET. Unlike other .NET dynamic weavers, no changes are required to the source code of clients of functional components for dynamic weaving, the same weaving engine is used in both a static tool and dynamic weaving run time host, and it is implemented completely in C#.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: D.3.2 Language Classifications - *Multiparadigm languages, Object-oriented languages*

General Terms

Design, Experimentation, Languages

Keywords

Aspect oriented programming, Common Language Infrastructure, AOP.NET, Reflection, Language-independence

1. INTRODUCTION

Aspect-oriented techniques allow programmers to modularize cross cutting concerns (called aspects) from functional object oriented or procedural systems. Typically the functional components [7] of system are specified by conventional modules, procedures or classes but since aspect descriptions cannot often be described in existing generalized procedural (GP) languages [7], language extensions are often provided as part of a aspect-oriented system to express aspects including cross cutting statements and aspect behavior. An aspect weaving technology is used to interpret aspects language extensions and compose them with components.

In the AspectJ system [2] for example, aspects are described in an *aspect* declaration. An aspect declaration includes the behavior that will be composed with the components of a system in *advice* declarations. In AspectJ advice can be applied to "well-defined points of execution of the program" [2] called a *join point*. A *pointcut* is a set of join points where advice is to be applied. In AspectJ, pointcuts are expressed in a pointcut language and may stand on their own as a named pointcut, or may be included in an advice declaration. A single aspect declaration may contain named pointcut definitions, advice definitions, as well as some standard methods and fields encapsulating all of the relevant properties of a cross cutting concern in one module.

Another approach is to use an existing language to modularize aspect behavior in conventional module or class declarations and then add cross cutting or composition statements in a separate specification [3-5, 8]. This forces the programmer to break the overall modularization of an aspect with cross cutting statements in one file, and aspect behavior in another.

Finally it is possible to express cross cutting using the existing component language. PROSE, for example embeds both advice and cross cutting statements in anonymous classes embedded in an Aspect class [6].

Weaving is the process of composing a component with a cross cutting aspect. A weaver can compose objects at statically at compile time, or after aspects and components are compiled to object code, byte codes or intermediate language generated by the compilers. It is also possible to compose components with aspects at run time my calling a weaving library, supplying the aspects and components as parameters, or dynamically when the component is loaded for use by an application when it is first called [9, 10].

Some weaving systems [3, 11] require changes to application source code to initiate the weaving process on specified components. Calls are made to a weaver library at appropriate locations in the code, such as at the beginning of an application launch. Unfortunately this counters the idea that a functional

component should be oblivious to the aspects that may alter its behavior [12]. In fact, if the programmer decides to sprinkle code to weave components throughout the code, weaving itself would become a cross cutting concern. A dynamic weaving system should also make it possible to compose aspects with components at run time and only when those components require composition. This will avoid overhead associated with weaving except where necessary. Examples of a dynamic weaver with these qualities (for Java) is PROSE [6, 9] and CLAW [5] for C#.

When compared to dynamic weaving, static weaving has the advantage of low runtime overhead since all weaving is done before application execution, and any degradation in performance of functional components is associated mostly with the additional aspect behaviors composed with it. Since there are advantages to both dynamic and static weaving [10], it would be ideal if a system could support both depending on the application.

This paper describes a prototype weaver called AOP.NET used in a static tool and dynamic weaver host for .NET. This prototype does not require external cross cutting statement files, and does not require extensions to the C# language by leveraging .NET attributes[1].

The basic unit of compilation for the .NET system is called an assembly. .NET assemblies contain metadata associated with a code element such as a class, interface, method or fields. Using a language construct called an attribute, .NET also supports the extension of such metadata. The metadata in an attribute associated with a class, method or field can then be used by tools or at runtime using the FCL reflection APIs to implement various services. The .NET Framework for example uses attributes to support services such as serialization and method interception[2]. Other existing AOP technologies for C# and .NET leverage the use of custom attributes [11, 14].

One advantage of expressing aspects including cross cutting statements in the source language, is that it may be possible to create new AOP constructs that may not be supported by a given language extension's AOP model [6]. For example, in PROSE a cross cutting specialization was created to weave aspects with components that are encountered only between certain times. Perhaps an aspect that adds backup capabilities to the system would be composed during those times [6].

The aspect oriented programming model used by this prototype is based on AspectJ since it is well documented and widely understood. Section 2 of this paper discusses this model in more detail and discusses how aspects are encapsulated in C# classes for use with the weaver. Section 3 discusses how AOP.NET works in both a static weaving tool and a dynamic weaving application host. The weaving prototype is then evaluated in section 4 with an example component and aspect. The prototype is compared and contrasted with previous approaches for C# and .NET and some other languages such as Smalltalk and Java in section 5. The paper concludes in section 6.

[1] [13] Drayton, P., Albahari, B., and Neward, T. *C# in a Nutshell*. 2002, Sebastopol, CA: O'Reilly & Associates, Inc. p 79.

[2] [13] Ibid. p. 177.

2. AOP.NET ASPECT MODEL

AspectJ supports two types of cross cutting implementations. The first called dynamic cross cutting makes it possible to modify the behavior of implementations by changing the execution at join points in different ways. The second, called static cross cutting allows aspects to extend the type signature of components, adding fields or inherited interfaces to a class for example [2]. AOP.NET only supports dynamic cross cutting, but it may be possible to support static cross cutting in future work.

In AspectJ, the execution of the program refers to the composed program, whereas in this system, it refers to the execution of the uncomposed program. While it is possible to apply advice to advice in AspectJ, AOP.NET currently does not support this.

AspectJ supports a wide range of join points. These include but are not limited to the call of a method or constructor, the execution of a method, object initialization, field get and set and others. Currently our system supports only method execution join points.

AspectJ advice may be applied in many ways to a join point. It can be applied *before*, *after* or around a join point, with some special cases on *after* advice related to how a method returns or whether an exception is thrown [2]. Our system only supports before and after advice.

While AspectJ includes a comprehensive and easy to understand pointcut language, our prototype uses regular expressions to match pointcut expressions with method execution join points. It is therefore difficult or impossible to express some pointcuts that are straightforward to express in AspectJ with the prototype. One approach to addressing this would be to create a family of Advice attributes for different types of pointcuts. A similar approach is used by PROSE by providing a set of pre-defined libraries for cross cutting that can then be extended [6].

For comparison, Figure 1 is a simple aspect definition written in AspectJ; a similar aspect is written using C# for use with AOP.NET in Figure 2.

```
public aspect LoggingAspect {

    before() : execution(* *.Add (..) )
    {
      System.out.println(
        "LogAdvice1 called");
    }

    after() : execution(* *.Divide (..) )
    {
      System.out.println(
        "LogAdvice2 called");
    }
}
```

Figure 1. Simple AspectJ logging aspect definition

2.1 Aspects in C#

This section discusses how aspects, their behavior in advice, and pointcuts are expressed in C# using existing language constructs for AOP.NET.

Since C# and other CLR languages such as VB.NET are GP languages, there is no specified way to express aspects or cross cutting. In AOP.NET aspects are expressed in a conventional C# class definitions. To provide services common to all aspects in the system, aspects inherit from an Aspect base class. With a common base class, the weaver can more easily identify aspects if

they inherit from a common base class using the .NET reflection APIs. In a future system, the Aspect base class could provide additional services common to all aspects; for example, it could supply a reference to the current component affected by the advice.

Advice is expressed as methods within an aspect class. The advice methods contain code that is used to modify functional code at join points specified by pointcuts.

To distinguish advice from non-advice methods in an aspect, a custom attribute called `Advice` is used. This attribute also provides information to the weaver about the pointcuts, and about how to apply the advice to join points defined by the pointcuts. Figure 2 shows a typical aspect including `Advice` attributes that specifies two separate pointcuts for two different advice definitions. Note that the current prototypes do not support advice parameters, but it should be possible to extend the system to do so in a future implementation.

```
public class LoggingAspect : Aspect
{
    [Advice(Type = AdviceType.before,
       DirectType = PointcutType.execution,
       DirectPointcut
          = @"[\w\. ]*:[\w ]* Add([\w, ]*)")]
    public static void LogAdvice1()
    {
      System.Console.WriteLine(
        "LogAdvice1 advice called");
    }
    [Advice(Type = AdviceType.after,
       DirectType = PointcutType.execution,
       DirectPointcut = "^CalculatorLibrary")]
    public static void LogAdvice2()
    {
      System.Console.WriteLine(
        "LogAdvice2 advice called");
    }
}
```

Figure 2. Simple AOP.NET C# logging aspect definition

In Figure 2, there are two advice declarations within `LoggingAspect`: `LogAdvice1` and `LogAdvice2`, as indicated by the `Advice` custom attribute associated with each. The `DirectPointcut` parameter in the `Advice` attribute for the `LogAdvice1` advice specifies that this advice will be applied to all methods called `Add` in any type using a regular expression. The `DirectPointcut` for the `LogAdvice2` specifies that this advice will be applied to all methods in the `CalculatorLibrary` namespace. The `Type` parameter in an `Advice` attribute tells the weaver to apply the advice in a certain way. `LogAdvice1` is applied after an `Add` method is executed, whereas `LogAdvice2` is applied after methods that match.

The `Advice` custom attribute contains four fields and a method as shown in Figure 3. The `Type` field of the `Advice` attribute specifies whether the advice should be applied before or after the associated pointcut. The `DirectPointcut` string field is used when the pointcut associated with the advice is specified in the advice attribute itself. `DirectType` specifies the join point type. Only method execute join points are currently supported. A named pointcut is specified using the `PointcutRef` attribute described later.

Note that the `Advice` custom attribute itself has attributes associated with it. The `AttributeUsage` attribute controls how the custom `Advice` attribute should be treated by the compiler, that is, how it can be applied to various targets such as methods, fields, classes, etc. In this case advice can only be applied to methods, and more than one advice attribute can be associated with a single method.

```
public enum AdviceType
{
    before,     // before method execution
    after,      // after method execution
    around      // not supported yet
}

// Direct named pointcuts only
[AttributeUsage(AttributeTargets.Method,
AllowMultiple = true)]
public class Advice : System.Attribute
{
    public AdviceType Type; // before, after
    public string PointcutRef;
    public PointcutType DirectType
    public string DirectPointcut;

    virtual public bool Match(
      AdviceType adviceType,
      string methodSignature)
    {
      if (Type == adviceType)
      {
        if (Regex.IsMatch(
            methodSignature, DirectPointcut))
          return true;
      }
      return false;
    }
}
```

Figure 3. Advice custom attribute definition

To express cross cutting, a way to specifying pointcuts needs to be defined and embedded into aspect declarations. The approach used here is to express pointcuts as regular expressions. These expressions are embedded in strings that are then matched against the signature of candidate in the code at weave time. In a future system a more comprehensive pointcut language could be developed.

A language-independent method signature is defined in AOP.NET by leveraging the Common Type System (CTS)[3] of the CLR so that pointcut regular expressions can be used in any .NET language. The template for the of a method in a namespaces is as follows:

`<namespace>.type: <method-signature>`

For example, the signature of the `Add` method in the `CalculatorLibrary` namespace, `Calculator` class is as follows:

```
CalculatorLibrary.Calculator:Int32
Add(Int32, Int32);
```

The `Advice` attribute contains a method called `Match` which uses the regular expression in the `DirectPointcut` string to check for matches with the signature of a supplied method corresponding to a possible join point. By creating a family of `Advice` attributes using inheritance where a child of `Advice` implements the `Match` method differently, other join points may

[3] [13] Ibid. p. 9.

be supported. During the weaving procedure, AOP.NET could look for `Advice` attributes of a certain type related to a possible join point, and then call the `Match` method of this attribute to see if the associated advice applies.

To support named pointcuts associated with more than one advice in an aspect, pointcuts can be expressed using string fields with an associated custom attributes called a `Pointcut` and then referred to by the `PointcutRef` `Advice` attribute field.

Figure 4 shows a named pointcut called `Write` that applies to `Write` methods in any class called `Terminal` specified in the regular expression embedded in the `WritePointcut` string. The `Write` pointcut is used to apply different advice before and after the method execution of a `Terminal` type's `Write` method (`LogAdvice1` and `LogAdvice2` respectively).

```
public class LoggingAspect : Aspect
{
    public LoggingAspect() {}

    [Pointcut(Name = "Write")]
    private const string WritePointcut
      = @"[\w\. ]*.Terminal:[\w ]*
        Write([\w, ]*)";

    [Advice(
      PointcutRef =
      "Write"
      Execute = AdviceType.before)]
    public static void LogAdvice1()
    {
      Log.LogEvent("LogAdvice1 called");
    }
    [Advice(
      PointcutRef =
      "Write"
      Execute = AdviceType.after)]
    public static void LogAdvice2()
    {
      Log.LogEvent("LogAdvice2 called");
    }
}
```

Figure 4. PointcutRef example

This is a simple example, but one could extend it to include support for additional named parameters to the attributes, enhancing the pointcut language embedded in strings, or by creating a family of `Pointcut` or `Advice` attributes as mentioned previously.

3. WEAVING ASPECTS

AOP.NET currently composes a single component type with a single aspect. Although the current prototype is limited to single components and aspects, it should be straightforward to support collections of components and aspects in future work. Both the dynamic and static weaver tool use the procedure described in this section.

The weaver takes as input a component type and an aspect type. Without modifying the intermediate language code of either, it then composes the two by generating a proxy object that delegates to aspect code or the original component appropriately as shown in Figure 5. This is similar to the technique used by CLAW [5], ACGEN tool [14], Schult et al. [11] and AOP/ST [10]. This technique was used since the .NET Framework Class Library (FCL) does not include a library for retrieving the Common Intermediate Language (CIL) byte codes from types, only meta data. Without access to the CIL bytecodes, it was impossible to existing code into new components. The Weave.NET [3] system composes aspects with components inline by leveraging a CIL parser library [15]. Future work on AOP.NET may leverage this library or a similar one to provide inline weaving.

One advantage of the proxy approach is that it is possible to identify and trace functional and aspect code in woven code using source level debuggers. One disadvantage is the performance loss associated with delegated method calls to the aspects and shadowed objects. Since the function component is not modified, the aspect will not affect the methods called by the component itself, only those outside the component.

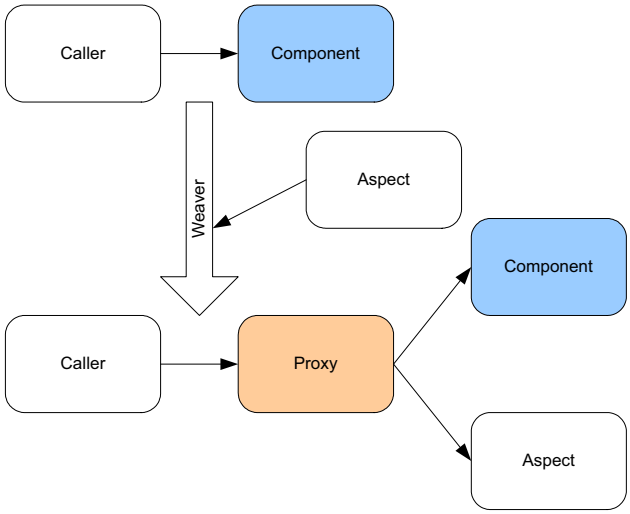

Figure 5. Weaving process

The main procedure of AOP.NET takes six parameters as shown below.

```
public static Type Weave(
       Type componentType,
       Type aspectType,
       string outputAssembly,
       string moduleName,
       string namePrefix,
       bool isStatic)
```

The first and second parameter specifies the type of the component and aspect respectively. The following three parameters: `outputAssembly`, `moduleName` and `namePrefix` specify the output assembly and module name for the dynamic or static proxy object that is generated during composition. The `namePrefix` is an optional name appended to the beginning of the composed object namespace. This was used during development to differentiate the proxy object from the component when both were accessible from the test application. The last parameter specifies whether to save the generated proxy in a assembly library (.DLL) file (set to true by the static weaver tool) for use by the application during run time.

The Reflection library [16] of the FCL was used to retrieve type information from both the component and the aspect library assemblies. An assembly is a basic unit of compilation in .NET and contains modules, which contain types. Types contain

members such as methods, and fields. The reflection library provides objects that encapsulate all of these allowing them to be instantiated, fields to be accessed and methods called. The Reflection library can also be used to query types for custom attributes associated with various program elements [17].

The Reflection.Emit library provides classes such as `AssemblyBuilder`, `ModuleBuilder`, `TypeBuilder` and `ILGenerator` to create types at runtime. These classes were used by the weaver to create the proxy object. The weaver creates an assembly and a module using `AssemblyBuilder` and `ModuleBuilder`. It then creates a type with the same signature as the original (uncomposed) functional component. Within the proxy, it creates a new field called `_refObject` that holds a reference to the uncomposed functional component. In the constructor, it generates IL code to call the constructor of the original type, and assign this object to the `_refObject` field. Sample generated CIL code for a proxy constructor is shown in Figure 6. Each method of the proxy is then generated using `MethodBuilder` and emitting CIL code to call either the component or the advice appropriately. CIL code generated for an Add method with advice applied before and after execution is shown in Figure 7.

```
.method public specialname rtspecialname
        instance void  .ctor() cil managed
{
  // Code size       18 (0x12)
  .maxstack  4
  IL_0000:  ldarg.0
  IL_0001:  call     instance void
    [mscorlib]System.Object::.ctor()
  IL_0006:  ldarg.0
  IL_0007:  newobj   instance void
    [CalculatorLibrary]
      CalculatorLibrary.Calculator::.ctor()
  IL_000c:  stfld    class
    [CalculatorLibrary]
      CalculatorLibrary.Calculator
      CalculatorLibrary.Calculator::_realObject
  IL_0011:  ret
} // end of method Calculator::.ctor
```

Figure 6. CIL for proxy constructor

The pseudocode for the weaver is shown in Figure 9. To summarize, the weaver creates a dynamic proxy type and then walks through each method in the supplied component. It compares each method against the list of advice attributes found in the aspect definition using the reflection libraries, and emits calls to the advice and the original component in the new proxy appropriately.

If access to the intermediate language byte codes of the component and aspect were available, these could be copied to a new assembly that integrates aspect and component code in the same component as shown in Figure 8 using the same basic approach. In the weaving procedure, calls to the functional component would be replaced with the insertion of original component CIL code. By scanning the component CIL for method calls, and inserting advice CIL code directly, the weaver could be extended to support method call join points as well.

As mentioned, AOP.NET uses the RegularExpressions .NET library [18] in the `Advice` attribute `Match` method shown in Figure 3 to match regular expressions in the attribute with method signatures. By encapsulating the method for matching advice to join points in the `Advice` attribute, rather than the weaver procedure itself, it should be possible to support more pointcuts without modifications to the weaver, by sub-classing the Advice attribute.

```
.method public hidebysig newslot virtual
        instance int32  Add(int32 A_1,
                            int32 A_2) cil
managed
{
  // Code size       28 (0x1c)
  .maxstack  4
  .locals init (int32 V_0)
  IL_0000:  call       void
    [Logger]Logger.LoggingAspect::LogAdvice1()
  IL_0005:  ldarg.0
  IL_0006:  ldfld      class
    [CalculatorLibrary]
      CalculatorLibrary.Calculator
      CalculatorLibrary.Calculator::_realObject
  IL_000b:  ldarg.1
  IL_000c:  ldarg.2
  IL_000d:  callvirt   instance int32
    [CalculatorLibrary]
      CalculatorLibrary.Calculator::Add(
        int32, int32)
  IL_0012:  stloc.0
  IL_0013:  call       void
    [Logger]Logger.LoggingAspect::LogAdvice2()
  IL_0018:  br.s       IL_001a
  IL_001a:  ldloc.0
  IL_001b:  ret
} // end of method Calculator::Add
```

Figure 7. CIL for proxy Add method with calls to advice before and after calls to functional component

For example, to support embedded XML fragments for compatibility with XML-based pointcut definitions used in Weave.NET [3], one could parse the XML in a new `Match` method. With more work perhaps a pointcut language similar to AspectJ could be implemented, or new criteria for matching advice to join points can be developed. One could create an Advice attribute that weaves an aspect with a component based on current run time conditions such as a set backup time by subclassing the Advice attribute.

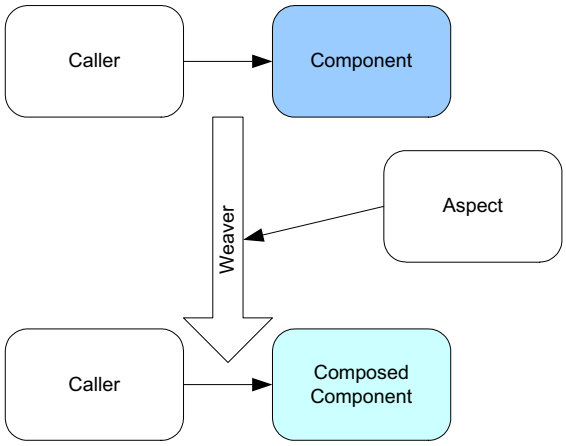

Figure 8. Weaving inline, without the use of a proxy

3.1 Static Weaving

AOP.NET's static weaving tool simply takes its parameters specifying the component and aspect assemblies and types from the command line and then applies the weaving procedure to these files, creating a new assembly file containing the proxy.

For example:

```
c:>weave CalculatorLibrary.dll
CalculatorLibrary.Calculator Logger.dll
Logger.LoggingAspect AspectProxy.dll
```

This causes the tool to generate a new assembly containing a type called CalculatorLibrary.Calculator within the AspectProxy DLL.

After running the tool, the application then must be configured to refer to the proxy DLL (in this case AspectProxy.dll) rather than the uncomposed component (CalculatorLibrary.dll).

```
Type composedType WeaverEngine (Type component,
Type aspect)

Begin WeaverEngine
    create a new assembly for the proxy
    create proxy module
    create proxy type with same signature as
      component
    create _realObject field in the proxy type
      to delegate to component

    generate constructor code for the type
    generate code to instantiate _realObject
      reference to component

    for each method in the component (loop)
      emit same method signature in the proxy

      query aspect type for Advice attributes
      that apply before method execution
      if pointcut matches current method signature
        emit call to Aspect advice into the proxy

      emit delegation call to the unwovenClass

      query aspect type for Advice attributes that
      apply after method execution
      if pointcut matches current method signature
        emit call to Aspect advice into the proxy
      emit proxy method return code if needed.

    end for loop
    emit closing code for the type, module,
      and assembly

end WeaverEngine
```

Figure 9. Weaver engine pseudocode

3.2 Dynamic Weaving

The goal of a dynamic weaver is to transparently support the weaving of components at runtime as they are needed. To do this, components must be weaved when assemblies to be affected by aspects are referenced by other functional components of the system.

The prototype dynamic weaving system leverages the CLR support for application domains. Application domains are a unit of isolation for the CLR and run inside a process. Most applications never create an application domain explicitly, since one is created for them, but to support dynamic weaving, a CLR host that transparently weaves components on demand was needed. A host is responsible creating an application domain for executing code within that domain [19]. With a custom host, it is also possible to receive events from the application domain when an assembly such as an executable resolves a reference to another assembly such as a library [20].

By plugging in the weaving procedure into the event handler for resolving assemblies, the host can dynamically weave components with aspects.

The prototype code for the custom CLR weaving host is shown in Figure 10 and Figure 11. The main function (Figure 10) creates a new `AppDomain` object which encapsulates a CLR host. A method called `dom_AssemblyResolve` (Figure 11) is then registered with the new domain object event called `AssemblyResolve` to be called when an assembly referenced during execution cannot be found.

This method (`dom_AssemblyResolve`) will call the weaving procedure to dynamically generate a new composed type when required. Currently the feasibility prototype is hard coded to only support one specific type (`CalculatorLibrary`).

In the prototype system, to cause the `AssemblyResolve` event to fire, the original components are moved to a directory where the new host cannot find it, that is, outside of its base directory. In this case, the components and the aspects are both moved to a sub directory called "weave". The `WeaveAssembly` method then accesses the component and weaves libraries in this directory, composes them as described previously, and returns the assembly to the host.

```
static
void Main(string[] args)
{
    // create a new application domain
    AppDomainSetup setup = new AppDomainSetup();
    setup.ApplicationBase =
      AppDomain.CurrentDomain.BaseDirectory;
    setup.ApplicationName = "DynamicWeaver";
    AppDomain dom = AppDomain.CreateDomain(
        "HelloWorldApp", null, setup);
    dom.AssemblyResolve += new
        ResolveEventHandler(dom_AssemblyResolve);
    dom.ExecuteAssembly("HelloWorld.exe");
    AppDomain.Unload(dom);
}
```

Figure 10. Dyanamic weaver CLR host main method

```
public static Assembly dom_AssemblyResolve(
    object sender,
    ResolveEventArgs args)
{

    Type newType;

    if (Regex.IsMatch(
        args.Name, @"^CalculatorLibrary"))
    {
      newType = Weaver.WeaveAssembly(
        "weave\\CalculatorLibrary.dll",
        "CalculatorLibrary.Calculator",
        "weave\\Logger.dll",
        "Logger.LoggingAspect",
        "AspectProxy.dll",
        "AspectProxy",
        false);
      return newType.Assembly;
    }
    else
    {
      // we don't know how to weave this assembly
      return null;
    }
}
```

Figure 11. Dyanamic weaver AssemblyResolve event handler

4. PROTOTYPE EVALUATION

This section describes an example execution of AOP.NET where a component and an aspect are combined dynamically.

To test the dynamic weaver, a component called `Calculator` was created to do arithmetic on integers as shown in Figure 15. A simple aspect was created to log the execution of certain methods shown in Figure 2. The test application used to exercise the component is shown in Figure 12. When executed, the application produced the output shown in Figure 13 to the console.

The `CalculatorLibrary` assembly was then moved to the *weave* sub directory to trigger component weaving and then executed. The output from this execution is shown in Figure 14. Note that the weaver logs some diagnostic output from the dynamic host (not shown) before the code runs indicated at ">>>Application started".

The dynamic weaver prototype composed the `CalculatorLibrary` with the logger as expected without changes to the source code of the executable. The advice `LogAdvice1` was added before the `Add` method, and `LogAdvice2` was added to every method of the `Calculator` component.

```
class HelloWorld
    {
    [STAThread]
    static void Main(string[] args)
    {
      System.Console.Out.WriteLine(
        ">>>Application started");
      CalculatorLibrary.Calculator calc
        = new CalculatorLibrary.Calculator();
      Console.WriteLine(
        "In the application domain: " +
         AppDomain.CurrentDomain.FriendlyName);
      int answer = calc.Add(3,5);
      answer = calc.Divide(answer, 4);
      answer = calc.Multiply(answer, 10);
      answer = calc.Subtract(answer, 10);
      System.Console.Out.WriteLine(
        "Hit enter to quit");
      System.Console.In.ReadLine();
    }
}
```

Figure 12. Test Application

```
>>>Application started
In the application domain: HelloWorld.exe
Add 3 + 5 = 8
Divide 8 / 4 = 2
Multiply 2 * 10 = 20
Subtract 20 - 10 = 10
Hit enter to quit
```

Figure 13. Application output

```
>>>Application started
In the application domain: HelloWorldApp
LogAdvice1 advice called
Add 3 + 5 = 8
LogAdvice2 advice called
Divide 8 / 4 = 2
LogAdvice2 advice called
Multiply 2 * 10 = 20
LogAdvice2 advice called
Subtract 20 - 10 = 10
LogAdvice2 advice called
Hit enter to quit
```

Figure 14. Weaved application output

```
public class Calculator
{
    public Calculator()
    {
    }

    public virtual int Add(int x, int y)
    {
      System.Console.Out.WriteLine(
        "Add {0} + {1} = {2}",x,y,x+y);
      return x+y;
    }

    public virtual int Subtract(int x, int y)
    {
      System.Console.Out.WriteLine(
        "Subtract {0} - {1} = {2}",x,y,x-y);
      return x-y;
    }

    public virtual int Multiply(int x, int y)
    {
      Add(3,2);
      System.Console.Out.WriteLine(
        "Multiply {0} * {1} = {2}",x,y,x*y);
      return x*y;
    }

    public virtual int Divide(int x, int y)
    {
      System.Console.Out.WriteLine(
        "Divide {0} / {1} = {2}",x,y,x/y);
      return x/y;
    }
}
```

Figure 15. Calculator component

5. RELATED WORK

The CAMEO project [1] extended the C# compiler supplied by Microsoft to add language extensions for AOP similar to those in AspectJ. To do this it takes as input the extended C# language, XML aspect definition files and outputs standard C# code which is compiled by the standard compiler. Unlike AOP.NET, CAMEO is a static weaver only, is C# language specific and uses an outside XML specification for some cross cutting statements.

Schultz and Polze [21] describe an aspect-specific tool that adds fault tolerance to .NET components using aspect oriented techniques. They use custom attributes using the existing C# language and automatically create proxy objects as AOP.NET does. The tools support static weaving only.

In another paper [11] Schultz and Polze describe a more general purpose system that supports what they call dynamic aspect weaving. To weave components with aspects the application creates a composed component by calling a weaving library. Unlike AOP.NET, clients of composed components are aware that a component may be composed by this library, and are therefore not oblivious to aspect composition, often cited as a defining characteristic of aspect oriented systems [12]. Also since woven components inherit from the functional components, advice can only apply to virtual methods, a limitation not shared by AOP.NET.

LOOM.NET [22] is a set of tools based on Schultz and Polze previous work as described. In the LOOM.NET system cross cutting statements are managed by a GUI.

Weave.NET [3] is a weaving tool that takes a component, an aspect and an XML file as input. The XML file contains the cross

cutting statements linking aspect advice in C# classes to components. Weave.NET is a load-time weaver, defined here as weaving after the components and aspects are compiled and when the weaver library is called by an application. Like LOOM.NET, Weave.NET is not an "oblivious" dynamic weaver since clients of the dynamic weaver are aware that components may be composed [23]. Static CIL weaving looks straightforward to implement with Weave.NET, but it is not clearly supported. Unlike our system Weave.NET does inline weaving, creating a new assembly from components and aspects by retrieving intermediate language code from assembly files using a CIL parser library.

CLAW [5] is a .NET a dynamic weaver implemented in C++ and using the Common Object Model (COM) to extend the CLR by linking in to the profiling mechanism supplied with the runtime. With this mechanism, it is possible to add a new method at runtime, inject new CIL code at runtime for an existing method body, relocate methods from one type to another, and recompile existing methods. Like AOP.NET, CLAW only supports execution join points, and creates "shadow proxies" at run time. Unlike AOP.NET, it uses XML for pointcut to advice mappings so that cross cutting statements are outside of the source language and is implemented in C++.

ACGEN [14] is an AOP-related technology that leverages the use of the Reflection.Emit library and attributes in C# to compose caching functionality with existing objects to improve the overall performance of certain remote procedure calls. Attributes associated with component methods cause ACGEN to add caching to that method when the tool is applied. Depending on the caching strategy and parameters supplied to the tool the proxy object either calls a component for caching, or the original object. The caching component can be associated with any specified object using this tool. The tool can use attributes, in this case associated with the target object, to signify that certain methods should be cached. In some ways, this tool can be considered to be a very limited aspect weaver that only supports a single aspect type described by the caching interface. Functional objects are not oblivious to the caching aspect since they contain attributes specifying which methods should be cached.

Shulka et al [24] describe an approach to incorporating AOP into .NET using the .NET libraries to intercept method calls to a component. Weaving tools or dynamic weaving systems are not needed since aspects can associate themselves with objects using standard object oriented programming techniques. However with this approach, functional components cannot be oblivious to aspects that may be applied by participating in the design pattern. Another problem with this approach is that the aspect-programming model and support is often limited by the patterns used, for example it may be easy to support method execution, but not method call join points. Unfortunately the use of composition patterns in itself can become a cross cutting concern throughout the system.

AOP/ST [10] is a dynamic aspect weaver for VisualWorks Smalltalk. Like AOP.NET, AOP/ST does not modify the source code or the byte codes of functional components. Rather than creating a proxy with the same signature as the component it uses inheritance to add aspect code to classes and every aspect is added in a separate subclass of the component. Like AOP.NET, AOP/ST is limited in the types of join points that can be supported by this approach.

PROSE [6, 9] is a dynamic weaver for Java. Aspects and cross cutting statements are expressed without changes to the component language as in AOP.NET. While AOP.NET uses attributes, PROSE uses anonymous classes to embed advice and pointcuts together in aspects. While the weaver engine only supports weaving at the run time, PROSE supports aspect weaving, unweaving and replacement at run time. In a similar approach to CLAW in .NET, the PROSE implementation leverages the debugging and profiling interfaces of the Java virtual machine.

6. CONCLUSIONS

This work combines features of previous aspect weaving systems for .NET and C#. It also borrows some of approaches used in dynamic weaving systems for Smalltalk and Java. Specifically it leverages the use of custom attributes to avoid separate cross cutting statement files using XML for example, and implements a transparent or oblivious dynamic weaver host. This system illustrates the feasibility of a dynamic aspect oriented programming system implemented completely in C#, using existing language features for C# and .NET.

This work describes a prototype for feasibility testing. Its development raised a number of technical questions, required testing, and future research directions.

In future implementations I hope to address the limitations of the proxy approach to composing components with aspects. Although a proxy approach has some advantages, such as source level debugging of aspects and advice, it also has some limitations related to the join points that can be supported. Another limitation of the current prototype is that if a component references itself, it does not reference the composed component, but the original uncomposed one. This could be addressed by composing aspects with components inline in a similar way to Weave.NET for example.

Additional work involves the testing of the feasibility prototype with other .NET languages such as VB.NET and J#. Since AOP.NET does not leverage any C# specific features, it is expected to be .NET language independent, but this needs verification. A number of limitations to the prototype itself, related to scalability, especially supporting multiple component assemblies and aspects, and supporting executable assemblies as well as libraries needs to be addressed.

Finally, the aspect oriented programming model should be extended to support static cross cutting, more join points, possibly even supporting a wider range of join points than that supported by AspectJ using a family of pointcut or advice attributes.

7. REFERENCES

[1] Prasad, M.D. and Chaudhary, B.D. AOP Support for C#. *in AOSD Workshop on Aspects, Components and Patterns for Infrastructure Software 2003*. Boston. March 17, 2003. pp. 49-53.

[2] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W.G. An Overview of AspectJ. *in ECOOP 2001*. pp. 327-355.

[3] Lafferty, D. and Cahill, V. Language-Independent Aspect-Oriented Programming. *in OOPSLA 2003*. Anaheim, CA. pp. 1-12.

[4] Ossher, H. and Tarr, P. Multi-Dimensional Separation of Concerns and The Hyperspace Approach. *in Symposium on Software Architectures and Component Technology 2000*.

[5] Lam, J. *Cross Language Aspect Weaving (CLAW) Power Point Presentation*. Retrieved November 10 from http://www.iunknown.com/000092.html

[6] Popovici, A., Gross, T., and Alonso, G. Dynamic weaving for aspect oriented programming. *in Proceedings of the 1st International Conference on Aspect-Oriented Software Development 2002*. Enschede, The Netherlands. April 2002. pp. 141-147.

[7] Kiczales, G., Lamping, J., Menhdhekar, A., Lopes, C., Loingtier, J., and Irwin, J. Aspect-Oriented Programming. *in ECOOP 1997*. pp. 220-242.

[8] Lai, A., Murphy, G.C., and Walker, R.J. Separating Concerns with Hyper/J: An Experience Report. *in International Conference on Software Engineering 2000*. Limerick, Ireleand. June 6. pp. 79-91.

[9] Popovici, A., Alonso, G., and Gross, T. Just-In-Time Aspects: Efficient Dynamic Weaving for Java. *in Proceedings of the 2nd International Conference on Aspect-Oriented Software Development 2003*. Boston. March 2003. pp. 100-109.

[10] Bollert, K. On weaving aspects. *in International Workshop on Aspect-Oriented Programming ECOOP'99 1999*.

[11] Schult, W. and Polze, A. Dynamic Aspect-Weaving with .NET. *in Workshop zur Beherrschung nicht-funktionaler Eigenschaften in Betriebssystemen und Verteilten Systemen 2002*. Berlin, Germany.

[12] Filman, R.E. and Friedman, D.P. Aspect-Oriented Programming is Quantification and Obliviousness. *in Workshop on Advanced Separation of Concerns, OOPSLA 2000*. Minneapolis. October 2000.

[13] Drayton, P., Albahari, B., and Neward, T. *C# in a Nutshell*. 2002, Sebastopol, CA: O'Reilly & Associates, Inc.

[14] Guest, S. *Using Reflection Emit to Cache .NET Assemblies*. Retrieved 2003 from http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/rflemitcache.asp

[15] *CLIFileReader Demo Demonstrates use of CLIFileReader library and System.Reflection.Emit API*. Retrieved November 11 from http://dotnet.di.unipi.it/MultipleContentView.aspx?code=155

[16] Microsoft. *Reflection Overview*. Microsoft, Corp. Retrieved November 14 from http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconreflectionoverview.asp

[17] *Accessing Custom Attributes*. Retrieved from http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconreflectionoverview.asp

[18] Microsoft. *System.Text.RegularExpressions Namespace*. Microsoft, Corp. Retrieved November 29 from http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfSystemTextRegularExpressions.asp

[19] Microsoft. *Using App Domains*. Microsoft, Corp. Retrieved November 14 from http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconusingapplicationdomains.asp

[20] Pratschner, S. *Microsoft .NET: Implement a Custom Common Language Runtime Host for Your Managed App*. Microsoft, Corp. Retrieved November 14 from http://msdn.microsoft.com/msdnmag/issues/01/03/clr/default.aspx

[21] Schult, W. and Polze, A. Aspect Oriented Programming with C# and .NET. *in International Symposium on Object-oriented Real-time distributed Computing (ISORC) 2002*. pp. 241-248.

[22] Schult, W. and Polze, A. *Welcome to Loom.net*. Hasso - Plattner - Institute at University of Potsdam Operating Systems and Middleware Group. Retrieved October 24 from http://www.dcl.hpi.uni-potsdam.de/cms/research/loom/

[23] Lafferty, D. Language-independent Aspect Oriented Programming UBC Presentation. November 5, 2003. Vancouver, BC, Canada. Presentation slides and related question about how Weave.NET was used clarified load time weaving as the use of a weaver library.

[24] Shukla, D., Fell, S., and Sells, C. *Aspect-Oriented Programming Enables Better Code Encapsulation and Reuse*. Microsoft, Corp. Retrieved October 23 from http://msdn.microsoft.com/msdnmag/issues/02/03/AOP/default.aspx