Language-Independent Aspect-Oriented Programming

Donal Lafferty
Donal.Lafferty@cs.tcd.ie

Vinny Cahill
Vinny.Cahill@cs.tcd.ie

Distributed Systems Group
Department of Computer Science
Trinity College Dublin

ABSTRACT

The term aspect-oriented programming (AOP) has come to describe the set of programming mechanisms developed specifically to express crosscutting concerns. Since crosscutting concerns cannot be properly modularized within object-oriented programming, they are expressed as *aspects* and are composed, or *woven*, with traditionally encapsulated functionality referred to as *components*.

Many AOP models exist, but their implementations are typically coupled with a single language. To allow weaving of existing components with aspects written in the language of choice, AOP requires a *language-independent* tool.

This paper presents Weave.NET, a load-time weaver that allows aspects and components to be written in a variety of languages and freely intermixed. Weave.NET relies on XML to specify aspect bindings and standardized Common Language Infrastructure to avoid coupling aspects or components with a particular language.

By demonstrating language-independence, Weave.NET provides a migration path to the AOP paradigm by preserving existing developer knowledge, tools, and software components. The tool's capabilities are demonstrated with logging aspects written in and applied to Visual Basic and C# components.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications - Language Constructs and Features – *multiparadigm languages;* D.1 [**Software**]: Programming Techniques – Aspect-Oriented Programming

General Terms

Design, Experimentation, Standardization, Languages.

Keywords

Aspect-oriented programming, Weave.NET, Common Language Infrastructure, language-independence.

1. INTRODUCTION

Crosscutting concerns are "properties or areas of interest" [8] that normally defy object-oriented (OO) modelling, because the deployment of functionality to support them does not align with the composition operations available in an object model [3]. Even conceptually simple crosscutting concerns, such as tracing during debugging and synchronization, lead to tangling, in which code statements addressing the crosscutting concern become interlaced with those addressing other concerns within the application.

"To ameliorate this problem, AOP offers aspects: mechanisms beyond subroutines and inheritance for localizing the expression of a crosscutting concern." [8] An *aspect* [15] provides a unit of encapsulation that couples the behaviour of a crosscutting concern with a join point specification that details where in component code the behaviour is to be applied. In the context of AOP, *components* [15] correspond to units of well-encapsulated behaviour be it source code or binaries. The aspects and components of an application are composed, or *woven*, to produce a single program.

The principle AOP technologies [8], express a unique view of AOP in terms of their aspect model. The Demeter group [23] focuses on the succinct expression of object graph traversals to simplify programming concerns that crosscut object hierarchies. Multi-Dimensional Separation of Concerns (MDSOC) research [33] breaks up the different concerns that an object must address into separate programming tasks by providing special composition operators that compose classes from a set of partially complete behaviours called *hyperslices*. The Composition Filters (CF) model [1] exposes message passing between objects for the purposes of writing behaviour that requires high level coordination amongst objects. Finally, the AspectJ project [13] augments the Java object model with an explicit aspect construct that provides mechanisms to specify and manipulate the control-flow of a program.

Unfortunately, none of these AOP technologies is *language independent*, in that they do little to present their composition model as decoupled from source code, or demonstrate in their implementation strategies the ability to intermix aspects and components written in a variety of languages. AspectJ [14] views aspect and component implementation as a Java coding exercise. Aspects are only present in source code, and after compilation they are no longer discernable. Extending this aspect model to other languages is left to researchers outside the AspectJ team, and no provision is made to allow reuse of aspects across different languages. Demeter's aspect model is based around object graph traversal, which exists in most, if not all, object models. The latest Demeter technology, DJ [22], focuses on supporting graph traversal within the context of Java. Although DJ is based on a library, it cannot be easily decoupled from Java. As with previous Demeter implementations [23], the work does not present evidence of allowing aspects written within one language to operate within the context of another. MDSOC's hyperslice work focuses on software evolution. Its implementation focuses on binary composition [24], and in its current realisation, Hyper/J, composition is independent of source. However, composed behaviours must be written in Java, i.e.

aspect and component behaviour cannot be written in a variety of languages. The CF model is described in terms of how it is applied to an object model, but implementation strategies identified in [2] are compilers that address filters in one language or another. CF research does not look at how filters written in one language could be applied to objects of another. Moreover, use of a compiler implies that component source is required to affix filters correctly. The crux of the matter is that with some effort each aspect model could be interpreted as supporting language independence, but in practice none succeeds in making this mapping.

The status quo of allowing AOP to be tied to a particular language is both paradoxical and wasteful. Speaking philosophically, the motivation of AOP should be to break the "tyranny of the dominant decomposition" paradigm [33] of a particular programming language. Producing a system in which aspects or components are all written in only one language would seem to reinforce this tyranny. Certainly, such an approach would do little to improve the reusability of aspects as compared to one in which aspects written in one language could be applied to code written in another. Reuse should not be viewed as a concern specifically of software components. Programmer knowledge needs to be preserved by allowing current skills to be reapplied in new paradigms. It would seem absurd that having mastered writing components in a particular language that this skill would be discarded. However, coupling implementation language with AOP composition does just that. The same is true of development tools for which current approaches provide no migration path to AOP.

Weave.NET exploits the multi-language support of Microsoft's Common Language Infrastructure (CLI) [7], developed for the .NET Framework, to provide a solution for these problems. Weave.NET is a language-independent aspect weaver that avoids coupling aspects or components with a particular language. Weave.NET performs binary-level composition according to an XML-based composition script, meaning that the composition specification is not written in terms of, or using extensions to, a particular programming language. The script is applied at load-time, well after component and aspect behaviour is compiled to binary form. As such, the weaver is oblivious as to the implementation language of these behaviours.

The programming model of Weave.NET is derived from AspectJ and its architecture based on the CLI. The actual aspect model implemented in Weave.NET is a generalisation of that of AspectJ [34]. In particular, Weave.NET shares AspectJ's join point model and aims to make available the same pointcut and advice abstractions. AspectJ uses pointcuts to specify sets of join points, where one of these sets constitutes a crosscut. Advice affects an application's implementation by selecting behaviour that executes relative to, or instead of, the join points of a pointcut. Weave.NET supports both, albeit via an XML schema rather than the language extensions advocated by AspectJ. The format for binaries woven by Weave.NET is provided by the CLI. The CLI mandates that *extender*[1] code generation tools target the CLI's common intermediate language (CIL, or simply IL), which is analogous to Java byte-code. IL alone does not contain sufficient information to identify all join points, but with the CLI's metadata standard [21] it is possible to do so without the need for the source code used to generate a type. Moreover, the CLI's common type system (CTS) allows aspect behaviour written in any language to be applied to any type written in any other language.

With a subset of advice and pointcut statements, Weave.NET has been able to demonstrate language-independent AOP. That is, we have been able to weave aspects whose behaviour is written in one language with components written in another. Specifically, test cases have involved applying logging aspects written in C# and Visual Basic to an I/O library written in C# and a client application of the I/O library written in Visual Basic.

The rest of the paper is organized as follows. Section 2 explains the Weave.NET programming model. Section 3 relates elements of our aspect model to the corresponding CLI architectural element. Section 4 presents Weave.NET's XML weaving specification schema. Section 5 discusses the weaver's architecture. Section 6 provides an initial assessment of Weave.NET's language-independence and performance. Section 7 discusses related work. Finally, section 8 concludes with a discussion of our findings and a roadmap to future work.

2. PROGRAMMING MODEL

The Weave.NET programming model addresses two issues: how to specify aspects, and what architecture is used to compose those aspects with components. We provide an introduction to both issues and then contrast the Weave.NET approach to aspect specification in AspectJ.

2.1 Specifying Aspects

Our aspect writing system is drawn from the experience of the AspectJ project, according to which "AO languages have three critical elements: a join point model, a means of identifying join points, and a means of affecting implementation at join points" [13]. For these three elements, it was convenient to draw on the semantics of AspectJ [34], since its model was well documented and, in our opinion, easy to grasp. Fortunately, this aspect model is sufficiently general that its join points can be identified in CLI types. The AspectJ pointcut language, including its primitive pointcut designators, is used to specify sets of join points. Manipulation at join points is conducted using its advice operators.

AspectJ syntax allows aspects to contain the same members as Java classes in addition to a set of exclusively AO constructs, such as pointcuts and advice; however, Weave.NET keeps AO constructs separate. In Weave.NET the cross-cutting details of an aspect are written in an XML deployment script. Non-AO type members, and indeed the behaviour of aspect advice, are obtained from an existing type implementation.

Weave.NET allows aspect behaviour and components to be implemented in any language that targets the CLI. Weave.NET places the declarative elements of an aspect in an XML file separate from source code. The declarative elements reference binaries that implement aspect behaviour, while the target components are specified when the Weave.NET API is called. Thus, aspect behaviour, as well as that of components, is compiled separately from the weaving process. The aspect programmer can then choose a suitable implementation language for aspect behaviour without affecting the ability to apply that behaviour in a crosscutting manner.

[1] "Compilers that are designed to both produce and extend [a library consisting of CLS-compliant code] are referred to as 'extenders'"[7]

2.2 Weaving Aspects

At the centre of the composition architecture is the Weave.NET tool as shown in Figure 1. The input to Weave.NET is an existing CLI binary component, packaged as a .NET *assembly*, and an XML file containing the crosscutting specifications of an aspect. The behaviour of an aspect is provided separately in another assembly. Weave.NET recreates the input assembly, but in this new version join points are bound to behaviour in the aspect assembly as per the advice statements in the XML. Unlike .NET approaches that bind components and aspects via proxies [20, 30], Weave.NET modifies the CIL of the components to access aspect behaviour via method calls. As a result, clients of components are unaffected by weaving and weaving on call join points is fully supported.

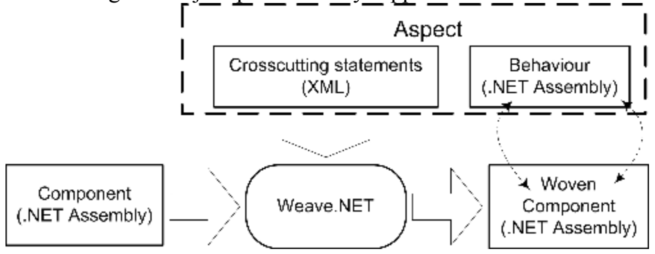

Aspect
Crosscutting statements (XML)
Behaviour (.NET Assembly)
Component (.NET Assembly)
Weave.NET
Woven Component (.NET Assembly)

Figure 1 User-level view of weaving

2.3 Contrasting Weave.NET and AspectJ

Figures 2 and 3 contrast the approach to implementing a logging aspect in AspectJ and Weave.NET respectively. We start by explaining the aspect's function using the AspectJ example, and then review the Weave.NET implementation looking for contrasts with the AspectJ approach.

Broadly speaking, the logging aspect is meant to report the data being written to I/O by a terminal emulator package called `tcdIO`. This I/O library was developed for introductory OO and C# instruction [4], and is referred to in examples throughout this paper. In the AspectJ implementation of Figure 2, the body of an advice statement implements the aspect's behaviour. Arrow 1 highlights how *before advice* references another member of the aspect type, `LogWrite`, to print data to the logging output. The `before` advice is applied to join points identified by the `Write` named pointcut, as indicated by arrow 2. `Write` specifies an intersection of execution join points specified with the `execution` and `args` primitive pointcut designators The `execution` designator identifies the output methods of a `Terminal` type, while the `args` designator selects from among these methods those that take a single argument[2]. `args` also exposes this parameter for manipulation by aspect advice. Among the join points selected is the execution of the `WriteLine` method as indicated by arrow 3. At compile time, AspectJ composes the aspect with component behaviour such that the execution of `WriteLine` initially transfers control to the before advice, as visualised by arrow 4.

The Weave.NET aspect of Figure 3 has all the elements of the aspect of Figure 2, but in a slightly different form. The crosscutting details of a Weave.NET aspect are specified in XML. An abridged version of the XML for the logging aspect is shown on the right of Figure 3. The behaviour of an advice statement is contained in a type referenced by the XML, as shown by arrow 1a. The behaviour of a specific advice statement corresponds to methods within that type as shown by arrow 1b. In the current implementation of Weave.NET, advice statements reference named pointcuts rather than using primitive pointcut designators themselves. As such, the before advice references a named pointcut as shown by arrow 2. Finally, the XML `primitive` tags articulate the same specification as the primitive pointcut designators of Figure 2 as shown by arrow 3. The transfer of control from component to aspect that results from weaving is visualised by arrow 4.

[2] The `object` type has certain wildcard characteristics that allow it to match parameters of any type.[34]

3. MAPPING THE ASPECT MODEL TO CLI

The aspect model in Weave.NET is derived from that of AspectJ. In this section we summarize this model's elements, and where possible, relate the elements to CLI architecture.

3.1 Join Point Model

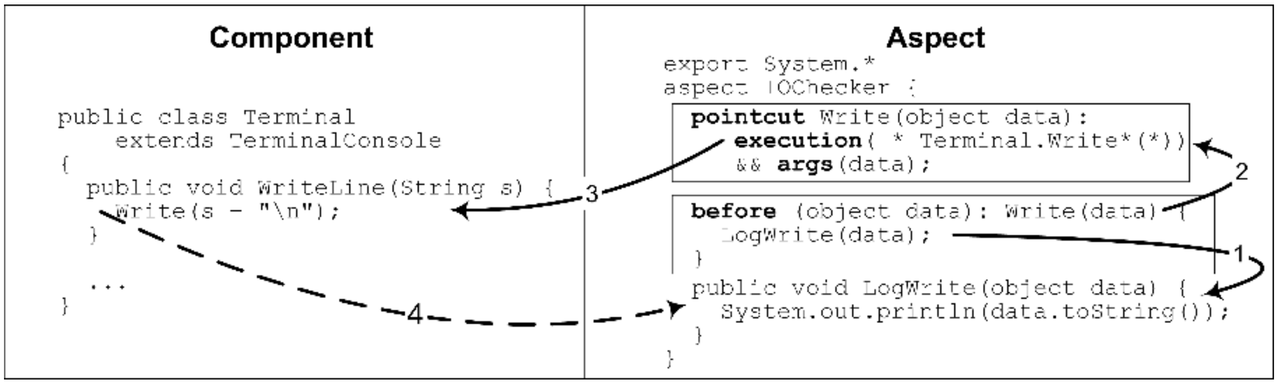
The Weave.NET aspect model contains only dynamic join points. Dynamic join points are "well-defined points in the execution flow of the program" [13]. In contrast, static join points correspond to types to which new members can be added. The focus on dynamic join points stems from their identification as core to the AspectJ aspect model [13].

Dynamic join points are best understood by organising them into three categories: execution join points, call join points and field access join points. This organisation is show in Table 1. AspectJ documentation [34] provides a better characterisation of specific join point types.

Join point category	Join point types
Execution	Method execution Initializer execution Constructor execution Static initializer execution Handler execution Object initialization
Call	Method call Constructor call Object pre-initialization
Field access	Field reference Field assignment

Table 1 Categorization of dynamic join points

Execution join points roughly correspond to the execution of a block of code, as opposed to a call or dispatch to that block. In the simplest case, the block may correspond to the body of a method. However, finer distinctions exist when it comes to the execution of exception handlers and the sequence of constructor executions and data member initializations during object creation.

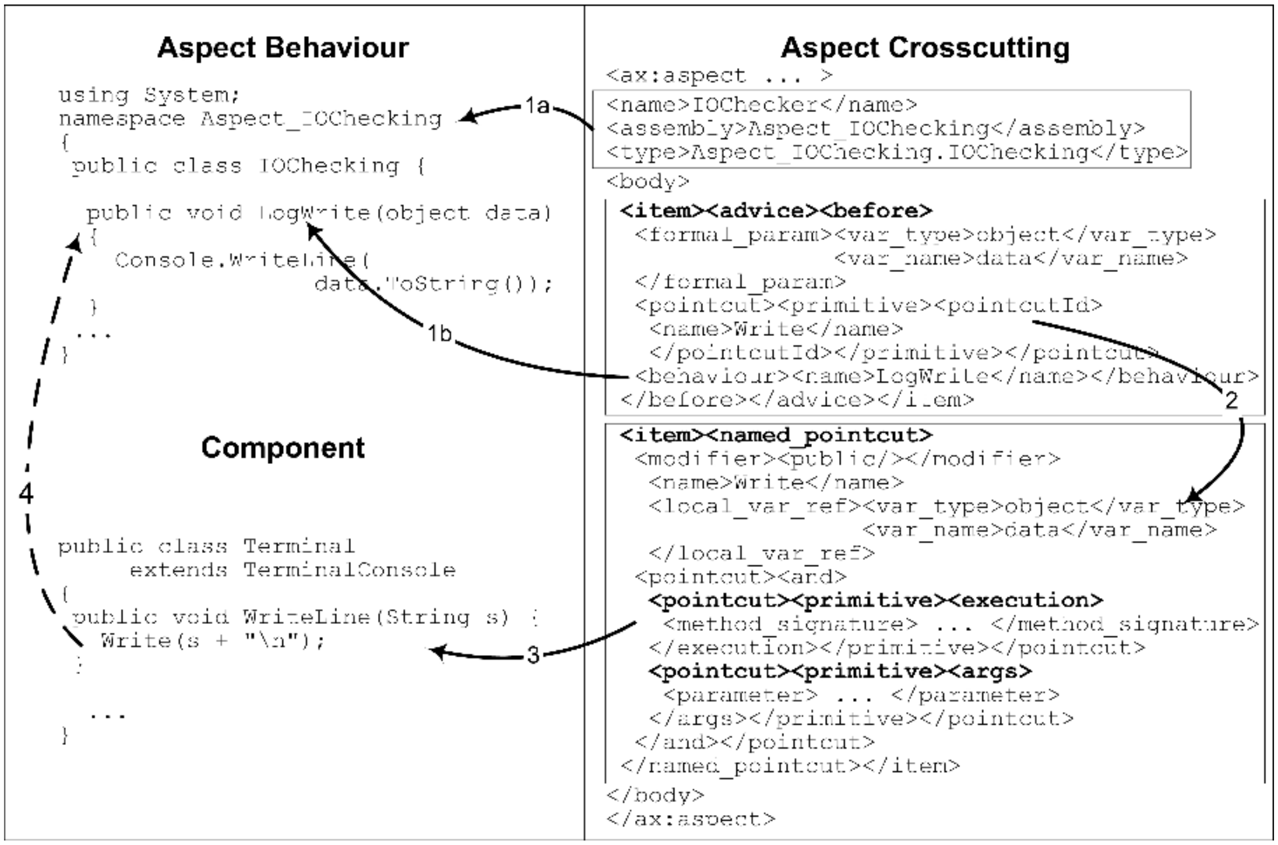


```
public class Terminal
    extends TerminalConsole
{
  public void WriteLine(String s) {
    Write(s + "\n");
  }

  ...
}
```

```
export System.*
aspect IOChecker {
  pointcut Write(object data):
     execution( * Terminal.Write*(*))
     && args(data);
  before (object data): Write(data) {
    LogWrite(data);
  }
  public void LogWrite(object data) {
    System.out.println(data.toString());
  }
}
```

Figure 2 Interpretation of an AspectJ aspect.



```
using System;
namespace Aspect_IOChecking
{
 public class IOChecking {

  public void LogWrite(object data)
  {
    Console.WriteLine(
                  data.ToString());
  }
  ...
}
```

```
public class Terminal
     extends TerminalConsole
{
 public void WriteLine(String s) {
   Write(s + "\n");
 }

  ...
}
```

```
<ax:aspect ... >
<name>IOChecker</name>
<assembly>Aspect_IOChecking</assembly>
<type>Aspect_IOChecking.IOChecking</type>
<body>
 <item><advice><before>
  <formal_param><var_type>object</var_type>
               <var_name>data</var_name>
  </formal_param>
  <pointcut><primitive><pointcutId>
   <name>Write</name>
   </pointcutId></primitive></pointcut>
  <behaviour><name>LogWrite</name></behaviour>
 </before></advice></item>
 <item><named_pointcut>
  <modifier><public/></modifier>
   <name>Write</name>
   <local_var_ref><var_type>object</var_type>
                 <var_name>data</var_name>
   </local_var_ref>
  <pointcut><and>
   <pointcut><primitive><execution>
    <method_signature> ... </method_signature>
   </execution></primitive></pointcut>
   <pointcut><primitive><args>
    <parameter> ... </parameter>
   </args></primitive></pointcut>
  </and></pointcut>
 </named_pointcut></item>
</body>
</ax:aspect>
```

Figure 3 Weave.NET equivalent of Figure 2

Weave.NET execution join points correspond to blocks of CIL. In a .NET assembly, IL code is located on a method by method basis. The assembly's metadata identifies which block of IL code corresponds to which method signature. This is true for constructors as well, since constructor bodies are modelled as methods with special names, such as `.ctor` in case of an instance constructor, and with certain metadata flags used to distinguish them from other methods.

Fine grained join points are resolved by closer inspection of the implementation of the method body. In the case of exception handlers, extra metadata tables associated with the method's code identify blocks of exception handling code. For execution join points related to object instantiation, it is necessary to examine the IL at the start of the constructor to distinguish constructor execution from object initialization. This is because data member initialization and flow of control between different constructors in a class' inheritance hierarchy is written explicitly into each constructor method.

```
public void WriteFloat(float value){
  WriteLine(value.ToString());
}
```

```
.method public hidebysig instance void
        WriteFloat(float32 A_1) cil managed
{
 // Code size       14 (0xe)
 // Execution start
 .maxstack  2
 IL_0000:  ldarg.0
 IL_0001:  ldarga.s   A_1
 IL_0003:  call instance string
          [mscorlib]System.Single::ToString()
 IL_0008:  call instance void
          [tcdIO]tcdIO.Terminal::WriteLine(string)
 // Execution end
 IL_000d:  ret
} // end of method Terminal::WriteFloat
```

Figure 4 An execution join point

To clarify the concept of execution join points, the example in Figure 4 shows C# source code and corresponding IL of an execution join point in the `tcdIO` library. The start and end of the execution join point are identified relative to the CIL with embedded comments in bold font.

Call join points are present on the calling side of a method invocation or when the `new` operator is called for object construction. These points are observed as IL opcodes of type `InlineMethod`. These opcodes indicate the target method with a metadata token. Using this token, it is possible to lookup the signature of the method being called. The signature also indicates where on the stack the call context is located. Constructors present a special case. They may be accessed as part of a call join point, for instance as part of a `new` operation, or they can be accessed as part of an execution join point, for instance via `this()` and `super()` calls in Java. Fortunately, these two cases are distinguished by the opcode used to access the constructor, which is `NewObj` in the case of a constructor call join point.

Revisiting the example in Figure 4, we can identify two call join points. In Figure 5, we highlight the call join point for the invocation of the `WriteLine` method in bold font.

```
public void WriteFloat(float value){
  WriteLine(value.ToString());
}


.method public hidebysig instance void
        WriteFloat(float32 A_1) cil managed
{
 // Code size       14 (0xe)
 // Execution start
 .maxstack  2
 IL_0000:  ldarg.0
 IL_0001:  ldarga.s   A_1
 IL_0003:  call instance string
          [mscorlib]System.Single::ToString()
 IL_0008:  call instance void
          [tcdIO]tcdIO.Terminal::WriteLine(string)
 // Execution end
 IL_000d:  ret
} // end of method Terminal::WriteFloat
```

Figure 5 A call join point

The final category of join point is that of field access, which corresponds to a read or write access to a data member, or field in CLI terminology. These join points do not include `final` fields, i.e. constant fields emitted as literals in IL. These join points are observed as special IL opcodes used to access static and non-static fields. These opcodes are associated with a metadata token identifying the signature of the field being accessed.

3.2 Identifying Join Points

To a large extent, the point of our aspect model is to allow succinct identification of join points and expose portions of their execution context. To do so, we adopt AspectJ's pointcut mechanism and its join point selection operators, called *primitive pointcut designators*, used to specify pointcuts. A pointcut selects from among all the join points in a component those that are relevant to a particular crosscut. To do so it relies on primitive pointcut designators that select from certain join point types, as defined by that designator, those whose metadata description matches the designator's argument. Thus, this argument is usually a signature or type pattern, depending on the designator. Finally, several designators can be used together with logical operators that take the union or intersection of their join point sets.

Designators can be broken into three categories according to the argument that they take. Table 2 identifies designators that identify join points in control flow directly from signatures or type patterns associated with the source of these join points. Table 3 identifies designators that identify join points relative to those of another pointcut. Finally, Table 4 identifies designators that select join points according to objects and arguments used in the execution context of the join point. These designators can also be used to expose the join point's execution context to the aspect.

Designator	**Joint points selected**
`call(`*`Signature`*`)`	Method and constructor calls.
`execution(`*`Signature`*`)`	Method and constructor execution.
`initialization (`*`Signature`*`)`	Object initializer execution.
`get(`*`Signature`*`)`	Field reference.
`set(`*`Signature`*`)`	Field assignment.
`handler(`*`TypePattern`*`)`	Exception handler execution.
`staticinitialization (`*`TypePattern`*`)`	Static initializer execution.
`within(`*`TypePattern`*`)`	All join points defined by the selected type.
`withincode (`*`Signature`*`)`	All join points defined within method or constructor matching declarations

Table 2 Designators specified with a signature or type pattern.

Designator	**Joint points selected**
`cflow(`*`pointcut`*`)`	All join points encountered during the execution of join points identified by the pointcut.
`cflowbelow( `*`pointcut`*`)`	Identical to *`cflow`*, but does not include the join points identified by the pointcut argument.

Table 3 Designators specified with a pointcut.

Designator	Joint points selected
`this( TypePattern or Id)`	Join points in which the object bound to `this` is an instance of a particular type.
`target( TypePattern or Id)`	Join points in which the object on which a call or field operation is applied to is an instance of a particular type.
`args( TypePattern or Id, ...)`	Join points where there are arguments whose types match those listed by the designator.

Table 4 Designators that can expose execution context.

In the case of signatures and type patterns, Weave.NET supports both *name-based crosscutting* and *property-based crosscutting* [13]. Name-based crosscutting corresponds to the literal expression of signatures and type patterns. Thus, with name-based crosscutting the signatures and type patterns used in a pointcut must match those of the targeted join points exactly. The CLI provides the `System.Reflection` API to access this data. Property-based crosscutting exploits wildcards to partially specify designator arguments. In property-based crosscutting, the signatures and type patterns used in a pointcut correspond to regular expressions. Fortunately, the CLI supplies a library to support regular expression use that greatly simplifies resolving these wildcards.

Pointcuts imply a traversal of all join points in the targeted source code. The CLI provides limited tools for directly accessing metadata, but none for accessing IL directly. Fortunately, there is a performance-conscious library called CLIFile Reader [5] that allows direct access to IL streams.

3.3 Modifying Join Point Behaviour

Weave.NET specifies aspect intersession in join points in terms of advice constructs described by AspectJ [14]. An advice statement specifies how to execute behaviour relative to, or rather than join points, in a pointcut. In principle there are three kinds of advice. *Before advice* executes just before the join point. *After advice* executes after the join point. Finally, *around advice* executes in place of the join point, but retains the capability to activate the join point.

There are three conditions for the execution of after advice depending on whether a join point completes normally or as part of an exception throw. These three categories are named accordingly as *after returning*, *after throwing*, and *after* advice depending on whether the join point returns normally, on account of a throw or due to either. Another difference between these is what variables in the execution context of a join point may be exposed. In after advice, the return type is not known. In contrast, after returning advice can expose the declared return type or an object reference to it. Likewise, after throwing advice can expose the thrown object.

Code generation in the .NET Framework is supported by the `System.Refleciton.Emit` API, but strictly speaking this API is not supposed to support the modification of existing .NET assemblies [25]. Weave.NET work on code generation indicates the current limitation is due to difficulties accessing method IL as a stream. Using the CLIFile Reader library, sufficient detail can be obtained to create an assembly at runtime, i.e. a dynamic assembly, based on an existing persistent assembly.

4. XML SPECIFICATION

The XML schema used in Weave.NET [19] was developed from a BNF grammar extracted from the Language Semantics Appendix of the AspectJ programming guide [34], and implemented in the W3C XML Schema Language [9]. The appendix was chosen as the specification for AspectJ for its easy to digest descriptions and supporting examples. However, the difficulty of specifying a language in a non-rigorous manner emerged when some unexpected contradictions in the language semantics[3] were discovered.

Concentrating on consistency with language syntax, rather than usability, resulted in an overly verbose XML schema. Recall how the aspect in Figure 2 was so much more compact than the abridged Weave.NET version in Figure 3.

As much as possible, the schema exploits the validation capabilities of W3C Schema. First, aspects are expressed mainly in terms of XML tags rather than XML tag attributes. The organisation of the tags and their contents are defined by complex types that can then validate the grammar of user aspects. Naturally, some tags such as identifiers and type patterns must contain data. These tags are described with simple types whose data is limited according to regular expressions. This removes the need to support a great deal of error checking in the weaver itself.

As observed in Figure 3, binding to aspect behaviour is done by name. The aspect's implementation type is selected by naming the implementation class and the containing assembly. The implementation type is selected on an aspect-wide basis to simplify aspect instantiation to a matter of instantiation of the implementing class. Binding advice to implementing behaviour is done by having the XML advice description select the method implementing the required behaviour. This method's parameter list must match the advice's typed formal parameters. *Typed formal parameters* correspond to the list of declarations for variables bound to execution context in a named pointcut and advice statement [34]. Admittedly, this is a simplification of the AspectJ model as it does not provide advice with access to metadata objects describing the join point execution context. However convenient to aspect writers, reflective access to join point context is not core to AOP, and thus not crucial at Weave.NET's current stage of development.

5. WEAVER IMPLEMENTATION

Weave.NET is an aspect weaver implemented as a .NET component. Its weaving interface accepts as input a reference to a component assembly and to an XML document that contains the specification for an aspect. The result of calling this interface is a new version of the component assembly that is bound to aspect behaviour at the IL level.

The weaver implementation has two subsystems: code generation and aspect modelling. The aspect modelling system is responsible for interpreting the XML aspect specification, modelling aspects in terms of their pointcuts and advice, and detecting whether join points match any aspect advice. The code generation system is responsible for converting an existing assembly to a dynamic assembly and instantiating objects to represent join points. The bridge between these two systems is the `JoinPoint` class

[3] In the context of a type pattern [34]: "There is a special type name, `*`, which is also a type pattern. `*` picks out all types, including primitive types. So `call(void foo(*))` picks out all call join points to void methods named `foo`, taking one argument of any type." But in the next paragraph "The `*` wildcard matches zero or more characters". In this case , `call(void foo(*))` picks out all call join points to void methods named `foo`, taking one **or zero** argument**s** of any type."

hierarchy. Instances of this hierarchy encapsulate join point details for examination by the aspect modelling system. They also provide code generation capabilities for embedding advice for use by the code generation system. In this section we will review the code generation system and examine how it interacts with `JoinPoint` objects. Next, we will review the aspect modelling system, and examine how it too interacts with `JoinPoint` objects.

5.1 Code Generation Architecture

The code generation system creates a dynamic assembly, i.e. a `System.Reflection.Emit` object hierarchy, corresponding to the assembly targeted for weaving. Were it not for the modifications specified by the aspect, this hierarchy would be emitted as a new, but functionally identical assembly. However, as per the aspect, there will be some differences. The principle classes used by the Emit library to model a dynamic assembly are shown in Figure 6. Here, a module corresponds to a physical file. Thus, an assembly can span files. Types and their constituent members are contained in one module or another.

The `System.Reflection` API has been suggested as a tool for introspecting on existing assemblies [30], but, as noted previously, this API lacks the ability to directly access the IL stream. Without access to IL it is impossible to expose call join points, so the code generation system bypasses the convenience of the Reflection library and examines the assembly metadata directly with the CLIFile Reader API [5]. The CLIFile Reader library provides abstractions to access intra-method details such as the IL stream and exception handling table. Directly accessing the file was considered, but CLIFile Reader provides decompression, metadata table modelling and greatly simplifies resolving cross-references within table entries.

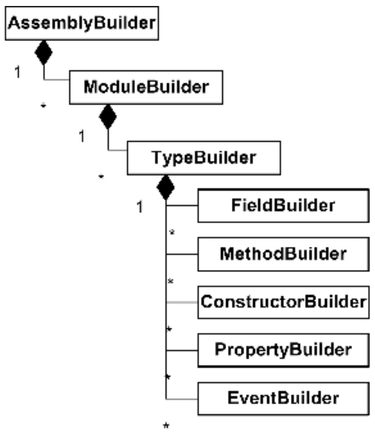


Figure 6 Dynamic assembly as modelled by Emit library.

The major drawback with using CLIFile Reader is that the metadata in a .NET assembly is organised on a module basis. That is, type members are keyed with module-wide identifiers that do not immediately identify their containing type. In contrast, the Emit library expects a type to directly reference its constituents. To bridge these two views, we introduce wrappers for each object class in the Emit library hierarchy to provide both views, as shown in Figure 7.

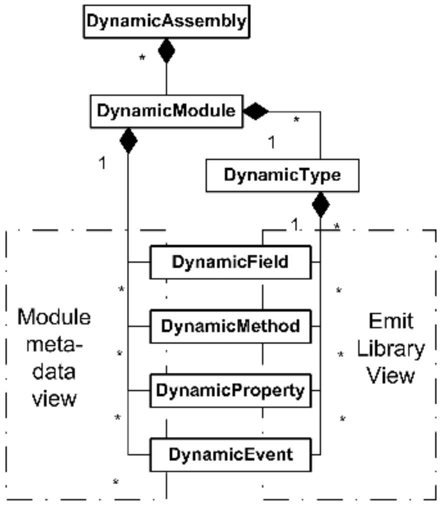


Figure 7 Resolving Emit object hierarchy and CLI metadata indexing.

In this system, conversion to a dynamic assembly requires a complete traversal of the CIL of every method. This traversal gives the code generation system an opportunity to expose supported join points. The join points are modelled by the class hierarchy defined in Figure 8, where `JoinPoint` and `JoinPointMethodSig` are abstract classes. Currently, Weave.NET only exposes call and execution join points.

As far as code generation is concerned, `JoinPoint` classes embed aspect advice by marshalling parameters and then calling the method that implements aspect advice. Embedding is requested by the code generation system before and after it emits the code corresponding to the join point. Separate classes are required to model each join point type as the opcodes required for marshalling parameters vary according to join point type.

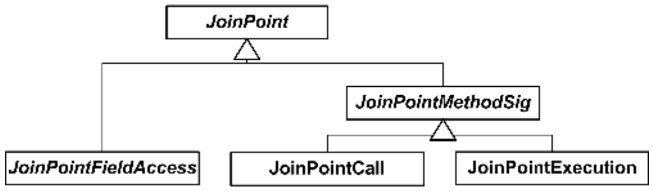


Figure 8 `JoinPoint` class hierarchy.

Aspect instances are associated with class objects through a field added during code generation. Proper instantiation of aspect instances requires advance knowledge of which component types are associated with which aspect instances. Our single-pass weaver cannot determine this information in advance, which leads to the addition of potentially unused fields corresponding to aspect instances. Thus, our work on aspect instantiation is incomplete.

5.2 Aspect Modelling Architecture

Aspect modelling is first activated in order to validate the aspect's XML and then convert it into the object hierarchy shown at the top of Figure 9. The .NET Framework provides the `System.Xml` library for modelling XML documents and `System.Xml.Schema` for modelling XML Schemas specifically. These APIs provide support for XML validation and W3C DOM [11] access for navigating the XML document. Specifically, the DOM API builds a navigable object graph corresponding to the

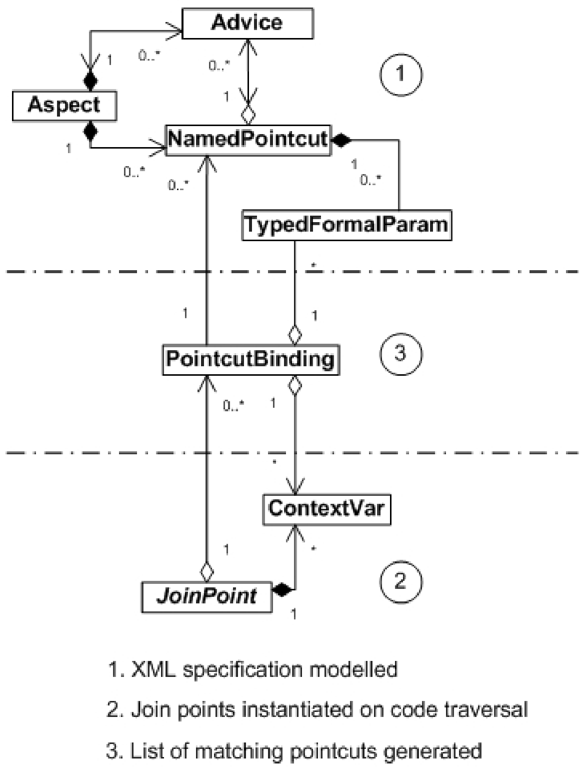
XML file. References to nodes in this graph, rather than copies of their data, are stored by the objects of classes in portion 1 of Figure 9.

An `Aspect` class object stores details that are static with respect to the advice and named pointcuts such as a reference to the type implementing advice behaviour. Specialisations of the `Advice` class exist for each kind of advice. Having the ability to modify existing assemblies allows all types of advice to be supported. However, the around advice is more reminiscent of method intersession in reflective programming and well supported in that domain by, for instance, the Iguana model [27]. As a consequence, in Weave.NET's current implementation, we have chosen to focus on before and after advice. With respect to the three kinds of after advice, code generation techniques allow any of these options. However, for an initial implementation, supporting returning after advice is sufficient as it can be expected that most methods complete normally, and that experimentation with after advice in general is most interesting when it is possible to act on the result of the join point. Regardless of whether they are named or not, all pointcuts are modelled as named pointcuts. `NamedPointcut` objects retain a reference to the XML element corresponding to the root of the pointcut description. The named pointcut also references its typed formal parameters. These correspond to the context variables that the pointcut exposes to advice.

The bottom portion of Figure 9 is instantiated by the code generation system whenever a join point is identified. The `JoinPoint` object contains the join point's signature and references to `ContextVar` objects describing the variables in the context of the join point. These variables are the parameters used to activate the join point as opposed to the set of all accessible variables within the scope of the join point.

The centre of Figure 9 arises when an `Aspect` object is asked to generate a `PointcutBinding` object for a particular join point. This involves the `Aspect` object traversing its list of named pointcuts asking each to determine if the join point is selected by its pointcut declaration. Matching with named pointcuts, rather than advice, reduces the maximum number of pointcut traversals to one, since multiple advice statements can reference the same named pointcut. The `PointcutBinding` object produced is used by the join point to determine which advice statements to apply and which context variables correspond to each typed formal parameter in the pointcut. Currently, the typed formal parameters in the advice method must match the type and order of those in the named pointcut.

For the purposes of prototyping, the most obvious designators to support would be the `execution` and `call` pointcut designators, as they are simple to conceptualize and are used in API programming. In terms of complexity, `execution` and `call` map directly to a particular type of join point, whereas other pointcut designators, such as `within`, select groups of join points that span the three categories of join points. With respect to APIs, the ability to capture execution join points in an API's implementation and call join points during its use allows interesting tests to be formulated to demonstrate Weave.NET, as we will see in section 6. As far as implementation is concerned, `execution` and `call` designators can be identified by signature alone. In contrast, `cflow` pointcuts must track the stack at runtime to determine when they have returned to their starting point.



Advice
Aspect
NamedPointcut
TypedFormalParam
PointcutBinding
ContextVar
JoinPoint
1
2
3
0..*
1
*
1. XML specification modelled
2. Join points instantiated on code traversal
3. List of matching pointcuts generated

Figure 9 Aspect modelling and join point matching architecture.

As far as context exposure is concerned, our prototype focuses on the `args` designator, because it provides quite a bit of detail on the context in which a join point is called. For completeness, our system also allows a typed formal parameter to be bound to the value returned by after-returning advice.

6. INITIAL ASSESSMENT

Our initial assessment examines two areas of interest to potential users of Weave.NET. Since the focus of our tool is language independence, our first assessment examines the application of aspect behaviour, written in different languages, to components, again written in different languages. Next, our performance analysis examines the practicality of Weave.NET aspects in terms of their ability to implement crosscutting concerns and the load-time and runtime overheads they introduce into an application.

6.1 Cross Language Weaving

To demonstrate cross language weaving, we formulated two crosscutting scenarios and implemented them with aspect behaviour written in two languages, for a total of four instances of weaving. These test cases are captured in Figure 10.

In the first scenario, an `execution` primitive pointcut designator is used to perform *service-side engineering* in the `tcdIO` library. The term service-side engineering captures the modification of the capabilities of an API. In this scenario, `tcdIO` is modified to log the use of its output methods, which, in a broader context, can be useful during teaching laboratories for diagnosis of student problems as well as grading during student demonstrations.

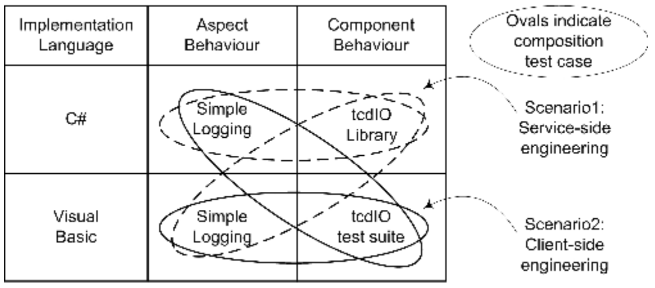


Figure 10 Language-independence test cases

In this test, the modified `tcdIO` library is created at runtime by calls to the Weave.NET API and this modified library is used by a test suite. Normally, `tcdIO` would be tested by a test suite that exercises each of its I/O calls. That test suite has its own entry point in the form of a public static method called `Main`. In our test, we modify the entry point to the test suite to be a different version of `Main` that first calls the weaver to apply the logging aspect to the `tcdIO` library and then passes control to the `Main` of the original test suite. The parameters to the weaving API are the target assembly, its location, the XML file detailing the aspect to be applied, and the location of this XML file. When control is passed to the test suite, the presence of logging calls indicates that the weaving was successful.

The test is formulated twice. In the first instance, the aspect behaviour is written in C#. This test is analogous to application of the aspect described in Figure 3. In the second instance, the aspect behaviour is written in the Visual Basic of Figure 11.

```
Public Class IOChecking
  Public Shared Sub LogWrites(ByVal data As System.Object)
    Console.WriteLine(data.ToString())
  End Sub

  Public Shared Sub LogReads(ByVal data As System.Object)
    Console.WriteLine(data.ToString())
  End Sub
End Class
```

Figure 11 Simple logging implemented with Visual Basic

In our second crosscutting specification, we perform *client-side engineering* on a test suite for `tcdIO`. The term client-side engineering captures the modification of the way in which assemblies call an API. Specifically, `call` primitive pointcut designators are used to apply logging advice to calls to `tcdIO` during testing. Such logging functionality is suited to the automation of assignment grading, where we wish to standardize input to student assignments or verify the results in an automated fashion.

The test procedure is the same used when we attempted service-side engineering, albeit with the test suite targeted for weaving rather than the `tcdIO` library. Again, this test is performed in two instances where the language implementing the logging functionality is varied between C# and Visual Basic. These tests are captured in the composition ovals in the lower half of Figure 10. The figure accurately describes the aspect functionality in the second scenario as being reused from the first scenario.

The exact procedure of the test involved calling the weaving API with the location of the XML document and the component assembly. While not cited, the aspect behaviour had to be available at weave time in order to properly generate the woven component.

Based on this testing, two recommendations were made. A metadata object describing the join point context would make the logging capabilities more interesting by allowing them to report which methods they were bound to. Also, the weaver needs to implement the full XML schema. At present, the weaver lacks the ability to distinguish join points by accessibility. As a consequence, logging in the first scenario reports the execution of private methods with signatures matching the pointcut specification, but whose execution is of no real interest. This issue will be addressed as we continue development on Weave.NET.

6.2 Performance

Our examination of performance focuses on establishing the practicality of Weave.NET. This assessment starts by asking what problems Weave.NET aspects can in theory solve. Having investigated its potential for improving programming, we look at whether applications that make use of Weave.NET will face unreasonable execution time overheads.

AOP in general has been justified through its ability to improve performance and at the same time address program complexity by reducing the number of lines of code [15], and these goals are met by Weave.NET. Weave.NET addresses tangling in a similar manner to AspectJ and can thus be expected to have a similar effect on the number of lines of code in an application. Indeed, this is confirmed when we compare the number of lines of code required to implement the components of Figure 10 with logging to the number of lines of code that Weave.NET requires to implement the same functionality. This comparison is shown in Table 5, which provides the source code line count for implementing component and aspect behaviour separately and for a new implementation in which the two are combined or tangled. The separately compiled versions correspond to the source code base when Weave.NET is used, while the tangled versions correspond to when it is not used. In Table 5, the aspect behaviour contributes 8 lines of source to the untangled implementations. This line count is on the code size of the Visual Basic implementation. Unfortunately, our comparison is somewhat naïve as the Weave.NET source code size does not take into account the XML specification of the logging aspect.

Component targeted for logging	**Lines of source in untangled version**	**Lines of source in tangled version**
`tcdIO` test suite	90	141
`tcdIO` library	361	394

Table 5 Comparison of lines of source code in test components before and after logging calls added manually.

As far as improving application performance by making aspects available, the tool being used is less of an issue as compared to the AOP constructs available and their usefulness. Currently Weave.NET supports a subset of the dynamic crosscutting constructs of AspectJ. Due to type checking, the current design fails to support introduction statements required to support static crosscutting. The full set of dynamic crosscutting constructs specified, as well as their Weave.NET support, is documented in Tables 6, 7 and 8. Although the current implementation is limited, the underlying design and schema used by Weave.NET should be sufficient to fully support the full set of dynamic crosscutting operators. Thus, the ability of Weave.NET aspects to improve application performance is comparable to the ability of AspectJ to do the same.

Pointcut designators	Supported?
Call	Yes
Execution	Yes
Get	No
Set	No
Handler	No
Initialization	No
Staticinitialization	No
Within	Yes
Withincode	No
Cflow	No
Cflowbelow	No
This	Yes
Args	Yes
Target	No

Table 6 Primitive pointcut designator support currently implemented in Weave.NET.

Advice	Supported?
Before	Yes
Around	No
After	Yes
After returning	Yes
After throw	No

Table 7 Advice support currently implemented in Weave.NET.

Aspect Instantiation	Supported?
Singleton	Yes
Perthis	No
Pertarget	No
Percflow	No
Percflowbelow	No

Table 8 Aspect instantiation support currently implemented in Weave.NET.

Where Weave.NET distinguishes itself from AspectJ is in the use of load-time weaving. By weaving at load-time, Weave.NET introduces application overhead not present when AspectJ applications execute. To put this overhead into perspective, we have devised a simple test to relate the time required to weave with Weave.NET to that required by AspectJ to weave pre-runtime. Although AspectJ is known as a source-level weaver, the most recent release makes available byte-code weaving so that byte-code in existing `.jar` files can be targeted. While it still appears that source is required for the aspect, the byte-code level component weaving is not unlike the task carried out by Weave.NET. So, we have taken a Java version of `tcdIO`, very similar in design to the .NET version, and measured the amount of time required for AspectJ to weave logging into the Java based `tcdIO` binary. Logging was also added to a Java port of the `tcdIO` test suite.

A side by side comparison of weaving times required by Weave.NET and AspectJ is shown in Table 9 and Weave.NET compares favourably. The times were recorded during trials on a 1.7 GHz Pentium 4 based PC with 512 MB of RAM running WindowsXP. Weave.NET ran in the .NET Framework, version 1.1.4322, while AspectJ ran in J2RE version 1.4.2. Weaving was launched from a command shell, and a mean average taken of the three best trials among five tests. In examining the results, it should be pointed out that the AspectJ aspect differed from that used by Weave.NET in the number of pointcuts required to specify logging. In the CLI, an object reference can be obtained for any type, including what Java terms *primitive* types. Thus, the primitive pointcut `args(Object)` matches all methods with a single parameter in Weave.NET, but in AspectJ it does not include methods whose single parameter is a primitive type.

The execution times in Table 9 indicate that Weave.NET weave time is not unreasonable with respect to that of AspectJ. However, it should be pointed out that the test was performed on a very small component. Moreover, the version of Weave.NET tested may benefit from supporting a smaller set of join point types and primitive pointcut designators than AspectJ. Finally, AspectJ's weaving appears to scale better than that of Weave.NET as its weave time increases more slowly as the size of the component being woven increases.

Weaving task	Weaving time for Weave.NET	Weaving time for AspectJ
`tcdIO` weaving	502.5	3568
test suite weaving	365.8	2967

Table 9 Load-time overhead Weave.NET compared to similar weaving in AspectJ (time in milliseconds).

Finally, we would look for Weave.NET to not introduce any run-time overheard. Beyond aspect instantiation, the current implementation does not introduce any runtime structures. This will change as support is added to allow reflection on join point context and to allow control flow pointcuts that must dynamically track execution context. So, after accounting for the load-time weaving, Weave.NET overhead should be zero. This is worth verifying, as performance has been a major problem for other technologies that in some sense intercede in normal program execution. For instance, the runtime reflective programming tool IguanaJ initially reported that object instantiation time increased by 25 times when the object's class was targeted for intercession. [26]. Table 10 provides a comparison of the execution of `tcdIO` and its test suite with logging added by Weave.NET in the first case and added with manual embedded method calls in the second case. The test platform and procedure for collecting results here is the same as for the previous test. The results indicate that Weave.NET introduces no appreciable increase in runtime overhead in this case.

Weave.NET woven logging	Manually introduced logging
380.145	381.116

Table 10 Comparison execution time of Weave.NET and manually written logging (time in milliseconds).

7. RELATED WORK

The AOSD community website (http://www.aosd.net) provides links to several other AOSD tools and languages, both supported and experimental, as well as a host of AOSD methodologies. Among these are the long standing AOP technologies, MDSOC [24], Demeter [22], Composition Filters model [2] and AspectJ [14], examined when section 1 established an absence of language-independence in AOP technology.

The use of XML in Weave.NET contrasts with previous approaches that have used CLI custom attributes in expressing crosscutting. The work on WrapperAssistent [30] has succeeded in providing an aspect-specific language for expressing fault tolerance, which is implemented with an aspect specific weaver. Join points are identified by embedding declarative statements in the component code, which is a matter of great debate within the AO community. Depending on the importance placed on making join points oblivious to aspects [10], such annotations would be less preferable to an approach that required no modification of the component code. In a more recent version of WrapperAssistent, named Loom.NET [29], join point selection has become a GUI activity and support has been added for the *aspect specific templates* mechanism first described in [30]. Aspect specific templates provide a way to write proxies using macros that map to declarative elements in the component being proxied. The code surrounding these macros is C#. So, while a component written in any language can be targeted, the aspects must be written in C#. As part of Loom.NET work a dynamic library has emerged [31] that provides a mechanism for weaving aspects at runtime time on an object by object basis. The aspect model is reminiscent of composition filters in that method calls to the woven object are delegated to an aspect instance. By using only CLI elements, the dynamic system should theoretically allow aspects written in any language to be applied to components written in any language, but cross language support has not been verified.

The use of XML to specify aspects is not without precedence. CLAW [20] and AOP# [32] each present a strategy for language-independence that relies on XML to specify the crosscutting elements of an aspect. CLAW explores weaving using a profiling interface specific to the .NET Framework in order to build dynamic proxies for aspect weaving. Although CLAW promises to be language-independent, it lacks an aspect model to dictate how join points are selected or manipulated. For instance, there is no XML schema defined for describing aspects. AOP# presents the concept of *aspectual polymorphism* in which the implementation used for an aspect bound to an object will vary according to the context in which the object is used. Another interesting property is the ability for aspects to specify requirements in the form of methods that target components must support. These requirements are mapped to elements of the component in an XML script. Most recently, the project has changed name to AOP.NET [28], and produced a composition library that exploits the .NET Framework's profiling interface; however an implementation of the aspect model has yet to emerge.

Work on AspectC# [16] predates that of Weave.NET. AspectC# provides an aspect model for C# in which weaving is specified in XML to avoid extending language syntax. The weaving mechanism works by manipulating an abstract syntax tree (AST) representation generated from component source. However, this AST cannot currently model all languages that target the CLI. Continuing work on AspectC# centres on plans to adopt the Weave.NET XML schema for specifying aspect composition.

The code modification mechanism of Weave.NET is explored in the domain of byte-code instrumentation, which has been exploited by JMangler [17] [18] to provide an AOP composition mechanism that allows aspect users to avoid conflicts during composition of independently developed aspects. JMangler allows transformations to be written in XML for Java by supplying a new implementation of the class loader.

Load-time instrumentation was explored at the same time in Binary Component Adaptation (BCA) [12] and JOIE [6]. BCA allows adaptation of component interfaces in Java to simplify component integration in light of evolving component interfaces. With respect to AOP, BCA is a language specific solution to the interface evolution crosscutting concern. JOIE characterises the concept of load-time transformation in the context of a transformation library that is analogous to the Emit API of the CLI summarized in Figure 6.

8. CONCLUSION AND FUTURE WORK

The purpose of Weave.NET is to make AOP language independent in so far as the behaviour of aspects and the components to which they are applied can be written in any language. In contrast to previous efforts, Weave.NET allows aspect writers to choose the language in which they implement aspect behaviour. More importantly, Weave.NET allows existing code to be targeted by aspects, regardless of implementation language, broadening the set of components to which AOP can be applied. As a result, programmers can avoid discarding their existing skills in order to adopt the AOP paradigm.

This paper describes the operation of Weave.NET from a programmer's point of view, and provides details on the underlying aspect model. The aspect model is drawn from AspectJ, while language interoperability is based on the Common Language Infrastructure (CLI) designed for the .NET Framework. The aspect programmer is responsible for implementing aspect behaviour in the language of their choice and generating the corresponding binary component. The crosscutting statements of the aspect are written with an XML script based on the syntax of AspectJ, and they apply behaviour from the aspect's binary component. The schema for this script is rigorously specified in W3C XML Schema language. The weaver is implemented with two subsystems, one responsible for code generation and the other for aspect modelling. Interchange between the two systems is achieved using objects that model join points in terms of the details required to match join points to crosscutting statements in an aspect and the code generation capabilities required to compose join points with aspect behaviour. Language-independence was verified in service-side and client-side engineering scenarios. Specifically, logging, written in C# and Visual Basic code, was added to the execution of methods in an I/O package written in C# and to calls to this API by a test suite written in Visual Basic. Weave.NET's CLI focus is shared by other technologies, but these do not match its language-independence capabilities. Neither do the implementations of other popular aspect models.

Future work in Weave.NET will involve broadening its crosscutting capabilities and reflection support to allow for more interesting aspect behaviour. While the aspect XML schema is complete, the full set of primitive pointcut designators and advice statements are not supported, which limits the effectiveness of our aspects. For example, our initial assessment noted proper logging requires signature specification be broadened to include accessibility modifiers. Also, testing indicates the need to make available a metadata object to provide aspects with reflective access to the join point's execution context.

9. ACKNOWLEDGMENTS

We would like to acknowledge Microsoft Research for their sponsorship of the Weave.NET project, as well as Jim Dowling and especially Elisa Baniassad for their helpful comments on the final text.

10. REFERENCES

1. Bergmans, L. *The Composition-Filters Object Model*. PhD Thesis, Department of Computer Science, University of Twente, 1994.
2. Bergmans, L. and Aksit, M. "Composing Crosscutting Concerns Using Composition Filters". *Communications of the ACM*, *44* (10), October 2001, pp.51-57.
3. Booch, G. *Object-oriented Analysis and Design with Applications*. Benjamin/Cummings, Redwood City, California, 1994.
4. Cahill, V. and Lafferty, D. *Learning to Program the Object-Oriented Way with C#*. Springer-Verlag UK, London, 2002.
5. Cisternino, A. *CLIFileReader Library*, C# Source Code, http://dotnet.di.unipi.it/MultipleContentView.aspx?code=103, 2002.
6. Cohen, G., Chase, J. and Kaminsky, D., Automatic Program Transformation with JOIE. In *USENIX Annual Technical Conference '98*, (1998).
7. ECMA International. *Standard ECMA-335 Common Language Infrastructure (CLI)*, ECMA Standard, http://www.ecma-international.org/publications/standards/ecma-335.htm, 2003.
8. Elrad, T., Filman, R.E. and Bader, A. "Aspect-oriented Programming". *Communications of the ACM*, *44* (10), October 2001, pp.29-32.
9. Fallside, D.C. *XML Schema Part 0: Primer*, W3C Recommendation, http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/, 2001.
10. Filman, R.E. and Friedman, D.P., Aspect-Oriented Programming is Quantification and Obliviousness. In *OOPSLA 2000 Workshop on Advanced Separation of Concerns*, (Minneapolis, USA, 2000).
11. Hors, A.L., Hégaret, P.L., Wood, L., Nicol, G., Robie, J., Champion, M. and Byrne, S. *Document Object Model (DOM) Level 2 Core Specification*, Website, http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113, 2000.
12. Keller, R. and Hölzle, U., Binary Component Adaptation. In *ECOOP*, (Brussels, Belgium, 1998), Springer-Verlag, pp.309-329.
13. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.G. "Getting Started with AspectJ". *Communications of the ACM*, *44* (10), October 2001, pp.59-65.
14. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.G., An Overview of AspectJ. In *ECOOP 2001*, (Budapest, Hungary, 2001), Springer-Verlag, pp.327-355.
15. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.-M. and Irwin, J., Aspect-Oriented Programming. In *The 1997 European Conference on Object-Oriented Programming (ECOOP'97)*, (Jyväskylä, Finland, 1997), Springer-Verlag, pp.220-242.
16. Kim, H. *AspectC#: An AOSD implementation for C#*. MSc Thesis, Comp.Sci, Trinity College, Dublin, Dublin, 2002.
17. Kniesel, G., Costanza, P. and Austermann, M., Independent Extensibility for Aspect-Oriented Systems. In *ASC Workshop, ECOOP 2001*, (Budapest, Hungary, 2001).
18. Kniesel, G., Costanza, P. and Austermann, M., JMangler - A Framework for Load-Time Transformation of Java Class Files. In *IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2001)*, (Florence, Italy, 2001).
19. Lafferty, D. *W3C XML Schema for AspectJ Aspects*, XML Schema, http://aosd.dsg.cs.tcd.ie/XMLSchema/aspect_Schema.xsd, 2002.
20. Lam, J. Cross Language Aspect Weaving, Demonstration, AOSD 2002, Enschede, 2002.
21. Lidin, S. *Inside Microsoft .NET IL Assembler*. Microsoft Press, Redmond, Washington, 2002.
22. Lieberherr, K., Orleans, D. and Ovlinger, J. "Aspect-Oriented Programming with Adaptive Methods". *Communications of the ACM*, *44* (10), October 2001, pp.39-41.
23. Lieberherr, K.J. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996.
24. Ossher, H. and Tarr, P. "Using Multidimensional Separation of Concerns to (Re)Shape Evolving Software". *Communications of the ACM*, *44* (10) 2001, pp.43-50.
25. Rajan, J. *Re: Reflection.Emit question*, Newsgroup posting, http://discuss.develop.com/archives/wa.exe?A2=ind0012B&L=DOTNET&P=R2535, 2002.
26. Redmond, B. and Cahill, V., Supporting Unanticipated Dynamic Adaptation of Application Behaviour. In *ECOOP 2002*, (Malaga, Spain, 2002), Springer-Verlag.
27. Schaefer, T. *Supporting Meta-types in a Compiled, Reflective Programming Language*. PhD Thesis, Department of Computer Science, University of Dublin, Dublin, 2001.
28. Schmied, F. *AOP.NET*, http://students.fhs-hagenberg.ac.at/se/se99047/english/aop_net.html, 2003.
29. Schult, W. *LOOM.NET*, http://www.dcl.hpi.uni-potsdam.de/cms/research/loom/, 2003.
30. Schult, W. and Polze, A., Aspect-Oriented Programming with C# and .NET. In *5th IEEE International Symposium on Object-oriented Real-time Distributed Computing*, (Washington, DC, 2002), IEEE Computer Society Press, pp.241-248.
31. Schult, W. and Polze, A., Speed vs. Memory Usage - An Approach to Deal with Contrary Aspects. In *2nd AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)* in *AOSD 2003*, (Boston, Massachusetts, 2003).
32. Schüpany, M., Schwanninger, C. and Wuchner, E., Aspect-Oriented Programming for .NET. In *First AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, (Enschede, The Netherlands, 2002), pp.59-64.
33. Tarr, P., Ossher, H., Harrison, W. and Stanley M. Sutton, J., N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proceedings of the 21st International Conference on Software Engineering*, (Los Angeles, USA, 1999), IEEE Computer Society Press, pp.107-119.
34. The AspectJ Team. *The AspectJ Programming Guide (V1.0.6)*, http://download.eclipse.org/technology/ajdt/aspectj-docs-1.0.6.tgz, 2002.