SourceWeave.NET: Cross-Language Aspect-Oriented Programming

Andrew Jackson, Siobhán Clarke

Distributed Systems Group, Department of Computer Science, Trinity College Dublin, Dublin 2, Ireland
{Andrew.Jackson, Siobhan.Clarke}@cs.tcd.ie

Abstract. Aspect-Oriented Programming (AOP) addresses limitations in the Object-Oriented (OO) paradigm relating to modularisation of crosscutting behaviour. In AOP, crosscutting behaviour is expressed as *aspects* that are integrated with a base program through a *weaving* process. Many language-specific AOP models already exist, requiring the programmer to work with a single language for base and aspect programs. The .NET framework, with its multi-language standards and architecture, has presented a new opportunity for *cross-language* AOP within .NET. Advances have been made to take advantage of this opportunity, but at the expense of qualities such as the ability to debug executing code, or define some kinds of weaving capabilities. This paper describes an investigation into providing cross-language AOP in .NET without making such compromises. The approach, called SourceWeave.NET, allows debugging because it weaves source code, and also provides an extensive weaving model. We describe what can be learned from SourceWeave.NET, in terms of both its benefits, and also its limitations.

1 Introduction

When modelling software systems using the Object-Oriented (OO) paradigm, some kinds of behaviours cannot be cleanly modularised into the primary OO encapsulation unit, the object. Behaviour that has an impact on multiple objects in a system (often termed *crosscutting*) is likely to be scattered and tangled throughout the system. Aspect-Oriented (AO) programming addresses this problem by providing constructs to modularise crosscutting behaviour, and to state where in the system this behaviour should be executed [9].

There have been many different approaches to implementing AOP that target a single programming language [3, 4, 7, 8, 17, 6, 36]. The .NET framework opens up a whole new set of opportunities to provide the benefits of AOP in a multi-language environment [22]. .NET is an architecture that enables multi-language programming. It is based on a set of standards that specify a common infrastructure for language implementation, and provides a run-time environment on which supported languages can execute. The architecture is extensive, and offers opportunities for different strategies for employing language independent AOP. Possible strategies include weaving at pre-compile-time, compile-time, load-time or run-time. There has already been a considerable amount of activity related to applying AOP to .NET in different ways. In general for such approaches, programmers lose the ability to debug their source code. Current approaches that work with source code in .NET, such as EOS [25] and Aspect.NET [28] do so by extending a particular language, and lose the benefit of cross-language AOP.

This paper describes SourceWeave.NET, which provides cross-language AOP at the source code level while providing a high level of support for *debugging expressiveness*. Debugging expressiveness relates to the extent to which a developer may step through and inspect woven source code. With a high level of debugging expressiveness, a developer may scrutinise state changes as woven crosscutting behaviour executes. This level of developer support is becoming more important as a number of technological advances further complicate modern application development. For example, advances in wireless network technologies, together with requirements to support user mobility and pervasive computing, pose new challenges to developers that are required to provide mobile, context-aware applications. It is likely that application components in such an environment will be expressed in many different languages (e.g., domain specific languages (DSLs)) and will be required to collaborate and interact to provide the context-aware services a mobile user requires. The highly dynamic nature of pervasive environments also requires applications to deal with unpredictable state changes, compounding the need for complexity management and a high level of debugging expressiveness. AOP can help reduce the inherent complexity of this pervasive environment, and SourceWeave.NET provides an AOP environment that can deal with the multiple languages of collaborating components and also provide debugging expressiveness for dynamic state change scrutiny.

The SourceWeave.Net architecture is based on a .NET standard for representing source code as abstract syntax trees called CodeDOM. Using SourceWeave.NET, a developer can write base and aspect components in standard C#, VB.NET and J#, specify how these components should be woven with an XML descriptor, and step through the original source when debugging woven systems.

Section 2 provides some background into the technologies either used in, or informing, SourceWeave.NET. Section 3 describes the SourceWeave.NET architecture, and how it works. Section 4 provides an assessment of the benefits and liabilities of the approach. Section 5 describes related work, while Section 6 summarises and concludes.

2 Background

This section provides a brief overview of AspectJ and .NET. The AOP model used by AspectJ was considered as a requirements specification for SourceWeave.NET's weaving model. We describe .NET especially in terms of the standards it imposes and uses. The .NET architecture gives considerable scope for different strategies for applying the AO paradigm to the architecture. A brief discussion of some possibilities is included.

2.1 AspectJ

AspectJ extends Java and enables the developer to modularise additional behaviour to run at certain well-defined points of execution in the program, called “crosscutting” behaviour [35]. Handling crosscutting in AspectJ is based on identifying points of execution in the program (*joinpoints*) and specifying additional behaviour to be executed at those execution points (*advice*). This specification of a point of execution in a program is termed a *pointcut* and it identifies joinpoints. A pointcut can be primitive or composite. A primitive pointcut identifies a particular code section based on a singular characteristic of the code. For example, the primitive handler(FooException) pointcut identifies joinpoints at catch constructs where the type or super-type of the exception caught matches the exception-type specified in the handler pointcut. A composite pointcut is a series of pointcuts composed via logical operators. A joinpoint can be mapped to a specific construct in the program code which when executed results in the generation of the joinpoint. The particular construct exists in a code segment called the shadow of the joinpoint (*joinpoint shadow*) [14]. Through comparing the characteristics of pointcuts and the program text, joinpoint shadows are identified. The handler(FooException) pointcut matches catch constructs that refer to the FooException type and its super-types. The executable code segments of catch constructs matched are then joinpoint shadows. At run-time joinpoints are generated based on the positive evaluation of the pointcuts against the joinpoint shadows. Advice is a method-like mechanism used to identify behaviour that should be executed at joinpoints. AspectJ provides a sophisticated model for joinpoints, pointcuts and advice that we considered a good basis from which to conduct our research.

2.2 .NET

The .NET framework is an architecture that enables multi-language programming and is based on a series of ECMA [2] and Microsoft standards. These standards specify a common infrastructure on which different programming languages can be implemented. The layered architecture of the .NET framework is illustrated in Figure 1. There are two high-level layers within the architecture. The first is the compilation layer where source code written in various languages can be compiled to assemblies. The second layer is the run-time environment, which takes the assemblies and executes the Intermediate Language (IL) code packaged within the assemblies.

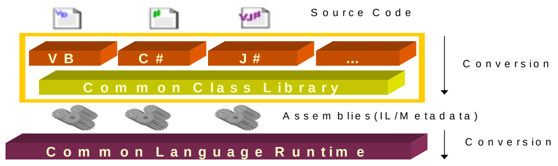


Figure 1: .NET Layered Architecture

The compilation layer is defined by the Common Language Specification [23] (CLS) standard. This defines conventions that languages must support in order to be inter-operable within .NET. Any .NET compliant language that conforms to the CLS can compile to a common IL. Each language implementer provides a compiler that takes in source code and outputs IL packaged as assemblies or modules. A Common Class Library [20] (CCL) is a set of base namespaces or libraries that are common to all .NET languages and are available in all .NET compliant languages. A Common Type System [21] (CTS) is a subset of the CLS that defines OO types and values that must be supported by all .NET compliant languages. A CTS is the most critical prerequisite for language integration.

Assemblies are deployable units that contain IL code executed by the Common Language Runtime (CLR) [15]. The CLR acts as a virtual machine that manages code execution, memory and threads. An assembly forms a security, type, reference scope and version boundary in .NET [18]. It follows a strict format and contains an assembly manifest. This manifest exposes metadata about the assembly, allowing it to be self-describing.

2.3 Applying AOP to .NET

One of the main attractions of the .NET framework is its support for language independence. Approaches to applying AOP to .NET have an opportunity to take corresponding advantage of this to achieve cross-language aspect-oriented programming. There are a number of possible strategies that can be followed to achieve this. These are discussed briefly here and are later expanded in Section 5.

One strategy is to extend the CLR. This strategy has been shown to exhibit several drawbacks including performance degradation; current approaches that have followed this strategy support limited joinpoint models and do not support essential developer tools such as debuggers.

Another strategy is to apply AOP to .NET through weaving assembly packaged IL. IL based approaches have had varying levels of success. This strategy however, suffers a loss of tool support such as debugging. This is due to breaking the bonds that bind the IL to source code through IL weaving. This adversely affects the debugger's ability to step through woven source code.

An extension of the CLS/CTS to introduce AOP at the compilation level is another possible strategy to achieve cross-language AOP. However, changing these specifications means that the .NET language providers have to update their language implementation accordingly. Beyond this, the framework tool-set would also have to be upgraded to handle the new AO constructs. Applying the new AOP extension to the CLS/CTS would be infeasible in the short-to-medium term.

A strategy based on working at the source-code level avoids these problems encountered by other approaches. This strategy ensures language-independent AOP, and provides the rich joinpoint model that AOP developers have come to expect. More importantly this strategy offers the developer facilities, such as debugging expressiveness, currently unique to source-code level AOP.

3 SourceWeave.NET

SourceWeave.NET is a cross-language source code weaver that allows modularisation and re-composition of crosscutting concerns independent of implementation language. SourceWeave.NET supports an extensive joinpoint and aspect model based on AspectJ, but does not require extensions to any .NET languages. Each component, whether base or aspect, is declaratively complete. A separate XML specification describes the pointcuts and advice necessary for weaving.

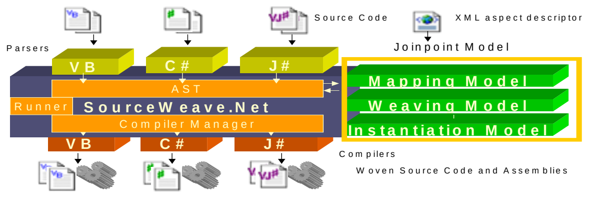


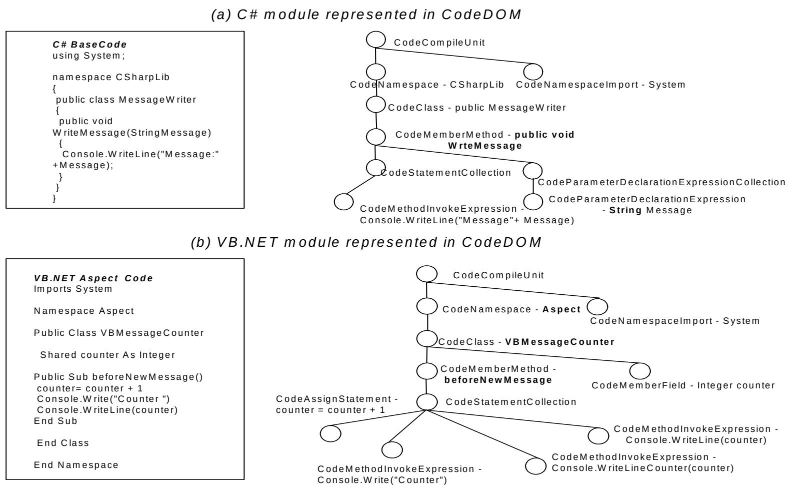
Figure 2: SourceWeave.NET Architecture

In the following sections we will describe the architecture of SourceWeave.NET as shown in Figure 2. The main components of the architecture are: a component which contains a set of language parsers that converts all input source code to an AST, a joinpoint model component that weaves the input components and aspects, and a compilation component that compiles the woven code into assemblies.

3.1 AST & Parsers

It is the responsibility of the parsers and AST component to produce an object graph representation of the input source code. The Joinpoint Model component works with these object graph representations to perform its weaving process. There is a different parser for each language used, each of which converts the input code to the same object graph format. The object graph format we used is a .NET CCL component called CodeDOM (Code Document Object Model) [19]. The CodeDOM namespace provides interfaces and classes to represent the structure of a source code document, independent of language, as an object graph or tree.

Figure 3 demonstrates how source code is abstracted onto an object graph. The root of all elements of the graph is a `CodeCompileUnit`, and every source code element is linked to `CodeCompileUnit` in a hierarchical containment structure (i.e. a `CodeCompileUnit` contains `CodeNamespace`es which in turn contain `CodeClasse`es). Figure 3(a) illustrates an example of a C# program that has a method `WriteMessage` in class `MessageWriter` that writes a message to the console. Figure 3(b) illustrates an example of a VB program that has a method `be-foreNewMessage` in class `VBMessageCounter` which writes an integer counter to the console. In the corresponding CodeDOM graphs we can see how the differing source code syntax maps to the similar abstracted types in CodeDOM.


(a) C# module represented in CodeDOM
CodeCompileUnit
CodeNamespace - CSharpLib
CodeNamespaceImport - System
CodeClass - public MessageWriter
CodeMemberMethod - **public void WrteMessage**
CodeStatementCollection
CodeParameterDeclarationExpressionCollection
CodeMethodInvokeExpression - Console.WriteLine("Message"+ Message)
CodeParameterDeclarationExpression - **String** Message
(b) VB.NET module represented in CodeDOM
CodeCompileUnit
CodeNamespace - **Aspect**
CodeNamespaceImport - System
CodeClass - **VBMessageCounter**
CodeMemberMethod - **beforeNewMessage**
CodeMemberField - Integer counter
CodeAssignStatement - counter = counter + 1
CodeStatementCollection
CodeMethodInvokeExpression - Console.WriteLine(counter)
CodeMethodInvokeExpression - Console.WriteLineCounter(counter)
CodeMethodInvokeExpression - Console.Write("Counter")

C# BaseCode
```
using System;

namespace CSharpLib
{
 public class MessageWriter
 {
  public void
WriteMessage(StringMessage)
  {
   Console.WriteLine("Message:"
+Message);
  }
 }
}
```

VB.NET Aspect Code
```
Imports System

Namespace Aspect

Public Class VBMessageCounter

  Shared counter As Integer

Public Sub beforeNewMessage()
 counter= counter + 1
 Console.Write("Counter ")
 Console.WriteLine(counter)
End Sub

 End Class

End Namespace
```

Figure 3: CodeDOM Source Code Abstraction

3.2 Joinpoint Model

In AOP, joinpoints are described as "well-defined places in the structure or execution flow of a program where additional behaviour can be attached" [9]. Aspect languages have "three critical elements: a join point model, a means of identifying join points, and a means of affecting implementation at join points" [8]. This section describes the joinpoints supported in SourceWeave.NET. Section 3.3 describes the AOP language used to identify joinpoints. Section 3.4 describes how SourceWeave.NET identifies joinpoints and affects implementation at joinpoints. First though, we look at the AspectJ joinpoints that SourceWeave.NET supports.

Joinpoint Category	Joinpoint types	Supported
Execution	method execution	Yes
	Constructor execution	Yes
	handler execution	Yes
	initializer execution	No
	staticInitializer execution	No
	object initialization	No
Call	method call	Yes
	constructor call	Yes
	object pre-initialization	No
Field Access	field reference	Yes
	field assignment	Yes

Table 1: Supported Joinpoints

As listed in Table 1, SourceWeave.NET handles most of the joinpoints supported by AspectJ. Those that are not were deemed to be so similar to other ones that they did not require further research. Joinpoints exist at run-time during program execution. In the following sections we will show how SourceWeave.NET enables joinpoint identification and how implementation is affected at joinpoints.

3.3 AOP Language

A developer wishing to leverage AOP must have a means of identifying joinpoints and specifying the behaviour modification required at identified joinpoints. SourceWeave.NET allows the developer to achieve this through an XML AOP language, based on the AspectJ language, which was introduced in Weave.NET [11]. The AspectJ language is an AOP extension of the java language. As raised in section 2.3, it would be impractical to follow the same strategy in .NET. To avoid language extension the developer identifies joinpoints through an XML pointcut specification language which refers to OO source code.

In the XML schema that defines the AOP language used in SourceWeave.NET, an aspect is considered to be analogous to a class, and methods analogous to advice. XML tags define advice and pointcuts, with refinements to capture the varying subtleties required. See [12] for a detailed specification.

3.3.1 Pointcuts

Pointcut Category	Primitive	Supported
Signature and type-based	execution(signature)	Yes
	call(signature)	Yes
	get(signature)	Yes
	set(signature)	Yes
	handler(type pattern)	Yes
	initializer(type pattern)	Yes
	staticInitializer(signature)	Yes
	object initialization(signature)	Yes
	within(type pattern)	Yes
	withincode(signature)	Yes
Control Flow	cflow(pointcut)	Yes
	cflowbelow(pointcut)	Yes
Context	this(type pattern/variable)	Yes
	target(type pattern/variable)	Yes
	args(type pattern/variable)	Yes

Table 2: Supported Pointcuts

SourceWeave.NET supports primitive signature and type-based pointcuts, control flow pointcuts, contextual pointcuts and compositions of pointcuts. Table 2 lists the primitive pointcuts that SourceWeave.NET supports.

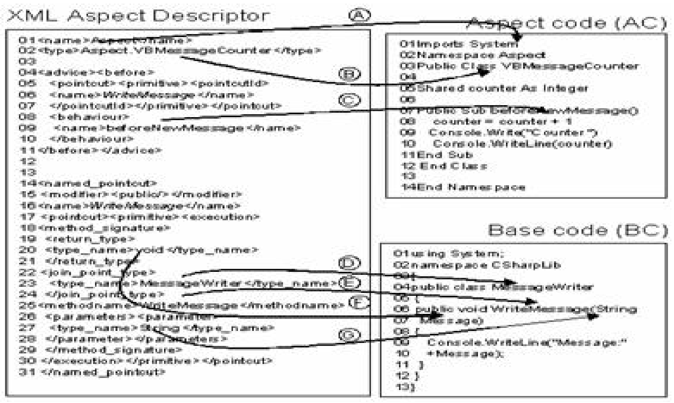
Signature and type-based pointcuts identify joinpoints through matching code segments that correspond to the code features described in the pointcut. SourceWeave.NET, like AspectJ, also supports the use of property type-based pointcuts. This is where pointcuts are specified "in terms of properties of methods other than their exact name" [35]. Control flow based pointcuts relate to other pointcuts and not signatures. These pointcuts select joinpoints in the control flow of joinpoints from its related pointcut. Contextual pointcuts expose part of the execution context at their joinpoint. The "context" is information that can be derived at run-time when the code is executing, and can be used within advice. Finally, in SourceWeave.NET pointcuts can be composed with logical `and`, `or` and `not` operators, to form composite pointcuts.

3.3.2 Advice

A developer must also have a means of associating advice with pointcuts, to specify what behaviour change should occur at a joinpoint that matches the associated pointuct. The AOP language employed by SourceWeave.NET supports three types of advice `before`, `after` and `around`. Each advice type denotes how advice is woven at a joinpoint. `Before` for example denotes that the advice behaviour should be introduced before joinpoint execution.

3.3.3 XML Aspect Descriptor

Figure 4 and its accompanying description in Table 3 provide a practical example of the AOP language used in SourceWeave.NET. In the XML aspect descriptor there is a declaration of a pointcut and advice associated with that pointcut. The arrows (in Figure 4) drawn between the XML aspect descriptor and Base/Aspect code (described in Table 3) illustrate the relationship between the XML AOP language and source code. The aspect code is written in VB.NET and the base code is written in C#. The XML advice declaration, typing and pointcut association is separated from source code to ensure AOP language neutrality. Here we see how the XML AOP language brings advice declaration and actual advice implementation together. The example also depicts how the language enables the developer to identify execution points in the program that they wish to crosscut with advice behaviour. This is achieved in this example through the declaration of typed-based pointcut expressed here as a series of XML tags.



```
<name>Aspect</name>
<type>Aspect.VBMessageCounter</type>

<advice><before>
  <pointcut><primitive><pointcutId>
  <name>WriteMessage</name>
  </pointcutId></primitive></pointcut>
  <behaviour>
   <name>beforeNewMessage</name>
  </behaviour>
</before></advice>


<named_pointcut>
 <modifier><public/></modifier>
<name>WriteMessage</name>
 <pointcut><primitive><execution>
<method_signature>
 <return_type>
 <type_name>void</type_name>
 </return_type>
<join_point_type>
  <type_name>MessageWriter</type_name>
 </join_point_type>
<methodname>WriteMessage</methodname>
 <parameters><parameter>
  <type_name>String</type_name>
 </parameter></parameters>
 </method_signature>
</execution></primitive></pointcut>
</named_pointcut>
```

```
Imports System
Namespace Aspect
Public Class VBMessageCounter

Shared counter As Integer

Public Sub beforeNewMessage()
  counter = counter + 1
 Console.Write("Counter ")
  Console.WriteLine(counter)
End Sub
End Class

End Namespace
```

```
using System;
namespace CSharpLib
{
public class MessageWriter
{
public void WriteMessage(String
 Message)
{
   Console.WriteLine("Message:"
   +Message);
 }
}
}
```

Figure 4: Cross-language Weaving Specification

Table 3 describes the relationship between the XML elements and the Aspect and Base source code presented in Figure 4.

Arrow	XML Line	Code Line	Description
A,B	02	AC 02-03	Arrow A from XML line 02 to AC line 02 and Arrow B to line 03 in the AC code emphasises a match between the declaration of the aspect `Aspect.VBMessageCounter` to the `Aspect` namespace and `VBMessageCounter` class.
C	04-11	AC 07	Arrow C identifies a method named `beforeNewMessage` the signature of which is at line 07 of the AC as advice. The `<before>` tag on line 04 indicates the advice type.
D	20	BC 06	Arrow D from line 20 to line 06 of the BC illustrates a match between the return type of the execution pointcut and the BC.
E	23	BC 04	Arrow E leading from line 23 to 04 of the BC shows the identification of the `MessageWriter` type in which joinpoint may be crosscut.
F	22	BC 06	The `WriteMessage` method name as we see, declared on line 22, identifies the method of the same name on line 06 of the BC. This is again highlighted by arrow F that links the identical names
G	27	BC 06	Arrow G from line 27 to 06 of the BC matches the `String` parameter specified in the pointcut.

Table 3: XML Aspect Descriptor – separating AOP constructs and .NET code

3.4 Weaver Implementation

The joinpoint model and AOP language supported in SourceWeave.NET is explained in sections 3.2 and 3.3. In this section we illustrate how this model has been implemented.

3.4.1 Mapping Model

The mapping model component is responsible for identifying all of the joinpoint shadows in the CodeDOM representation of the base program that correspond to the joinpoints that may match a given pointcut. First, a DOM is created based on the XML Aspect Descriptor file, this is wrapped by a series of interfaces, presented in Figure 5, that map to the XML schema and enables simplified usage of the weaving specification. For example, the `Aspect` type enables identification of the class in which the crosscutting behaviour is modularised. The `Aspect` type references `Pointcut` and `Advice` types.

Pointcuts match joinpoints at run-time. To enable run-time joinpoint generation the mapping model must firstly identify joinpoint shadows. As discussed in Section 2.1 joinpoint shadows [14] are areas in program text where joinpoints may originate. The declared characteristics of pointcuts are used to discover joinpoint shadows in the CodeDOM AST. Once generated, the CodeDOM graph is traversed and the instances of the `JoinpointShadow` type are created at points where joinpoints could be generated. Instances of the `pointcut` type containing the joinpoint identification criteria are used on a subsequent `JoinpointShadow` traversal. At each joinpoint shadow the pointcut is compared against the relevant CodeDOM information obtained through the JoinpointShadow interface. Where comparisons are positive a PointcutToJoinPointBinding is created to represent the match.

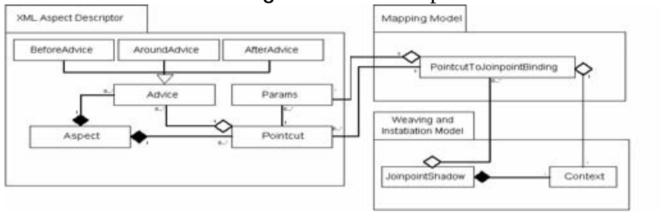


Figure 5: Mapping Model Structure

Joinpoint shadows are represented as types in SourceWeave.NET, as illustrated in Figure 6. An execution joinpoint shadow, for example, is expressed as a MethodExecutionJoinpointShadow type. Each JoinpointShadow type references and wraps a CodeDOM object which the joinpoint shadow represents. For example the, MethodExecutionJoinpointSadow and wraps the CodeDOM CodeMemberMethod.

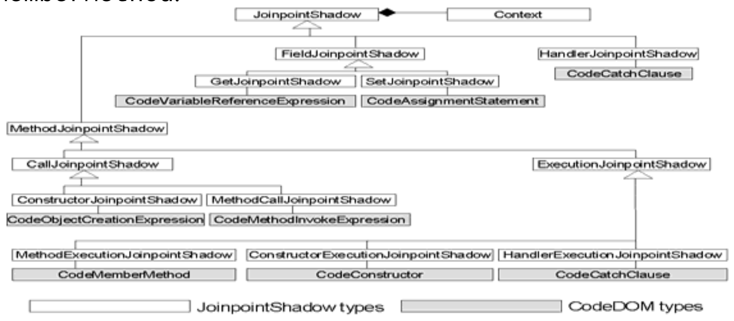


Figure 6: Joinpoint Shadow Model

3.4.1.1. Signature and Type-based Pointcuts

Providing mappings to signature and type-based pointcuts is relatively straightforward. SourceWeave.NET matches the attributes exposed by the CodeDOM objects against the signatures declared within the XML signature pointcut representations.

Returning to our example presented in Section 3.3.3, a signature based MethodExecutionJoinpointShadow which wraps a CodeMemberMethod object that represents the method void MessageWriter.WriteMessage(String Message). The CodeMemberMethod is an object created when the source code is parsed to CodeDOM (see Figure 3). The pointcut in Figure 4 describes the features held in the MethodExecutionJoinpointShadow instance which when matched against the instance of the Pointcut type, representing the declared pointcut, results in a match and the creation of a representative PointcutToJoinPointBinding.

3.4.1.2. Control Flow Pointcuts

To identify joinpoints based on cflow pointcuts, we insert Boolean flags (initially `false`) into the aspect class CodeDOM image to represent the `cflow`. This amounts to a CodeDOM `CodeConditionStatement` representing the test being inserted into the CodeDOM representation. When the joinpoint matches the `cflow`'s associated pointcut, then this flag is activated. Activation takes the form of an invocation at the joinpoint of an inserted CodeDOM `CodeAssignStatement`, where the cflow flag is set to true. At all joinpoint shadows the state of the cflow flags are tested. Joinpoints are then matched by checking to see if they are part of any cflow. At run-time, this is achieved by querying the aspect's cflow activation flags or execution of the inserted `CodeConditionStatement`. If a flag is true then the joinpoint shadow has matched the cflow otherwise there is no match. This joinpoint can then be crosscut by the advice associated with cflow pointcut.

3.4.1.3 Contextual Pointcuts

Context pointcut matching is achieved by matching the type information held within the CodeDOM objects that represent signature based joinpoint shadows. For example, from `MethodExecutionJoinpointShadows`'s encapsulation of the `CodeMemberMethod` instance, this instance exposes a `CodeParameterDeclarationExpressionCollection`. The parameter types can be examined for matching against types declared in an `args` pointcut. The context information is represented in the joinpoint shadow as instances of the `Context` type and is matched against `TypedFormalParameters`, which represents the XML declaration of context pointcuts.

Context is exposed at the method execution joinpoint shadow representing the `WriteMessage` method shown in the base code section of Figure 4. This method contains a `String` parameter named `Message` seen at lines 06-07. The method is also declared within the scope of the `MessageWriter` class. The objects of those types can be exposed to the crosscutting behaviour at runtime by using the `args`, `this` or `target` pointcuts. For example, if the XML described a pointcut of the form `String messages:args(message)` the joinpoint shadow would match the pointcut and the message would be exposed to the crosscutting behaviour.

3.4.2 Weaving and Instantiation Model

Once the mapping model has matched all joinpoint shadows in the target code against the pointcuts and the proper bindings have been created, the next step is to affect implementation at the bound joinpoint shadows. The weaving model intersects the matched joinpoint shadows with the appropriate advice for the related pointcut. Each `JoinpointShadow` type encapsulates how advice should be woven at the source code point that it represents. Advice types `before`, `after` and `around` are modelled as types `BeforeAdvice`, `AfterAdvice` and `AroundAdvice`. These types and their relationships with other types used by SourceWeave.NET can seen in

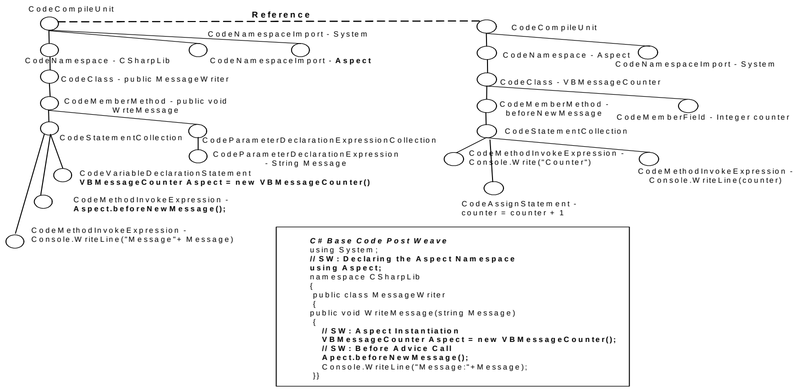
Figure 5. Advice types encapsulate information about the advice methods within the class identified as an aspect by the XML Aspect Descriptor.

SourceWeave.NET implements two types of weaving – direct and indirect. With direct weaving, a joinpoint shadow has matched a pointcut and there is advice associated with that pointcut. This advice is the behaviour that is to be introduced at the CodeDOM source code representation in a manner determined by the type of the advice. Indirect weaving is required for control flow joinpoints. Because we can only know at run-time the joinpoint shadows that need to be affected due to `cflow` membership, abstracted code representations are inserted at every joinpoint shadow to check for `cflow` membership and depending on a positive outcome at run-time, behaviour is then introduced at the joinpoint. In either scheme, as the crosscutting code is weaved into the base source code a set of comments is also interwoven around injected code. These comments give a clear and precise understanding of what is happening in the generated advice call and why this code was injected, with reference to the Aspect XML Descriptors weaving specification. This commenting strengthens the provided debugging expressiveness as the developer will be able to gain a quick appreciation of what is occurring in the weaved code.

There are a number of steps required for both direct and indirect weaves. Firstly, if the aspect class is in another `CodeCompileUnit` (i.e. a different aspect or base code module) then a reference between the `CodeCompileUnit` and the aspects `CodeCompileUnit` is made. This ensures that the compiler can find the types used in the woven code at compile-time. Then, the list of imports or using statements is amended if the aspect's namespace is not present. An aspect declaration and instantiation of the aspect are required.

To support debugging expressiveness we used a different instantiation model to AspectJ. An instantiation model dictates when instances of aspects will be created to support the execution of the crosscutting behaviour that is encapsulated by that aspect. Debugging expressiveness partially relates to the investigation of state changes based on operations within a component. Per-advice instantiation involves instantiation of the aspect containing the advice at each advice call. As there is a new instance for each call, there is no scope to maintain any state across calls (and indeed, no corresponding need to debug state changes across calls). However, this restriction does not cater for many requirements, for example, the message counter from Figure 4. AspectJ also has a singleton instantiation model that we considered, where one instance of the aspect is shared through out the system. However, this approach is very state heavy for debugging purposes. SourceWeave.NET's instantiation model is a compromise between the two, and follows a per-method instantiation strategy. A per-method model means the aspect is instantiated once in every method in which advice on that aspect is called. We also investigated instantiation models which were less granular, such as per-class instantiation, but found the flow based state change on a per-method basis more intuitive for a developer when debugging. As well as tracing state changes based on control flows, we also found that the generated source code looked much cleaner when using the per-method instantiation model, which again supports debugging expressiveness.

Once the aspect declaration and instantiation has been inserted into the CodeDOM method representation, we can then insert advice calls on the aspect. Depending on the insertion rules that vary depending on joinpoint type, a CodeDOM `CodeMethodInvokeExpression` is created to introduce the advice at the joinpoint. In this call there may be context information required which is added to the call in the form of arguments that correspond to parameters on the advice method being targeted in the call.


CodeCompileUnit
Reference
CodeNamespaceImport - System
CodeNamespace - CSharpLib
CodeNamespaceImport - Aspect
CodeClass - public MessageWriter
CodeMemberMethod - public void WrteMessage
CodeStatementCollection
CodeParameterDeclarationExpressionCollection
CodeParameterDeclarationExpression - String Message
CodeVariableDeclarationStatement VBMessageCounter Aspect = new VBMessageCounter()
CodeMethodInvokeExpression - Aspect.beforeNewMessage();
CodeMethodInvokeExpression - Console.WriteLine("Message"+ Message)
CodeCompileUnit
CodeNamespace - Aspect
CodeNamespaceImport - System
CodeClass - VBMessageCounter
CodeMemberMethod - beforeNewMessage
CodeMemberField - Integer counter
CodeStatementCollection
CodeMethodInvokeExpression - Console.Write("Counter")
CodeMethodInvokeExpression - Console.WriteLine(counter)
CodeAssignStatement - counter = counter + 1

```
C# Base Code Post Weave
using System;
// SW : Declaring the Aspect Namespace
using Aspect;
namespace CSharpLib
{
 public class MessageWriter
 {
public void WriteMessage(string Message)
 {
   // SW : Aspect Instantiation
   VBMessageCounter Aspect = new VBMessageCounter();
   // SW : Before Advice Call
   Apect.beforeNewMessage();
   Console.WriteLine("Message:"+Message);
 }}
```

Figure 7: Woven Code

Figure 7 illustrates the result of weaving the example introduced Section 3.1, expanded in Section 3.3 and referenced through section 3.4.1. The highlighted code represents code that has been woven into the base program. The first thing to note is the introduction of a `using` statement. Within the `WriteMessage` method we can see the aspect type `VBMessageCounter` being declared and also instantiated. Then on the next line the before advice is called on the class identified as an aspect. With the inserted code we can also see the inserted comments which explain the inserted aspect instantiation and advice call.

3.4.3 Compilation

By now, we have a woven CodeDOM abstract representation of the code. The next step is to compile this code into assemblies. Inter-compilation-unit dependencies are created when types in one compilation-unit are referenced in another. These dependencies affect the order in which source code units can be compiled. For example if we are trying to compile source unit A that references types held in (as yet un-compiled) source unit B, then A will not compile as the compiler cannot find the types that A is using. The SourceWeave.NET compiler manager ensures a compilation order that avoids compiler error.

To provide the developer with a consistent view of the potentially cross-language code, each `CodeCompileUnit` unit is converted back to the language in which it was originally expressed and is available for the developer to inspect. A compiler for the original language compiles the `CodeCompileUnit` in to an assembly with the debug information particular to that language. In this manner, the developer can debug the woven code in the original source code language in which it was written.

CodeDOM provides a set of tool interfaces for CodeDOM related tasks such as compilation and source code generation from the CodeDOM object graph.

4 Discussion

The motivation for the development of SourceWeave.NET was to investigate cross-language weaving possibilities when working with source code. We wanted to research the extent to which we would have to compromise the joinpoint model in order to support *cross-language* weaving of source code. Since .NET achieves language independence by being very standards driven, we wanted to stay within the .NET standards as far as possible. In so doing, we were able leverage .NET facilities such as CodeDOM and Reflection. In this section, we assess the benefits and limitations of this approach.

4.1 Benefits

The benefits that SourceWeave.NET gives to the developer are: an extensive joinpoint model, support for cross-language development of base code and aspect code, and debugging expressiveness.

- *Joinpoint model*: SourceWeave.NET supports an extensive range of joinpoints described in section 3.2, a comprehensive AOP language described in section 3.3. The separation of the XML AOP language from the CLS also means that code and crosscutting specifications are more reusable and flexible. Moreover, the .NET standards are not altered and developers can express their code in standard .NET languages.
- *Cross language-support*: SourceWeave.NET's weaver works on CodeDOM representation of source code. As part of the .NET Base Class Library, CodeDOM is a standard representation for languages that conform to the CLS. Notwithstanding limitations we will discuss in the next section, any language that provides a parser to CodeDOM can be weaved by SourceWeave.NET. We have evaluated this for C#, VB.NET, and J#.
- *Debugging expressiveness*: SourceWeave.NET weaves at source code level, and as such does not affect the bonds created in assemblies for debugging. During weaving, comments are inserted into the code at weave points, and other interesting points of change within the code, to explain what has happened at that point. The developer will be able to trace through the execution of the woven code in the language in which it was written, where there will be comments to indicate the crosscutting nature of the behaviour. To our knowledge, there is no other approach to cross-language weaving that can achieve this.

4.2 Limitations

SourceWeave.NET also has some limitations, these include: a dependency on CodeDOM, which currently suffers some limitations in expressiveness and an inability to deal with circular CodeDOM dependencies.

- *CodeDOM dependency*: If a .NET language does not have a parser to CodeDOM, then SourceWeave.NET cannot weave code written in that language.

- *CodeDOM limitations:* In its current implementation, CodeDOM has limitations in terms of its expressiveness. C# constructs map reasonably well to CodeDOM, but that is not true of all .NET languages' constructs. For example, Eiffel's preconditions, postconditions and invariants are not supported [5]. The following constructs are also not supported [16, 37, 38]: Nested namespaces, variable declaration lists, namespace aliasing, unary expressions, inline conditional expressions, calling base constructors and nested or "jagged" arrays, safety modifiers readonly and volatile, structural safety modifiers virtual and override, event operator members and destructors, add and remove accessors for events, attribute targets and accessors on attributes, unsafe keyword used to denote unmanaged code, break and continue in case statements.
- *Compilation dependency*: SourceWeave.NET is limited because of compilation dependency issues. For example, take three compilation units A, B and C. If A references B and a reference to C is woven into A when B references C, then a circular dependency results. None of these compilation units of source code can compile as each compilation unit needs another to be compiled for itself to be compiled.
- *Source code dependency*: As SourceWeave.NET requires access to source code, precompiled, third-party components may not be included in the weaving process.

Two other areas of SourceWeave.NET are currently being addressed. The first is the use of XML for the aspect descriptor. XML was chosen originally because it appeared to be emerging as the de facto standard description language. Currently, SourceWeave.NET developers have to write this XML by hand, which is tedious. We are working on a wizard-like mechanism, similar to that available for EJB Deployment Descriptors [39] that will generate the required aspect descriptor. We also plan to consider languages other than XML, which may be more readable. The second area of concern is with supporting control flow based joinpoints. Nested control flow joinpoints are not yet supported. In the current implementation we are concerned with the performance of the system when supporting control flow activated joinpoints. Our initial attempt was to use Boolean flags inserted into the CodeDOM aspect representations to later determine at each joinpoint shadow whether a `cflow` is activated and matches a particular joinpoint. This is a solution, but we consider it to be a brute-force effort. There are significant refinements possible, such as code analysis to reduce the set of joinpoint shadows that could possibly be within a particular cflow, which we expect will considerably improve the performance.

5 Related Work

There has been a proliferation of approaches to applying AOP to .NET in recent years. We describe them here in the categories outlined in Section 2, and, where appropriate, assess them against criteria similar to those outlined for SourceWeave.NET in the previous section.

5.1. Altering the run-time environment

AspectBuilder [34] and LOOM.NET [32] are the two main approaches to introducing aspects to .NET by altering the run-time environment. AspectBuilder intercepts calls through transparent proxies that wrap targets of message calls. These transparent proxies are points at which crosscutting behaviour can be introduced during execution. .NET custom metadata are created that act as explicit joinpoints. Custom metadata (or programmer-defined *attributes*) allows a programmer explicitly identify joinpoints. As modified by AspectBuilder, the CLR uses the extended metadata to distinguish points in the execution where normal execution should be suspended and modularised crosscutting behaviour should execute. LOOM.NET also exploits metadata exposed in assemblies to declare joinpoints, modifying the CLR to recognise where crosscutting behaviour should execute. AspectBuilder and LOOM.NET have the significant advantage of being inherently language neutral. All languages that can execute on the CLR are subject to aspect weaving.

We also perceive some drawbacks. Using custom metadata, or attributes, requires an intrusive joinpoint model, which means the programmer must explicitly declare every joinpoint in the base code.

One issue relating to this run-time AOP is the potential for performance degradation. Although, to our knowledge, performance has not been measured within .NET, AOP research on the Java [1] platform reveals AOP implemented in the JVM degrades run-time performance over AspectJ, which is a byte-code based weaver. The similarity in platform architecture of .NET and Java suggests that an AOP run-time implementation would have similar results.

Another issue is the limited joinpoint models currently demonstrated. For example, the joinpoint model in AspectBuilder cannot modularise concerns that crosscut the structures internal to a component interface. The joinpoint model employed by an approach determines its ability to modularise crosscutting concerns, and so an extensive joinpoint model is an important goal for an AOP approach.

Finally, supporting debugging facilities appears difficult, and has not been demonstrated in approaches following this strategy. This is because the .NET run-time environment is altered, which breaks conformance to the standards. Debuggers within a framework such as .NET are dependant on the run-time as this is where execution occurs [27]. A standard debugger expects a run-time environment that conforms to standards similar to that of the debugger. When the run-time is altered this can break the assumptions the debugger was created on. Alteration then prevents correct debugging.

5.2. Weaving at IL level

There have been a number of attempts to introduce AOP concepts through IL manipulation. Weave.NET [11], AOP.NET [29] and its predecessor AOP# [33], and CLAW [13] weave IL components. LOOM.NET [32] also has a version that weaves IL. For each, weaving is based on information that identifies components with crosscutting behaviour and characterises how the crosscutting behaviour should be distributed across other IL components. CLAW and AOP.NET are run-time weavers that manipulate IL pre-execution time. They use the CLR events exposed through the unmanaged profiling interfaces [24] to trace the execution of a program in the CLR. Unmanaged profiling interfaces allow observation of the CLR, including the loading of code. When the weaver is notified of the activation of a joinpoint, it adds the relevant aspect code to be executed. The IL version of LOOM.NET (an update to Wrapper Assistant [30]) creates component proxies to act as points at which aspect behaviour can be introduced. The joinpoint model is GUI driven, where joinpoints are selected intrusively through a user interface [31]. LOOM.NET supports an aspect specific template mechanism that was initially described in [30]. These aspect templates are structures that have been customised to represent a particular type of concern, fault-tolerance is the example most used. Aspect specific templates provide mechanisms to match declarative expressions in components to macros which create C# aspect framework proxy code. The user can then write their aspects in C# to fit into this framework. Weave.NET delivers the most extensive joinpoint model of these approaches. It is implemented as a managed .NET component that works at load-time, utilising the .NET reflection libraries to both read and generate assemblies. It also has a specialised IL reader that can read IL to statement and expression levels to discover joinpoints internal to the class construct. This allows it to support more expressive joinpoints than signature and type-based ones, such as joinpoints that expose context.

Approaches that weave at IL level have the advantage of cross-language aspect support. It has yet to be proven, though, how extensive a joinpoint model is possible. Only Weave.NET has achieved close to the set listed in Table 1. There is also a dependency on having assemblies that conform to the CLI. This has shown itself to be restrictive because, for example, assemblies generated by the J# compiler do not expose the standard metadata. This means that IL-based approaches cannot work with J# components. Finally, programmers cannot debug woven source code. For debugging purposes, source code is bound to the IL it relates to at compile time. By manipulating the IL, the bond forged at compile-time is disturbed, removing the binding to the original source. IL weavers do expose their woven IL but developers may not understand IL, or even want to. Adding comments to the IL to help programmers is currently not possible, as comments may not be expressed in IL.

5.3. Working with source: Extending the CLS and CTS

Each .NET compiler converts source code that conforms to the CLS to standard IL. To support a cross-language AOP the CLS and CTS must be extended to specify AO constructs. This alteration would be difficult, as it would require extending the standard, and enforcing every language that conforms to the standard to include compiler support for the AOP constructs. EOS (formerly IlaC#) and Aspect.NET both take the approach of extending the compilers. Both target the C# compiler and are not focused on seeking language independence. The focus of EOS is instance-level aspect weaving described in [25] and [26].

5.4. Working with source: Unchanged CLS

Working with source code and weaving pre-compile time was an approach first taken by AspectC# [6]. AspectC#, which is the predecessor of SourceWeave.NET, works with C# source code. Like SourceWeave.NET, it expresses AO constructs in XML, superimposing crosscutting behaviour that is modularised in OO classes. AspectC# was an initial proof of concept project that demonstrated, using a primitive joinpoint model, the possibilities for weaving cross-language source code. It provided an initial platform on which SourceWeave.NET was developed, showing us that decoupling AO constructs from base and aspect languages and expressing the AO concepts in a language neutral fashion, avoids the need to alter the CLS.

6 Summary and Conclusions

SourceWeave.NET has achieved language-independent AOP through cross-language source level weaving. No extensions have been added to any language, as weaving specifications are separated from the input components through the use of XML. Because of this, and because the architecture weaves an AST representation of source code, we can easily support programmers stepping through woven code in the language in which it was written for debugging purposes. A high level of debugging expressiveness in a multi-language environment is becoming increasingly important because of the new challenges emerging for application development in mobile, context-aware, pervasive environments. In such environments, systems are composed of many interacting devices that are driven by software that may be written in various languages, with highly dynamic state changes that are difficult to anticipate prior to run-time. SourceWeave.NET provides an environment that caters for such application development. Its multi-language environment supports a level of complexity management through AOP, and that also supports the debugging expressiveness that developers need to fully assess the operation and possible states of their inter-woven components.

Multi-language debugging expressiveness has been achieved without sacrificing the joinpoint model that is employed, or limiting (at some level) the languages that can be supported. From the perspective of the joinpoint model, SourceWeave.NET supports signature and type-based pointcuts, control flow pointcuts, contextual pointcuts, and composite pointcuts. As for supported languages, any language for which there is a parser to CodeDOM is able to participate in the AOP model. We have evaluated this for C#, VB.NET and J#.

SourceWeave.NET uses .NET's CodeDOM as the AST representation of base and aspect source code. This has turned out to be both a strength and a weakness. It has contributed to the reason why programmers can step through woven code's source for debugging purposes. However, it has limitations in terms of its current expressiveness, which means that there are a number of language constructs that are not yet supported. We considered extending CodeDOM, but this would alter the standardised framework. We have, however, received communication that the relevant standards committees will extend its specification to be as expressive as it is advertised.

Nonetheless, our approach has considerable merit. The ability to step through source code in the language in which it was written is a strong requirement for cross-language AOP approaches. It is difficult to see how approaches that extend the run-time environment can achieve this. On the other hand, if IL weavers could be written to update the source to IL binding created for debugging purposes as part of the weaving process, it seems possible that a similar approach could work for IL. We plan to continue with our refinements to SourceWeave.NET, such as improving how `cflow` is handled, and making the XML aspect descriptors more accessible. We are also interested in incorporating ideas such as joinpoint encapsulation [10] and instance-level aspect weaving [25, 26]. SourceWeave.Net has the potential to provide an AOP environment that makes it easy for programmers to work directly with cross-language source code as written, while still providing support for an exhaustive AOP model.

References

1. Chiba, S., Sato, Y., Tatsubori, M.: Using HotSwap for Implementing Dynamic AOP Systems, ECOOP'03 Work-shop on (ASARTI) Advancing the State of the Art in Runtime Inspection, Darmstadt, Germany (2003)
2. ECMA International.: Standard ECMA-335 Common Language Infrastructure (CLI), ECMA Standard (2003)
3. Gal, A., Mahrenholz, D., Spinczyk, O.:AspectC++, http://www.aspectc.org/contact.html, (2003)
4. Hirschfeld, R.: AspectS – Aspect-Oriented Programming with Squeak. In M. Aksit, M. Mezini, R. Unland, editors, Objects, Components, Architectures, Services, and Applications for a Networked World, pp. 216-232, LNCS 2591,Springer (2003)
5. Holmes, J.: Using the CodeDOM to Wrap, Extend and Generate, Assemble and Load New Code on the Fly, VSLIVE, Visual Studio Developer Conference, Orlando, USA (2003)
6. Howard, K.: AspectC#: An AOSD implementation for C#. M.Sc Thesis, Comp.Sci, Trinity College Dublin, Dublin (2002) TCD-CS-2002-56
7. Kiczales, G., Coady.Y.: AspectC, www.cs.ubc.ca/labs/spl/projects/aspectc.html (2003)
8. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: Getting Started with AspectJ Communications of the ACM, 44 (10), (2001) 59-65.
9. Kiczales, G., Lampoing, J., Mendhekar, A., Maeda C., Lopez, C., Loingteir, J., Irwin, J.: Aspect-Oriented Programming, In proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland (1997)
10. Koreniowski, B.J.: Microsoft.NET CodeDOM Technology, http://www.15seconds.com-/issue/020917.htm (2003)
11. Lafferty, D.: Language Independent Aspect Oriented Programming, In proceedings of the Object-Oriented, Programming, Systems, Languages (OOPSLA), California, USA (2003)
12. Lafferty, D.: W3C XML Schema for AspectJ Aspects, XML Schema, http://aosd.dsg.cs.tcd-.ie/XMLSchema/aspect_Schema.xsd (2002)
13. Lam, J.: Cross Language Aspect Weaving, Demonstration, AOSD 2002, Enschede, (2002)
14. Masuhara, H., Kiczales, G. and Dutchyn, C.,A Compilation and Optimization Model for Aspect- Oriented Programs, In Proceedings of Compiler Construction (CC2003), LNCS 2622, pp.46-60, 2003.
15. Meijer, E., Gough, J.: Overview of the Common Language Runtime, Microsoft (2000)
16. Mercy, G.P.: Dynamic Code Generation and CodeCompilation, C# Corner, http://www.c-sharpcorner.com/Code/2002/Dec/DynamicCodeGenerator.asp (2002)
17. Mezini, M., Osterman, K.: Cesar, http://caesarj.org/, (2003)
18. Microsoft: Assemblies Overview, .Net framework developer's guide, http://msdn.microsoft. com/library/default.asp?url=/library/en-us/cpguide/html/cpconassembliesoverview.asp

(2003)
19. Microsoft.: CodeDOM, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemcodedom.asp (2003)
20. Microsoft.: MSDN Library, Common Class Library, http://msdn.microsoft.com/library /default.asp?url=/library/en-us/cpguide/html/cpconthenetframeworkclasslibrary.asp (2003)
21. Microsoft.: MSDN Library, Common Type System, http://msdn.microsoft.com/ library/de-url=/library/en- us/cpguide/html/cpconthecommontypesystem.asp (2003)
22. Microsoft.: .Net, http://www.microsoft.com/net/ (2003)
23. Microsoft.: MSDN Library, What is the Common language Specification ?, http://msdn-.microsoft.com/library/default.asp?url=/library/en- us/cpguide/html/ cpconwhatiscom-monlanguagespecification.asp (2003)
24. Pietrek, M.: The .NET Profiling API and the DNProfiler Tool, http://msdn.microsoft.com/-msdnmag/issues/01/12/hood/ (2003)
25. Rajan, H., Sullivan, K.: Eos: Instance-Level Aspects for Integrated System Design, In the proceedings of the ESEC/FSE, Helsinki, Finland (2003)
26. Rajan, H., Sullivan, K.: Need for Instance Level Aspects with Rich Pointcut Language, In the proceedings of the Workshop on Software Engineering Properties of Languages for Aspect Technologies (SPLAT) held in conjunction with AOSD 2003, Boston, USA (2003)
27. Rosenberg, J.: How Debuggers Work - Algorithms, Data Structures, and Architecture. John Wiley & Sons, New York, USA, (1996) 77-106
28. Safonov, V.:Aspect.NET- a Framework for Aspect-Oriented Programming for .Net platform and C# language, St, Petersberg, Russia.
29. Schmied, F.: AOP.NET, http://wwwse.fhs-hagenberg.ac.at/se/berufspraktika/2002/se99047-/contents/english/aop_net.html (2003)
30. Schult, W., 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 241-248
31. Schult, W., 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. Schult, W.: LOOM.NET, http://www.dcl.hpi.unipotsdam.de/cms/research/loom/ (2003)
33. Schüpany, M., Schwanninger, C., Wuchner, E.: Aspect-Oriented Programming for .NET. In First AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software, Enschede, The Netherlands (2002) 59-64.
34. Shukla, D., Fell, S., Sells, C.: Aspect–Oriented Programming enables better code encapsulation and reuse, http://msdn.microsoft.com/msdnmag/issues/02/03/AOP/default.aspx (2002)
35. The AspectJ Team.: The AspectJ Programming Guide (V1.0.6), http://download.eclipse-.org/ technology/ajdt/ aspectj-docs-1.0.6.tgz (2002)
36. Wichman J.C.: ComposeJ, The development of a pre-processor to facilitate Composition Filters in Java, M.Sc the sis, Comp.Sci, University of Twente, Twente (1999)
37. Wiharto,M..: Journal Entry for June 13, The Mars Project, School of Computer Science and Software Engineering, Monash University, Australia, http://www.csse.monash.edu.au/ ~marselin/archive/2003_06_08_journal_ archive.html (2003)
38. Whittington, J.: CodeDOM needs help, http://staff.develop.com/jasonw/weblog/2003/ 03/04.html (2003)
39. Xdoclet: Attribute-Oriented Programming, http://xdoclet.sourceforge.net/, (2003)