CCC: An Aspect-Oriented Intermediate Language on .Net Platform

Yingfei Xiong [1,2]

School of Computer Science and Engineering
University of Electronic Science and Technology of China
Chengdu, China

Institute of Software, Peking University
Beijing, China

Feng Wan [3]

School of Computer Science and Engineering
University of Electronic Science and Technology of China
Chengdu, China

Abstract

The AOSD tools and methodologies have developed in a rapid speed in Java area. However, no matter how homologous .Net and Java are, AOP tools on .Net platform are still in experimental stage. The complex infrastructure and multi-language support make it hard to establish aspect-oriented programming on .Net. Microsoft provides Microsoft Intermediate Language(MSIL) to facilitate language implementation on .Net platform. But MSIL doesn't support aspect-oriented languages. To facilitate aspect-oriented language implementation on .Net platform, it is best to provide an aspect-oriented intermediate language.

This paper presents CCC, an aspect-oriented intermediate language on .Net platform. CCC stands for Common language Cross Cutter. Any aspect-oriented .Net language could first be transformed to CCC and then woven by CCC compiler. Since CCC supports high-level aspect-oriented language constructs directly, such as aspect, there is a little work to do for language implementation and the developers could concentrate on language design.

Key words: CCC, AOP, .Net, intermediate language

[1] Email: flyinghero@21cn.com
[2] Fax: 86-10-62751792
[3] Email: windfast_2000@163.com

1 Introduction

Aspect-oriented programming (AOP) is an emerging technique for modularizing crosscutting concerns, which cut across several basic modules. These concerns cannot be modularized with existing techniques such as object-oriented programming.

After AOP was first announced by Gregor Kiczales in ECOOP'97, the AOSD tools and methodologies developed in a rapid speed, especially in Java area. So far, several major AOP tools in Java language, such as AspectJ[11], and Hyper/J[10], have gained wide commitment and have been used, although in a limited way, in some industrial projects.

Although.Net and Java are homologous, AOP tools on .Net platform are still in their experimental stage. For one thing, the .Net infrastructure is much more complex than Java, which makes the .Net languages, such as C#, have more language constructs than Java. Hence the mature AOP tools on Java platform can not port to .Net without redesign. For another, the .Net is language-independent in its nature. So the AOP tool on .Net platform is better to also be language-independent, which brings many new problems.

To facilitate language implementation, Microsoft provides Microsoft Intermediate Language(MSIL)[5]. This is a collection of machine-independent directives which support high-level language constructs, such as class, directly. Thus the compiler has to do a little work to transform source code into MSIL, and the Microsoft .Net framework transforms the MSIL to native code at running time.

So in order to facilitate aspect-oriented language implementation, it is best to provide an aspect-oriented intermediate language to which the aspect-oriented code can be easily transformed. This paper presents our efforts to develop such an intermediate language. Our approach is called CCC, which is an XML-based aspect-oriented intermediate language. Any other aspect-oriented language could firstly be transformed to CCC and then woven by CCC Compiler.

2 Programming Model

2.1 Specifying Aspect

One major problem in CCC language design is how to specify aspect. CCC is an intermediate language so it must be independent of any high-level language. Then the media used to specify aspect in CCC must also be independent of any high-level .Net language. Also, it must be flexible enough so that the complex elements in aspect-oriented languages could be explicitly and clearly specified.

We regard XML as suitable media for this situation. Derived from SGML, XML is language-independent in its nature. XML, as a "metalanguage", also

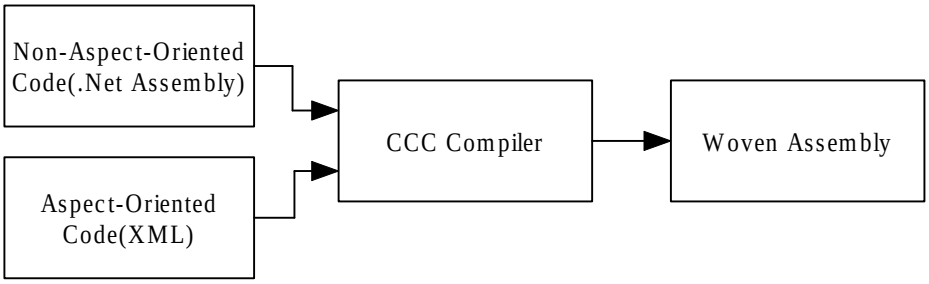

Fig. 1. Behavior of CCC Compiler

meets our second requirements. It is flexible enough to describe any language constructs.

At the mean time, with the help of XSD (eXtensible Structure Definition) [2], we can easily write a schema to automatically validate the grammar and syntax of user-written xml file, which saves a lot of code and developing time.

2.2 Run-Time, Load-Time and Compile-Time Weaving

Currently, many researches on AOP are focused on run-time weaving, and several available projects, such as RAPIER-LOOM.NET[9], have shown its feasibility. Also, MSIL is compiled to native code at run-time, the generic types in .Net 2.0 are instantiated at run-time. A run-time weaver will be compatible with the overall .Net design. So it seems reasonable for CCC to be a run-time woven language. However, current implementations of such languages are considerably slow at run-time, because the weave process adds a lot of extra work to object creation and no knowing technique can significantly lessen such work. We believe performance is an important issue to evaluate a language, so we reject this idea of run-time weaving.

Load-time weaving is another choice. Loading seems to be a better time to perform longer tasks. Some research projects are also aimed at creating a load-time weaver, such as Weave.NET[4]. But weaving at loading time means all types should be woven and the accumulative time is also unaffordable.

So we chose compile-time weaving as the weaving method of CCC, like most other aspect-oriented languages. Weaving once at compiling time will save a lot of time in assembly loading or running.

2.3 Scenarios of Using CCC

As shown in Figure 1, the input of CCC includes a .Net assembly containing the non-aspect-oriented code written in MSIL and XML files specifying aspect. CCC compiler weaves the aspect behaviour code into non-aspect-oriented assembly according to what is written in the XML files, generating a new, woven assembly.

To illustrate this idea, we give two scenarios. Consider a user programs CCC directly:

(i) The user writes the non-aspect-oriented code in existing languages, and then compiles it into an assembly.

(ii) The user writes the aspect-oriented code in several XML files according to CCC syntax.

(iii) The CCC compiler reads the XML files and the assembly, generates the woven assembly.

The second scenario is for language compiler compiling the source code with the help of CCC:

(i) The compiler reads the source code, separating them into aspect-orient part and non-aspect-oriented part.

(ii) The non-aspect-oriented part is compiled into MSIL in a temporary assembly.

(iii) The aspect-oriented part is compiled into an XML file according to CCC syntax.

(iv) The CCC compiler reads the temporary assembly and the xml file, generating the final assembly.

3 Language Syntax and Semantics

3.1 Join Point

In CCC, a join point is a well-defined point in the program structure. We define six types of join points in current version: Method Reference, Method Declaration, Field Set Reference, Field Get Reference, Type Reference, Type Declaration.

- *Method reference* represents the point on which a method is called, that is, a method is referred by the other parts of the system. The MSIL directives call and callvirt are examples of this join point.
- *Method declaration* represents the point where a method is declared. It works like the execution() pointcut in AspectJ. All method declarations, no matter private or public, are considered as method declaration join points.
- *Field set reference* represents the point where a field is being set. The MSIL directives stfld and stsfld are examples of this join point.
- *Field get reference* represents the point when the value of the field is being retrieved. The MSIL directives ldfld and ldsfld are examples of this join point.
- *Type reference* represents the point where a type is referred to. It also includes the dynamically created type such as array and managed pointer.
- *Type declaration* represents the point where a type is declared.

3.2 Pointcut

A pointcut is a set of conditions to select the join points which are of interest. In CCC, a pointcut is described by a pointcut expression. For example, the following pointcut expression selects all method reference join points where the method is declared in *MyClass* and begins with *Set*:

```
<Call><MethodNameMatches>MyClass.Set%</MethodNameMatches></Call>
```

The *Call* and *MethodNameMatches* in the above pointcut expression are called pointcut functions. Pointcut functions, which select specific types of join points, are the main component of pointcut expression. The *MethodNameMatches* pointcut function selects method reference join points or method declaration join points where the full name of the method matches its inner match pattern, which is *"MyClass.Set%"* in the above example. The *"%"* in the match pattern presents arbitrary number of characters. The *Call* pointcut function selects all method reference join points from its inner pointcut function.

A pointcut expression can be given a name, and thus is called named pointcut. Other parts of the system can refer to the pointcut using the name. This is the main way to reuse pointcuts in CCC. Continuing the above example, if we want to name the pointcut expression *"SetMyClass"*, we can write code like this:

```
<PointCut Name="SetMyClass" Accessibility="Public">
  <Call>
    <MethodNameMatches>MyClass.Set%</MethodNameMatches>
  </Call>
</PointCut>
```

The *Accessibility* attribute indicate to what extent the pointcut can be accessed. When the accessibility is *Public*, this pointcut can be referred to from any parts of the system. When the accessibility is *Private*, this pointcut can only be referred to within the aspect in which it declared.

3.3 Advice

Advice defines what will be executed at specific pointcut in execution of the program. The code to be executed is called behavior.

A problem related to behavior is where the behavior code lies. The language like AspectJ surrounded the behavior code with advice declaration, so that the code is stored together with the aspect. But this is not suitable for CCC. If we put behavior code within advice declaration, to maintain the independence of high-level languages, the only language can be used to write behavior code is MSIL. But writing MSIL directly is not convenient both for users and compilers of high-level languages.

Our approach is to let user write the behavior code within a public static method in the target assembly. This method is called behavior method. And

when declaring the advice, the user uses the method name to refer to the behavior method. The code stored in the behavior method will be used as behavior code. The reason of the method being static is that a static method can be called at anytime without object creation.

Now consider an example of advice:

```
<Advice PointCut="SetMyClass"
Behaviour="MyAspect.BeforeSetMyClass"
Type="Before"/>
```

The *PointCut* attribute is used to refer to a named pointcut already declared. Here we refer to the *SetMyClass* pointcut in the previous example. The *Behavior* attribute specifies which method will be used as behavior method. The *Type* attribute defines how the behavior code is executed at the join points. The advice has two types in this release. The before advice means the behavior code runs just before the join point picked out by the pointcut. The after advice means the behavior code runs just after the join point picked out by the pointcut, even if it throws an exception.

After this advice is declared, *MyAspect.BeforeSetMyClass* method will be called just before each call to the *Set* methods in *MyClass*.

Sometimes referring to a previously declared named pointcut is not convenient, we can also declare new, anonymous pointcut within the advice body, like this:

```
<Advice Behaviour="MyAspect.BeforeSetMyClass"
  Type="Before">
  <PointCut>
    <Call>
      <MethodNameMatches>MyClass.Set%</MethodNameMatches>
    </Call>
  </PointCut>
</Advice>
```

Advices also expose data from the execution context at the join points. This will be discussed in more detail in section 4.3.

3.4 Aspect

Aspect wraps named pointcuts and advices together to form a modular crosscutting unit. An aspect also has a name so that it can be referred to latter.

4 The Language Features of CCC

This section summarizes the features which characterize CCC different from other aspect-oriented languages. Some of the features are present because of the role of CCC: as an intermediate language. Some of the features are to solve the problems in existing languages.

4.1 Pointcuts Are as Primitive as Possible

In many aspect-oriented languages, pointcuts are not primitive. Take AspectJ as an example, the *call()* expression is not a primitive pointcut. It combines expressions of *args()* and *within()*, method name filtering and return type filtering.

The nonprimitive pointcut is easy to write, but not suitable for CCC. CCC is an intermediate language, so it should be convient to transform high-level language to CCC. But if the pointcut is nonprimitive, the transformation from some high-level languages to CCC might be difficult. For example, the aspect method in AspectC# is similar to the *call()* pointcut in AspectJ, but it does not match the method return type yet also matches the names of parameters. Such differences make it difficult to describe an aspect method of AspectC# using AspectJ's language constructs.(Although we can use "*" to represent arbitrary return type, but compiler does extra work to check the "*") So every pointcut function we defined in CCC is a primitive pointcut function. The counterparts in CCC of *call()* pointcut in AspectJ are six pointcut functions: `<Call>`, `<MethodNameMatches>`, `<DeclaringTypeIs>`, `<ParameterTypeIs>`, `<ParameterCountIs>`, `<ReturnTypeIs>`

4.2 Static Join Point Model

We classify the join point model into two categories: dynamic join point model and static join point model.

The join point model used in AspectJ and many other AspectJ-like languages is dynamic join point model. That is, some properties of the join point can only be determined at run-time. For example, consider the following code in AspectJ:

```
pointcut a() :  args(String)
```

This pointcut *a()* selects not only the method whose first formal parameter type is *String*, but also the method which will be called with *String* argument at runtime. That is to say, a method whose first formal parameter is *object* will be selected by the pointcut *a()* if it is called with an argument of *String*.

The dynamic join point model looks fine at first glance: pointcut *a()* does select all methods whose first argument is *String*. But it has significant effect towards system performance. Continuing the above example, in order to catch all the point when a method is called with *String* argument, the system has to type-check the arguments of all methods whose first formal parameter type is *object*. This overhead is too significant to be ignored, because a lot of methods in Java library use *object* as its first parameter, such as the methods in *Vector*. To make matter worse, using any type in *args()* expression will also cause this overhead, for all types are derived from *object*.

However, this problem are not that serious in practice, because *args* pointcuts normally used with the other pointcuts that statically discriminate join points so the overheads are in negligible range in most cases. But still the

overheads exist, and in the cyclic statements or in some critical functions, they cause problems. None the better, the programmers are unlikely to realize this overhead. When a programmer writes down the above statement, the things in his/her mind must be selecting some method whose first parameter is *String*. He/She seldom realizes the effect towards the system performance.

So we choose static join point model as our join point model in CCC. The static join point model means all properties of the join point can be statically determined at compile time. Consider writing the following code in CCC:

```
<PointCut Name="a">
   <And>
      <ParameterTypeIs>
         <ParameterIndex>1</ParameterIndex>
         <Type>
            <TypeNameMatches>System.String</TypeNameMatches>
         </Type>
      </ParameterTypeIs>
      <ParameterCountIs>1</ParameterCountIs>
   </And>
</PointCut>
```

This piece of code has almost the same meaning as the above AspectJ code except that: it only selects the method whose first formal parameter is *String*. Other methods, such as the methods whose first formal parameter is *object*, are not affected. So the programmers have more control over the system's behavior.

One virtue of static join point model is that one can easily simulate the dynamic join point model by writing a little code. So if a high-level language choose dynamic join point model, it can still be transformed to CCC without much work. For example, if we are going to simulate the following AspectJ code:

```
pointcut a() : args(String)
before() : a()
{
   //do something
}
```

We can write code in CCC like this:

```
<PointCut Name="a">
   <And>
      <ParameterTypeIs>
         <ParameterIndex>1</ParameterIndex>
         <Type><BaseOf>
         <TypeNameMatches>System.String</TypeNameMatches>
         </BaseOf></Type>
      </ParameterTypeIs>
      <ParameterCountIs>1</ParameterCountIs>
```

```
   </And>
</PointCut>
<Advice PointCut="a" Behaviour="Sample.Behaviour" Type="Before">
   <Parameters>
      <Parameter Name="StringArgument">
         <Type><Direct>System.Object</Direct></Type>
         <Binding Type="Argument" Index="0"/>
      </Parameter>
   </Parameters>
</Advice>
```

To select the join point where argument is of *String* type, we write a simple line in the behavior method:

```
public static void Behavior(object stringArgument)
{
  if (stringArgument is String)
  {
    //do something
  }
}
```

4.3 Context Binding Specified by Advice

In AspectJ and AspectJ-like languages, context binding is specified by pointcuts. For example:

```
pointcut a(int arg1) : call(* print(...)) && args(arg1);
```

This AspectJ statement binds *arg1* to the argument passed to *print(int)*. We believe this approach has the following drawbacks:

(i) The binding declaration is separated from the parameter declaration

When we declare a parameter in advice, our intention is to bind this parameter to some data in the execution context. Unfortunately, the binding declaration is in pointcut, so we have to navigate to the pointcut to check if the parameter has been correctly bound. Sometimes the context binding behavior defined in the pointcut is not what we wanted, then we have to declare new pointcut. Since every parameter declared in advice is aimed to be bound on something, putting the two declarations together can greatly improve the readability and modifiability.

(ii) Parameter binding must be explicitly checked

When parameter declaration and binding declaration is separated, there might be errors of parameter unbound or binding arguments not provided. Both situations should be explicitly checked, which adds burden to both user and compiler implementor.

(iii) Ambiguous expression could be formed

When context binding is specified by pointcuts, we can write code like

this:

```
pointcut a(int arg1) : call(* SomeMethod(..)) || args(arg1);
```

What will happen if *args(arg1)* evaluated as false and *call(* SomeMethod(..))* evaluated as true? What will be passed to arg1? This is an ambiguous expression. AspectJ considers such ambiguous expression as an error. But such semantic error is best to be avoided by a carefully designed the syntax.

Because of these drawbacks, the binding behavior in CCC is specified by advice. In CCC, every parameter declaration in advice contains a binding declaration specifying which data in the executing context this parameter should be bound to. For example, the following statements bind arg1 to the first argument passed to print method:

```
<Advice Behaviour="SomeBehaviorMethod"
Type="Before">
  <PointCut>
    <MethodNameMatches>print</MethodNameMatches>
  </PointCut>
  <Parameters>
    <Parameter Name="arg1">
      <Type><Direct>System.Int32</Direct></Type>
      <Binding Type="Argument" Index="0"/>
    </Parameter>
  </Parameters>
</Advice>
```

We believe this approach have solved all the problems above: binding declaration and parameter declaration are put together, and never will be parameter unbound error, binding arguments not provided error or ambiguous expression error occur.

Admittedly, this approach has its own drawbacks. One drawback is that it can not handle the case when a parameter to advice should be taken from different argument positions. For example, if advice runs with the first parameter to print method, or the second parameter to write method, AspectJ can specify the argument positions in a pointcut:

```
before(int x) : call(* *.print(int)) && args(x) ||
                call(* *.write(int,int)) && args(int,x) {...}
```

But users can not specify different argument positions in our approach. Fortunately, this can be simulated by writing multiple advices like the following:

```
<Advice Behaviour="BeforePrint" Type="Before">
  <PointCut>
    <MethodNameMatches>print</MethodNameMatches>
  </PointCut>
```

```
  <Parameters>
    <Parameter Name="x">
      <Type><Direct>System.Int32</Direct></Type>
      <Binding Type="Argument" Index="0"/>
    </Parameter>
  </Parameters>
</Advice>
<Advice Behaviour="BeforeWrite" Type="Before">
  <PointCut>
    <MethodNameMatches>write</MethodNameMatches>
  </PointCut>
  <Parameters>
    <Parameter Name="x">
      <Type><Direct>System.Int32</Direct></Type>
      <Binding Type="Argument" Index="1"/>
    </Parameter>
  </Parameters>
</Advice>

public static void BeforePrint(int x)
{
  AdviceBody(x);
}
public static void BeforeWrite(int x)
{
  AdviceBody(x);
}
public static void AdviceBody(int x){...}
```

Another drawback is when a pointcut is modified to match different join points, the binding specification in advice might be subject to modification. On the other hand, usually only the pointcut is affected in AspectJ. However, we don't consider this a serious problem. For one thing, this case is infrequent in development to our knowledge. For another, every approach has its own advantages and disadvantages. There are cases when AsepctJ code is subject to modification while CCC code is not.(e.g. When an advice needs to expose more data from the context, the user has to modify the pointcut in AspectJ. But in CCC, only the advice should be modified) So drawbacks like this are acceptable.

We also suggest the high-level languages constructed on CCC adopt this binding-in-advice approach. If a language adopts the binding-in-pointcut approach like AspectJ does, there is some extra work to do when transforming to CCC:

For each pointcut which contains binding declaration, do the following:

(i) If there is no *or* operator in the pointcut expression, or if there is *or* operator but the parameters to be bound are not contained in any one of its operands(e.g. `(call(* *.print(..)) || call(* *.write(..))) && this(x)`, x is not contained in any operands of the *or* operator), find what context data each parameter binds to. Then add binding declarations to all advices that use the pointcut.

(ii) If there is *or* operator in the poincut expression and its operands contain some parameter, do the following steps until there is no such *or* operator left in the expression:
   (a) Find the the *or* operator who has the lowest priority
   (b) Decompose the pointcut into two pointcuts at the point of the *or* operator. e.g. the pointcut
   ```
   pointcut a(int x) : call(* *.print(..)) &&
   (arg(String, x) || arg(x));
   ```
   is decomposed as
   ```
   pointcut a1(int x) : call(* *.print(..) && arg(String, x));
   pointcut a2(int x) : call(* *.print(..) && arg(x));
   ```
   (c) For each advice that uses the pointcut, duplicate the advice, and use the decomposed pointcuts respectively. For example, the following advice
   ```
   before(int x) : a(x){...}
   ```
   should be modified as:
   ```
   before(int x) : a1(x){...}
   before(int x) : a2(x){...}
   ```

5 Implementation

To make matters simple, we created a set of classes to represent the assembly structure in memory. This set of classes is called code graph. The system first parses the assembly into code graph, then performs weaving operations on code graph, and finally generates new assembly according to the woven code graph.

The System falls into five major parts: Assembly Parser, Weaver, Assembly Generator, Code Graph and Driver, their relationships are shown in figure 2.

Code graph is a set of classes to represent the type hierarchy in memory.

Assembly parser analyzes the structure of assembly and parses it into code graph.

Weaver reads the aspect specified in xml files and weaves them into code graph.

Assembly generator reads the code structure in code graph and writes them into the new assembly.

Driver controls the whole flow.

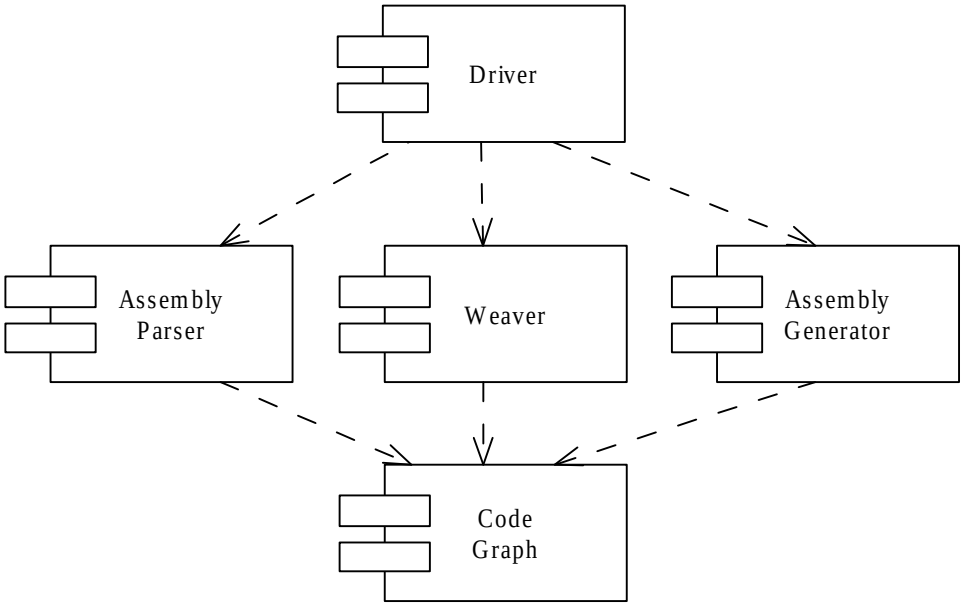

Fig. 2. System Architecture

6 Related Work

There are several tools trying to enable AOP on .Net platform, such as EOS[6], AspectC#[3], AOP.Net[7]. Some of these tools attempt to support all .Net languages, and these tools has greatly influenced the design of CCC.

Weave.NET[4] is a language-independent load-time weaver on .Net platform. Similar to CCC, Weave.NET uses XML to specify aspect binding. It weaves the behavior code, which stored in another assembly, into the target assembly according to the XML file at loading time. However, Weave.NET is designed to be used directly by the programmers, so its syntax greatly differs from the syntax of CCC.

Loom.NET[8] is another language-independent tool on .Net platform. It uses both xml file and an extension to C# language to specify crosscutting. But Loom.NET doesn't modify existing types but creates new ones. And the clients still have to modify their code to reference to the new assembly. In a large system where types always have a lot of clients, this approach would be less preferable than an approach that does not affects the clients.

There are also some Java AOP tools that describe pointcuts and advice specifications outside the Java language, and utilize method in Java byte code as advice bodies, which is similiar to what CCC does. JBoss-AOP[12] is the Java AOP architecture used for the JBOSS application server. AspectWerkz[1] is a dynamic, lightweight and high-performant AOP/AOSD framework for Java. They both use XML to specify aspect binding.

7 Conclusion and Future Work

The purpose of CCC is to provide an aspect-oriented intermediate language on .Net platform. We have strived to make CCC powerful, so that many language concepts are supported directly and the work of high-level language compiler is minimized, and flexible, so that the languages of different styles can all be supported by combining the language constructs in CCC differently.

We also summarized the characteristics which distinguish CCC from existing languages and stated the reason of such design:

(i) Primitive pointcuts.

(ii) Static join point model.

(iii) Specify context binding by advice.

We hope this tool will accelerate the development of aspect-oriented languages on .Net platform and will enable the .Net community to get more involved with aspect-oriented software development.

The current implementation of CCC compiler is just a prototype and its functionality is limited. e.g. It can not support assembly that contains unmanaged code. So the future work is mainly focused on creating a stable compiler. Also the current XML-based language is not so efficient as an intermediate language, so we plan to develop a binary equivalent which could be efficiently parsed and generated.

Aspect is often good reusable unit in its nature. But most languages only support aspect reuse on source code level. With the development of language in binary form, we also plan to support aspect reuse on binary level. One assembly can reuse the aspect defined in another assembly without referring to the source code of that assembly.

References

[1] Boner, J., and A. Vasseur, AspectWerkz Documentation, http://aspectwerkz.codehaus.org/introduction.html, 2004

[2] Fallside, D.C., *XML Schema Part 0: Primer*, http://www.w3.org/TR/2001/RECxmlschema-0-20010502/, 2001.

[3] Kim, H., "AspectC#: An AOSD implementation for C#, " M.Sc. thesis, Trinity College, Dublin, 2002.

[4] Lafferty, D., and V. Cahill, *Language Independent Aspect-Oriented Programming*, Proceedings of OOPSLA '03, 2003

[5] Lidin, S., "Inside Microsoft .NET IL Assembler. Microsoft Press, " Redmond, Washington, 2002.

[6] Rajan, H. and K. Sullivan, *Eos: Instance-Level Aspects for Integrated System Design*, Proceedings of the ESEC/FSE '03, 2003

[7] Schmied, F. AOP.NET Homepage, http://wwwse.fhs-hagenberg.ac.at/se/berufspraktika/2002/se99047/contents/english_home.html, 2003

[8] Schult, W., and A. Polze, *Aspect-Oriented Programming with C# and .NET*, Proceedings of International Symposium on Object-oriented Real-time distributed Computing '02, 2002, 241–248

[9] Schult, W., and A. Polze, *Dynamic Aspect-Weaving with .NET*, Workshop zur Beherrschung nicht-funktionaler Eigenschaften in Betriebssystemen und Verteilten Systemen, TU Berlin, Germany, 7-8 November 2002.

[10] Tarr, P. and H. Ossher, *Hyper/J User and Installation Manual*, http://www.research.ibm.com/hyperspace/, 2000

[11] The AspectJ Team, The AspectJ Programming Guide (V1.0.6), http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/aspectj-home/doc/progguide/index.html, 2004

[12] The JBoss Team, The JBoss AOP Homepage, http://www.jboss.org/index.html?module=html&op=userdisplay&id=developers/projects/jboss/aop, 2004