An Aspect-Oriented Approach for Solving the Inheritance Anomaly Problem

A. Shahen, Student Member, Institute of Statistical Studies and Research, Cairo University, Egypt.

Abstract— **Inheritance anomaly and crosscutting concerns are major problems in object-oriented programming. These problems have been discussed in several publications and there are still ongoing researches to find appropriate solutions. In this paper, we try to solve those two problems by presenting an aspect-oriented approach that handles the inheritance anomaly problem. In our proposed approach, both the functional components and aspects are presented in the Microsoft Intermediate Language (MSIL), which means that our approach is language independent.**

Index Terms— **Aspect-Oriented Programming, Crosscutting Concerns, Concurrent Programming, and Inheritance Anomaly.**

I. Introduction

Inheritance anomaly arises when additional methods of a subclass cause undesirable re-definitions of the methods in the superclass. Instead of being able to incrementally add code in a subclass the programmer may be required to re-define some inherited code, thus the benefits of inheritance are lost [1]. It has been pointed out that the combination of inheritance and synchronization constraints in concurrent object systems causes the inheritance anomaly problem [2].

Several approaches have been proposed for solving the inheritance anomaly problem in concurrent object-oriented languages (COOLs). The common idea of the proposed approaches is based on decoupling the synchronization code from the business code of class definition [3].

Crosscutting concerns are issues that could not be clearly localized or modularized into a single class often are implemented in multiple places throughout the program. Since the implementation of these concerns "crosscut" the system, they are called crosscutting concerns [4]. The most common concerns include: data representation, synchronization, location control, real-time constraints, failure recovery, and declarative specification [5,6].

Several approaches have been proposed for modularizing crosscutting concerns [7, 8, 9]. The approach that handles most of crosscutting concerns is known as aspect-oriented programming (AOP) [10]. The goal of AOP is to provide methods and techniques for decomposing problems into a number of *functional components* as well as a number of *aspects,* which crosscut *functional components* and then composing these components and aspects to obtain system implementations.

The current aspect-oriented languages that try to solve the problem of crosscutting concerns do not solve the inheritance anomaly problem. In this paper, we introduce an aspect-oriented approach that is suitable for modularizing crosscutting concerns and solves the inheritance anomaly problem.

The rest of this paper is organized as follows. In the next section, we show through an example where and how the inheritance anomaly problem occurs in COOLs. Section 3 describes the crosscutting concerns in more details. Section 4 shows that the current aspect-oriented languages that successfully handle the crosscutting concerns suffer from the inheritance anomaly problem. In section 5, we briefly describe our proposed approach that solves the inheritance anomaly problem. The effectiveness of our approach is demonstrated in section 6 through some examples. Finally, section 7 summarizes our conclusions and gives directions to future work.

II. The Inheritance Anomaly Problem

As briefly mentioned in the Introduction, the term *inheritance anomaly* refers to the problems arising from the coexistence of inheritance and concurrency in concurrent object oriented languages (COOLs). In this section, we use the classic bounded buffer example to show where and how the anomaly occurs. By using this example, we do not mean that the *inheritance anomaly* problem is a *reader-writer* problem. The *reader-writer* problem pertains to any situation where a data structure, database, or file system is read and modified by concurrent threads [11]. But, the inheritance anomaly arises when additional methods of a subclass cause undesirable re-definitions of the methods in the superclass.

Consider the following pseudo-code for a bounded buffer:

```
class Buffer
{    void put(Object el) { ... }
     Object get() { ... }
}
```

Manuscript received July 5, 2007.
A. Shahen, Computer and Information Science Dep., Institute of Statistical Studies and Research, Cairo University. Egypt. (e-mail: ashraf_shahen@yahoo.com).

The methods *get* and *put* respectively to remove and insert an element. In a concurrent setting we need to refine the code above with suitable synchronization code, so as to make sure that no *get* is executed on an empty buffer and, that no *put* is executed on a full buffer. The synchronization code is simply a Boolean expression, known as a *guard*, which must be true for the method to be executable. If the guard evaluates to false the calling thread must '*wait*' for the guard to become true.

The occurrences of the inheritance anomaly problem have been classified into three broad categories [1, 2, 3]. The inheritance anomaly problem in each of these categories will be explained through defining subclasses for the bounded buffer class.

Category 1: History-sensitiveness of acceptable states

Anomaly occurs when the synchronization code defined in a subclass depend on the history of the object. As an example, assume that we want to refine the bounded buffer class by defining a subclass with the method *gget* that works like the method *get* but cannot be executed immediately after the method *get*. Clearly, this can only be achieved by modifying code of the method *get* in the superclass to keep track of its invocations.

Category 2: Partitioning of states

Anomaly occurs when the addition of a subclass forces a refinement of the object's state partition. As an example, assume that we want to refine the bounded buffer class by defining a subclass with the method *get2* that retrieves two elements at once. Before adding such a subclass the object's state can be partitioned into three sets: *empty*, *partial,* and *full*. But adding the subclass that contain the method *get2* forces the state transitions to be re-described to include the state where the buffer contains exactly one element.

Category 3:Modification of acceptable states

A third kind of anomaly occurs in the multiple inheritance situations where the acceptance states of the original class' methods are influenced by adding subclasses. As an example, assume that we want to define a lockable buffer subclass based on two classes: the buffer class and another class, called *lock,* with lock capabilities (*lock* and *unlock*). In this case, we need to modify the *get* and *put* methods in the buffer class to keep into account the state of the *lock* component of the object.

III. Crosscutting concerns

As mentioned in the Introduction, crosscutting concerns are issues that appear in multiple places in the program and could not be easily modularized to a separate class. For example, consider a figure editor that is used to build figures/images on the screen from different objects (points, lines, circles, etc.). Every time that a change is made to one of the figures on the screen (like moving it from one point to the next), something must occur (like a method call) that tells the screen to repaint. The issue of repainting the screen would be considered a crosscutting concern since it is a concern with several different objects.

In typical object-oriented programming a crosscutting concern (such as screen repainting) is accomplished by calling the appropriate screen method whenever one of the several methods of the objects on the screen is called. One of the issues when working in this type of an environment involves the ability to maintain the code. Whenever the screen repaint method changes either its name or its parameters, the change would need to be propagated to the calling methods. Therefore, all of the objects that call on those methods must be changed as well. Those calls may not be easy to locate [4].

The idea behind aspect-oriented programming is to take those types of concerns that are scattered throughout the program and bring them together into a single structure called an aspect. By allowing a crosscutting concern to be handled within a single aspect, the implementation of the concern can be localized.

IV. Current Aspect-Oriented Languages

Several aspect-oriented languages (AOLs) have been proposed for solving modularity problems. Some of these languages solve the problems of *crosscutting concerns* very well, but they still suffer from the inheritance anomaly problem as a result of the difficulty of inheriting the aspect code in the presence of inheritance. As an example, suppose we have developed an aspect *TraceBefore* to trace the start of execution of the *get* and *put* methods in the *buffer* class as given in Fig. 1 and 2.

Now consider the subclass *SpecialBuffer*, as defined in Fig. 3, which redefines the method *put* of the *Buffer* class; and assume that we do not need to trace the start of execution of the method *put* in the subclass *SpecialBuffer*. In principle, it should be possible to "inherit" the *TraceBefore* aspect just modifying the code associated to the method *put*. Unfortunately, this is not possible because some languages don't support the "sub-aspect" concept. Thus, it is necessary to rewrite the aspect code as given in Fig. 4.

```
public class Buffer
{ protected Object[] buf;
  protected int MAX;
  protected int current = 0;
   Buffer(int max)
  {
    MAX = max;
    buf = new Object[MAX];
  }
  public synchronized Object get()  throws Exception {//….}
  public synchronized void put(Object v) throws Exception
  { //……….   }
}
```

Fig. 1: *Buffer* Class in Java

```
public aspect TraceBefore
{
    private void Buffer.print (String methodName, int currentValue)
    {
       System.out.println("Tracing method " +methodName+" before");
       System.out.println("current = " + currentValue);
    }

     before(Buffer x):(call(void Buffer.put(object)) ||
                        call(object  Buffer.get()))
                       && target(x)
    {
       x.print( thisJoinPoint.getSignature().getName(), x.current);
    }
}
```

Fig. 2: *TraceBefore* Aspect in AspectJ

```
class SpecialBuffer extends Buffer
{
   public SpecialBuffer(int max)
   {   super(max);   }

   public synchronized void put(String v) throws Exception
   {  //………  }
}
```

Fig. 3: *SpecialBuffer* class in Java

```
public aspect TraceBefore
{
    private void Buffer.print(String methodName,int currentValue)
    {
        System.out.println("Tracing    method    "+    methodName+"
        before");
       System.out.println("current= "+currentValue);
    }

     before(Buffer x):
       ((call(void Buffer.put(String))  ||
       call(String Buffer.get())) &&
       (!call(void SpecialBuffer.put(String))))
       && target(x)
    {
       x.print( thisJoinPoint.getSignature().getName(), x.current);
    }
}
```

Fig. 4: *TraceBefore* Aspect in AspectJ

V. Our Approach

Based on the discussion in the previous section, we propose an aspect-oriented approach that solves the problem of inheriting the aspect code. Our approach is composed of two models: *aspect model* and *weaving model*, together with a .NET tool: *.NET Weaver,* which implements the weaving model. The aspect model provides a technique for separating the synchronization constraints from functional requirement in development time. The weaving model composes the synchronization constraints with the functional requirement in runtime so that these constraints can be enforced on functional entities in run-time system. In the next subsections we describe these two models and the .NET Weaver.

A. Aspect Model

The aspect model has three elements: a join point model, a means of identifying join points, and a means of affecting implementation at join points. These elements correspond to the characteristics that allow an aspect-oriented mechanism to crosscut an application [12].

1) Join Point Model

The join point model provides the common frame of reference that makes it possible to define the structure of crosscutting concerns. The join point model includes several kinds of join points, which are certain well-defined points in the execution flow of the program. Join points can be categorized as follows:

Method call join point: A method call join point encompasses the actions of an object receiving a method call. It includes all the actions that comprise a method call, starting after all arguments are evaluated up to and including normal or unexpected return.

Create join point: When an object is built and a constructor is called.

Field reference and assignment: When a field is retrieved or assigned to.

2) Identifying Join Points

Pointcut designators (or simply pointcuts) identify particular join points by filtering out a subset of all the join points in the program flow. In our model, we use four pointcut designators: *call(Signature)*, *create(Signature)*, *get(Signature)*, and *set(Signature)*.

3) Modifying Join Point Behavior

Advice declarations are used to define additional code that runs at join points. In our model, we use the following advices:

Before advice: runs when a join point is reached and before the computation proceeds, i.e. it runs when computation reaches the method call and before the actual method starts running.

After advice: runs after the computation 'under the join point' finishes, i.e. after the method body has run, and just before control is returned to the caller, regardless of whether it returns normally or throws an exception.

Around advice: runs instead of the reached join point, and has explicit control over whether the computation under the join point is allowed to run at all.

After returning advice: runs just after each join point picked out by the pointcut, but only if it returns normally.

After Throwing advice: runs just after each join point picked out by the pointcut, but only if it throws an exception.

VI. WEAVING MODEL

The weaving model is used for describing the kinds of weaving techniques. There are two techniques in current aspect-oriented languages and tools, in which classes and aspects can be woven: static or dynamic [13]. Static weaving means to modify the source code of a class by inserting aspect-specific statements at join points. The result is highly optimized woven code, whose execution speed is comparable to that of code written without using aspects. However, static weaving makes it difficult to later identify aspect-specific statements in woven code.

Dynamic weaving means to weave aspects with classes during runtime. Dynamic weaving allows a dynamic adaptation of an application at any time. However, dynamic weaving increases the time and the amount of memory needed for the program's execution.

To combine the advantages of both techniques, we introduced the Just in time (JIT) weaving technique. During the program's execution, JIT weaver uses inheritance to add aspect-specific code to classes instead of modifying the source code of classes while weaving aspects. Generated subclasses are saved to a new component for the next use. Aspects are not weaved with classes until the first time the classes are used. This is done on a per-class, so the delay for JIT weaving is only as long as needed for the classes you want to use. The time spent in the JIT weaving is so minor that it is almost never noticeable, and once a class has been weaved, you never incur the cost for that class again. JIT Weaver can dynamically add, adapt, or remove aspects at any time. For example, if an aspect has been modified, the Weaver updates its woven methods. If an aspect is to be removed completely, the Weaver removes the respective subclass.

JIT Weaver relies on XML to perform binary-level weaving, meaning that the weaving specification is not written in terms of, or using extensions to, a particular programming language. By separating the weaving specification from the aspect code, we can modify each of them without affecting the other.

JIT Weaver uses metadata and reflection mechanisms provided by .NET Framework to examine the compiled assemblies and to generate weaved subclasses. Reflection information is mandatory for every .NET assembly. It does not care weather an assembly is written in java or in C#. This means that JIT Weaver works language independently.

VII. THE .NET WEAVER TOOL

We have implemented our approach as a .NET component called ".NET Weaver". .NET weaver weaves already compiled .NET aspect components with already compiled .NET functional components. We can change each of them without affecting the others. When we write .NET functional component, we do not need to know which aspects will be applied to it. In addition, we can write general aspects separately from any classes to which they may apply. This feature increases software reusability, flexibility and extendibility. The only restriction is that target class methods, which should become interwoven, have either to be virtual or to be defined via an interface. Each aspect is defined as a NETWeaver.Aspect subclass. Programmer uses .xml file to define assembly information and weaving specifications.

.NET weaver provides a class named weaver to weave a given target class. This class does the same as the new statement; it creates a new object of a given class (the target class). Furthermore, this class weaves the target class with an aspect-object. .NET weaver is depicted in Fig. 5.

Once the running application reaches one of the new statements, Instance Creator is invoked to create a new object of a given class. Instance Creator sends the class type to the Weaving Descriptor. Immediately after sending, Weaving Descriptor inspects the class type to find the set of join-points and sends them back to Instance Creator. Instance Creator passes the class type and the set of join-points to Weaver. During this step, Weaver creates a new subclass instance from the target class and creates aspect instances for the join-points. Weaver weaves all the created aspect instances to the subclass instance and sends them back to the Instance Creator. The weaver weaves different aspects with the class by determining and adapting all parts where aspect specific elements are needed. Finally, Instance Creator sends the new object to the running application.

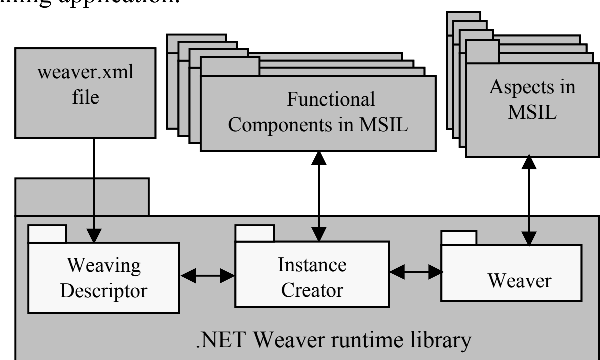

Fig.5: .NET Weaver

We used the Unified Software Development Process together with a popular CASE tool: Rational Rose, in developing .NET Weaver. The Unified Software Development Process is the end product of three decades of development and practical use. Its development has been guided by three leading figures in software development: Jacobson, Booch, and Rumbaugh [14].

VIII. OUR APPROACH EFFECTIVENESS

In this section, we demonstrate the effectiveness of our approach through some examples. The crosscutting concerns example in section 3 and the inheritance anomaly examples in

ISBN:978-988-98671-6-4

sections 2 and 4 are now solved using our proposed system.

A. Figure Editor Example

Fig. 6 shows the class diagram of the Figure Editor. A Figure consists of a number of FigureElements, which can be either Points or Lines. The Figure class is also a factory for figure elements. There is a single Display on which figure elements are drawn. Using .NET weaver we can easily modularize crosscutting concerns (such as screen repainting) to a separate class as shown in Fig. 7. Fig. 8 shows the weaving specification declared in XML file.

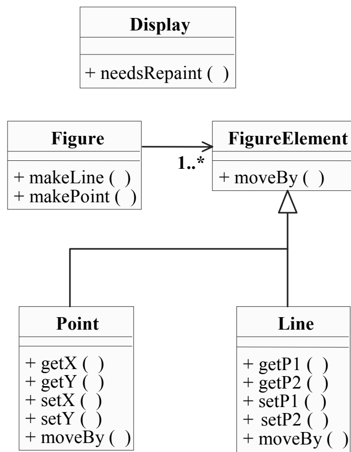

Fig. 6:Figure Editor Class Diagram

```
public class DisplayUpdating: NETWeaver.Aspect
{
        public DisplayUpdating() { … }
        public virtual void move(object[] args)
        {
                TestWeaver.Display.needsRepaint();
                return;
        }
    }
```

Fig. 7: DisplayUpdating aspect

We define a pointcut named move that designates any method call that moves figure elements. After advice on move poincut informs the display it needs to be refreshed whenever an object moves.

```
<Pointcuts>
   <Pointcut name="move">
      Call(*.* *.Point.setX(..))  ||
      Call(*.* *.Point.setY(..))  ||
      Call(*.* *.Line.setP1(..))  ||
      Call(*.* *.Line.setP2(..))  ||
      Call(*.* *.Point.moveBy(..))||
      Call(*.* *.Line.moveBy(..))
   </Pointcut>
</Pointcuts>
<Aspects>
   <Aspect name="DisplayUpdating" inherited="true">
      <Method name="move">
            <After>move</After>
      </Method>
      </Aspect >
</Aspects>
```

Fig. 8: Figure Editor Weaving Specification

B. Bounded Buffer Example

In this example, we use the .NET Weaver for solving the inheritance anomaly examples in sections 2 and 4.

i. History-sensitiveness of acceptable states

As mentioned in section 2, anomaly occurs when the synchronization code defined in a subclass depend on the history of the object. We have implemented the Buffer class in the C# without any synchronization code as shown in Fig. 9. In Fig. 10, the synchronization code of the Buffer class is implemented in the BufferSyn aspect.

```
public class Buffer
{
   protected object[] buf;
   public int max;
   public int current = 0;

   public Buffer(int max)
   {
   this.max = max;
   buf = new object [max];
   }

   [MethodImpl(MethodImplOptions.Synchronized)]
   public virtual void put(object v)
   {
   buf[current] = v;
   current++;
   }

   [MethodImpl(MethodImplOptions.Synchronized)]
   public virtual object get()
   {
   current--;
   object ret = buf[current];
   return ret;
   }
}
```

Fig. 9: The Buffer class in C#

```
public class BufferSyn: NETWeaver.Aspect
{   public BufferSyn():base()   { }
   [MethodImpl(MethodImplOptions.Synchronized)]
   public virtual void putsyn(object[] args)
   {   object instance = target.Instance;
       Monitor.Enter(this);
       while (((Buffer)(instance)).current>=((Buffer)(instance)).max)
       { Monitor.Wait(this); }
       target.Proceed(args);
       Monitor.PulseAll(this);
       Monitor.Exit(this);
   }
   [MethodImpl(MethodImplOptions.Synchronized)]
   public virtual object getsyn(object[] args)
   {   object instance = target.Instance;
       Monitor.Enter(this);
       while (((Buffer)(instance)).current<=0) { Monitor.Wait(this); }
       object ret=target.Proceed(args);
       Monitor.PulseAll(this);
       Monitor.Exit(this);
       return ret;
   }
}
```

Fig. 10: The BufferSyn aspect

The HistoryBuffer class extends the Buffer class with the method gget that works like the method get but that cannot be executed immediately after the method get. As shown in Fig. 11, we do not need to rewrite the entire Buffer class code. Only the gget method functional code is defined. The re-defined synchronization conditions are implemented in the HistoryBufferSyn aspect. HistoryBufferSyn aspect is shown in Fig. 12.

```
public class HistoryBuffer : Buffer
{
    public HistoryBuffer(int max) : base(max) {}

     [MethodImpl(MethodImplOptions.Synchronized)]
    public object gget()
    {
        return base.get();
    }
}
```

Fig. 11: HistoryBuffer class

```
public class HistoryBufferSyn: BufferSyn
    {
        public bool afterGet = false;
        public HistoryBufferSyn():base() { }

         [MethodImpl(MethodImplOptions.Synchronized)]
        public override void putsyn(object[] args)
        {
            base.putsyn( args);
            afterGet = false;
        }

         [MethodImpl(MethodImplOptions.Synchronized)]
        public  override object getsyn(object[] args)
        {
            object ret=base.getsyn( args);
            afterGet = true;
            return ret;
        }

         [MethodImpl(MethodImplOptions.Synchronized)]
        public virtual object ggetSyn(object[] args)
        {
            object instance = target.Instance;
            Monitor.Enter(this);
            while ((((HistoryBuffer)(instance)).current<=0)||(afterGet) )
                { Monitor.Wait(this); }
            object ret=base.getsyn( args);
            afterGet = false;
            Monitor.PulseAll(this);
            Monitor.Exit(this);
            return ret;
        }
    }
```

Fig. 12: HistoryBufferSyn aspect

ii. Partitioning of states

Synchronization conditions can be implemented by describing the enabling of methods according to a partition of the object's states. Fig. 13 shows the BufferSynWithStates aspect that implements the Buffer class synchronization code with states: empty, partial, and full. In Fig. 14, StatePartitioningBuffer class extends the Buffer class with the method get2 that retrieves two elements at once. As shown in Fig. 14, we do not need to re-define the methods get and put. Only the get2 method functional code is defined. The re-defined synchronization conditions are implemented in the StatePartitioningBufferSyn aspect. The StatePartitioning-BufferSyn aspect is shown in Fig. 15.

```
public class BufferSynWithStates: NETWeaver.Aspect
{
    public bool full;
    public bool empty;
    public bool partial;

    public BufferSynWithStates():base()
    {
        full=false;
        partial=false;
        empty=true;
    }

    [MethodImpl(MethodImplOptions.Synchronized)]
    public virtual void putsyn(object[] args)
    {
        object instance = target.Instance;
        Monitor.Enter(this);
        while (full) { Monitor.Wait(this); }
        target.Proceed(args);
        if(((Buffer)(instance)).current>=((Buffer)(instance)).max)
        {
            partial=false;
            full=true;
        }
        else
        {
            partial=true;
            full=false;
        }
        empty=false;
        Monitor.PulseAll(this);
        Monitor.Exit(this);
    }

    [MethodImpl(MethodImplOptions.Synchronized)]
    public virtual int getsyn(object[] args)
    {
        object instance = target.Instance;
        Monitor.Enter(this);
        while (empty) { Monitor.Wait(this); }
        object ret=(object)(target.Proceed(args));
        if(((Buffer)(instance)).current<=0)
        {
            partial=false;
            empty=true;
        }
        else
        {
            partial=true;
            empty=false;
        }
        full=false;
        Monitor.PulseAll(this);
        Monitor.Exit(this);
        return ret;
    }
}
```

Fig. 13: BufferSynWithStates aspect

```
public class StatePartitioningBuffer : Buffer
{
    public bool afterGet = false;
    public StatePartitioningBuffer(int max) : base(max) {}

     [MethodImpl(MethodImplOptions.Synchronized)]
    public virtual object[] get2()
    {
        object[] ReturnValues=new object[2];
        ReturnValues[0]=base.get();
        ReturnValues[1]=base.get();
        return ReturnValues;
    }
}
```

Fig. 14: StatePartitioningBuffer class

```
public class StatePartitioningBufferSyn:BufferSynWithStates
{     public bool one;

    public StatePartitioningBufferSyn(): base()      {     one=false;     }

     [MethodImpl(MethodImplOptions.Synchronized)]
    public override void putsyn(object[] args)
    {
        base.putsyn(args);
        object instance = target.Instance;
        if(((Buffer)(instance)).current==1)
            one=true;
        else
            one=false;
    }

    [MethodImpl(MethodImplOptions.Synchronized)]
    public override object getsyn(object[] args)
    {
        object instance = target.Instance;
        object ret=(object)(base.getsyn(args));
        if(((Buffer)(instance)).current==1)
            one=true;
        else
            one=false;
        return ret;
    }

    [MethodImpl(MethodImplOptions.Synchronized)]
    public virtual object[] get2syn(object[] args)
    {
        object instance = target.Instance;
        Monitor.Enter(this);
        while (empty ||one) { Monitor.Wait(this); }
        object[] ret=(object[])(target.Proceed(args));
        if(((Buffer)(instance)).current==1)
            one=true;
        else
            one=false;
        if(((Buffer)(instance)).current<=0)
            empty=true;
        else
            empty=false;
        full=false;
        Monitor.PulseAll(this);
        Monitor.Exit(this);
        return ret;
    }
}
```

Fig. 15: StatePartitioningBufferSyn aspect

iii. Inheriting the aspect code

.NET Weaver solves the difficulty of inheriting the aspect code by:

- decoupling the aspect code from the weaving specification.
- supporting sub-aspects.
- Weaving aspects with classes without modifying the source code of classes and aspects.

Assume, for example, we have developed the aspect TraceBefore and the weaving specification using the .NET Weaver as given in Fig. 16 and Fig.17. As shown in Fig 18, we can modify the weaving specification associated to the method put without rewriting and recompiling the aspect code.

```
public class TraceBefore: NETWeaver.Aspect
{
  protected void print(string methodName,int currentValue)
  {
    Console.Out.WriteLine("Tracing method "+  methodName+" before");
    Console.Out.WriteLine("current= "+currentValue);
  }
  public virtual object AspectMethod(object[]argu)
  {
    print( target.MethodName.ToString(), ((Buffer)(target.Instance)).current);
     return target.Proceed(argu);
   }
}
```

Fig. 16: TraceBefore Aspect in .NET Weaver

```
<Pointcuts>
   <Pointcut name="pointcut1">
        Call(* TracableBuffer.Buffer.put(..))
    </Pointcut>

    <Pointcut name="pointcut2">
        Call(*.* TracableBuffer.Buffer.get())
    </Pointcut>
</Pointcuts>

<Aspects>
    <Aspect name="TracableBuffer.TraceBefore" inherited="true">
        <Method name="AspectMethod">
            <Around>(pointcut1||pointcut2)</Around >
        </Method>
    </Aspect >
</Aspects>
```

Fig. 17: The weaving specification in .NET Weaver

```
<Pointcuts>
   <Pointcut name="pointcut1">
        Call(* TracableBuffer.Buffer.put(..))
   </Pointcut>
   <Pointcut name="pointcut2">
       Call(*.* TracableBuffer.Buffer.get())
    </Pointcut>
   <Pointcut name="pointcut3">
       Call(* TracableBuffer.SpecialBuffer.put(..))
    </Pointcut>
</Pointcuts>

<Aspects>
   <Aspect name="TracableBuffer.TraceBefore" inherited="true">
       <Method name="AspectMethod">
           <Around>(pointcut1||pointcut2)&amp;&amp;(!pointcut3)</Around >
       </Method>
   </Aspect >
</Aspects>
```

Fig. 18: Modifying the weaving specification associated to the method put

ISBN:978-988-98671-6-4

Now consider that we want to refine the aspect TraceBefore by defining the sub-aspect TraceBeforeSub, which redefines the method AspectMethod and trace the start of execution of the method put in the subclass SpecialBuffer. As shown in Fig. 19 and Fig. 20, we do not need to rewrite the aspect code or the weaving specification. The modification is only as long as needed.

```
public class TraceBeforeSub: TraceBefore
{
   public  override object AspectMethod(object[]argu)
   {
      print( target.DeclaringType.ToString()+"."+target.MethodName.
      ToString(), ((Buffer)(target.Instance)).current);
      return target.Proceed(argu);
   }
}
```

Fig. 19: The TraceBeforeSub Aspect in .NET Weaver

```
<Aspect name="TracableBuffer.TraceBeforeSub" inherited="true">
   <Method name="AspectMethod">
      <Around>pointcut3</Around>
   </Method>
</Aspect >
```

Fig. 20: The weaving specification in XML file

C. Performance

.NET Weaver addresses program complexity by reducing the number of lines of code in an application. This is confirmed when we compare the number of lines of code that .NET Weaver requires to implement the Figure editor example in Fig. 6 to the number of lines of code required by other aspect-oriented tools to implement the same functionality. This comparison is shown in Table 1, which provides the source code line count for implementing the Figure editor example in regular C#, .NET Weaver, EOS, LOOM.NET, and AspectC#.

Table 1: Comparison of lines of source code in the Figure editor example

	C#	.NET Weaver	EOS	LOOM	AspectC#
Number of lines	154	140	173	190	255

As shown in Table 1 .NET Weaver reduces 9% of lines of source code. This line count is on the code size of the C# implementation. Unfortunately, our comparison is somewhat native, as the .NET Weaver and AspectC# source code size does not take into account the XML weaving specification.

Table 2 shows the execution times of the Figure editor example implemented in C#, .NET Weaver, EOS, LOOM.NET, and AspectC#. Although .NET Weaver increases the program execution time compared with the regular C#, its performance is better than the others.

Table 2: Comparison execution time of the Figure editor example in different languages (Time in Microseconds)

C#	.NET Weaver	EOS	LOOM	AspectC#
866,637	1,034,150	5,372,101	5,630,869	2,008,097

.NET Weaver uses JIT weaver at run-time which means that it adds more run-time penalty to the program execution. The weaving process is done once on executing the program for the first time only. In other words, the first execution time of the program is increased by the weaving time. After the first execution time of the program is done, the weaving time will never be added again to the execution time. Table 3 shows the weaving times for weaving aspects with components in the Figure editor example. The execution and weaving times in Table 2 and 3 are the averages of 10 trials.

Table 3: Weaving times for the Figure editor in different aspect-oriented languages (Time in Microseconds)

.NET Weaver	EOS	LOOM	AspectC#
636,865	585,636	244,482	1,068,629

In dynamic weaving languages, like LOOM, the weaving time is added to each program execution time. In EOS and AspectC#, the weaving is done statically at compile-time. Therefore, there is no run-time overhead. The total execution times are shown in Table 4.

Table 4: Comparison of total execution times of the Figure editor example in different languages (Time in Microseconds)

	C#	.NET Weaver	EOS	LOOM	AspectC#
The first execution time	866,637	1,671,015	5,372,101	5,875,351	2,008,097
The second execution time	866,637	1,034,150	5,372,101	5,875,351	2,008,097
N	866,637	1,034,150	5,372,101	5,875,351	2,008,097

Fig. 21 compares the execution times for the same programs with the same numbers of source code lines implemented in C# and .NET Weaver. As shown in Fig. 21, .NET Weaver increases the program execution time by an amount that does not depend on the program size. Therefore, in large applications the execution time overhead is not noticeable.

IX. Conclusions and Future Work

This paper has presented the .NET Weaver, an aspect-oriented approach that is designed and implemented on the .NET platform. This implementation takes advantage of the language independence features present in the .NET platform.

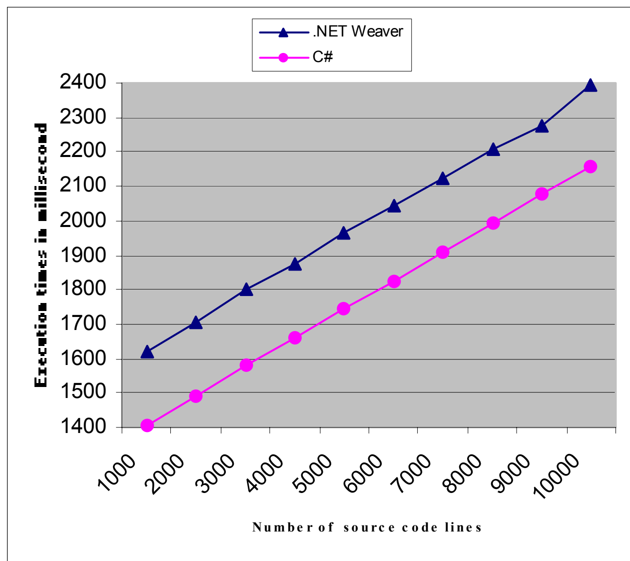

Fig. 21: The execution time overhead does not depend on the program size

The weaving specification is written in XML to avoid the need to recompile the components to store the crosscutting semantics. .NET Weaver does not suffer from inheritance anomaly problem (as was discussed in section 6), whereas most of the current aspect-oriented languages suffer from this problem.

A number of issues were not covered in this paper; these are the subjects of our future research. One of these issues is the weaving of aspects with aspects. Weaving aspects with aspects increases the reusability of aspects and reduces the number of lines of code.

In addition, we intended to evaluate our approach with different types of anomaly, for example, real-time constraints inheritance anomaly. Recently, there have been some attempts in defining real-time object-oriented languages. Similar to concurrent object-oriented languages, real-time object-oriented languages may suffer from the real-time constraints inheritance anomaly. In contrast to concurrent object-oriented languages, there has been almost no study on the origins of the real-time constraint inheritance anomaly problem. Needless to say, the combined analysis of concurrent and real-time constraint inheritance anomalies has not been addressed, although most real-time systems are concurrent. Finally, we evaluated our approach with C++, C#, VB.NET, and J# and we are keen to evaluate our approach with other .NET languages.

X. References

[1] L. Crnogorac, A. Rao, and K. Ramamohanarao, “Inheritance anomaly - a formal treatment”, *FMOODS'97*, England, July 1997, pages 319--334. Available: http://citeseer.ist.psu.edu/article/crnogorac97inheritance.html

[2] S., Matsuoka, and A., Yonezawa, “Analysis of inheritance anomaly in object-oriented concurrent programming language”, In Research Directions in Concurrent Object-Oriented Programming, pages 107–150, 1993. Available: http://citeseer.ist.psu.edu/cache/papers/cs/3223/ftpzSzzSzcamille.is.s.u-tokyo.ac.jpzSzpubzSzpaperszSzbook-inheritance-anomaly-a4.pdf/matsuoka93analysis.pdf

[3] G., Milicia, and V., Sassone, “The inheritance anomaly: ten years after”, *SAC'04*, March 14–17, 2004, Nicosia, Cyprus. *ACM* 1-58113-812-1/03/04.Available: http://www.cogs.susx.ac.uk/users/vs/research/paps/anomalySurvey .pdf

[4] R. Kaiser, “Aspect Oriented Programming”, *Technical Report Number CTU-CS-2002-006*, Colorado Technical University, 2002. Available: http://iis-web.coloradotech.edu/Computer_Science/Tech_Reports/KaiserTR.pdf

[5] W. Hürsch and C. Lopes, "Separation of concerns", *College of Computer Science*, Northeastern University, Boston, MA 02115, USA, February 24, 1995. Available: ftp://www.ccs.neu.edu/pub/people/crista/papers/ separation.ps

[6] J. Aldrich, “Evaluating Module Systems for Crosscutting Concerns”, *Ph.D. General Examination Report*, Department of Computer Science and Engineering, University of Washington, September 28, 2000. Available: http://www.cs.washington.edu/homes/jonal/papers/generals.pdf

[7] F. Holljen, “Compilation and Type-Safety in the Compose* .NET environment”, *M.Sc. thesis*, University of Twente, May 6, 2004.

[8] G. Tandon and S. Ghosh, "Using Subject-Oriented Modeling to Develop Jini Applications", *In Proceedings of the 8th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2004)*, pp. 111-122, Monterey, California,2004. Available: September 20-24, 2004. http://www.cs.colostate.edu/~ghosh/papers/edoc2004.pdf

[9] H. Ossher and P. Tarr. "Multi-Dimensional Separation of Concerns and The Hyperspace Approach." *In Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development.* Kluwer, 2000. Available: http://www.research.ibm.com/hyperspace/Papers/sac2000.pdf

[10] E. Kendall, “Aspect-oriented Programming in AspectJ”, *Evolve 2000*, Sydney, March, 2000. Available: http://www.pscit.monash.edu.au /~kendall/evolve2000.pdf

[11] A. Downey, “The Little Book of Semaphores”, *The Free Software Foundation*, Second Edition, 2007, page 71. Available: http://greenteapress.com/semaphores/downey05semaphores.pdf

[12] D. Lafferty, “Aspect-Based Properties”, *Ph.D. thesis*, University of Dublin, Trinity College, October 2004. Available: http://www.precise-concise.com/Thesis_Final.pdf

[13] S. Almajali and T. Elrad, “A Dynamic Aspect Oriented C++ using MOP with Minimal Hook Weaving Approach”, *Aspect-Oriented Software Development conference*, Lancaster, England, March, 2004, pages 1-8.

[14] I. Jacobson, G. Booch, and J. Rumbaugh, “The Unified Software Development Process”, *Addison-Wesley*, 1999.