Freely Annotating C#

Walter Cazzola
DICo - Department of Informatics and Communication,
Università degli Studi di Milano, Italy.
cazzola@dico.unimi.it

Antonio Cisternino
Department of Computer Science,
Università degli Studi di Pisa, Italy.
cisterni@di.unipi.it

Diego Colombo
IMT - Istituzione Mercati Tecnologie,
Alti Studi Lucca, Italy.
colombo@imtlucca.it

Reflective programming is becoming popular due to the increasing set of dynamic services provided by execution environments like JVM and CLR. With custom attributes Microsoft introduced an extensible model of reflection for CLR: they can be used as additional decorations on element declarations. The same notion has been introduced in Java 1.5. The annotation model, both in Java and in C#, limits annotations to classes and class members. In this paper we describe [a]C#[a], an extension of the C# programming language, that allows programmers to annotate statements and code blocks and retrieve these annotations at run-time. We show how this extension can be reduced to the existing model. A set of operations on annotated code blocks to retrieve annotations and manipulate bytecode is introduced. We also discuss how to use [a]C# to annotate programs giving hints on how to parallelize a sequential method and how it can be implemented by means of the abstractions provided by the run-time of the language. Finally, we show how our model for custom attributes has been realized.
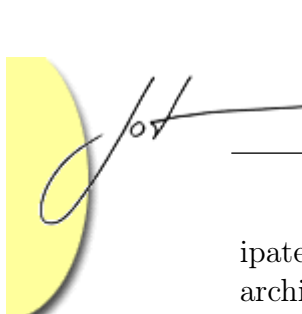**Keywords:** Reflection, Code Annotation,.NET,C#.

---

[a]The name [a]C# should be pronounced as *annotated C sharp*.

1   INTRODUCTION.

Reflection and dynamic loading are becoming essential elements of modern programs. Their usefulness is testified, for example, by the JDBC architecture that shows how to implement a driver-based architecture exploiting the Java dynamic loading.

Although reflection can be used to inspect the structure of types, to access fields and even to invoke methods dynamically, the concept of tagging has been antic-

---

ipated as an interesting application. Consider for instance the Java serialization architecture: the programmer can declare the instances of a serializable class simply by implementing the `Serializable` interface, which in fact is an empty interface. Thus, two types that differ only for the implementation of the `Serializable` interface are indistinguishable from the execution standpoint. Besides, the serialization of the instances of non-serializable types will not be allowed by the serialization support. Java serialization taught us that the meta-data stored with the code can be used for other purposes than mere execution. Other programs may rely on the reflective abilities of inspecting the compiled types and act differently depending on what they have found.

Although widely used by Java programs, the idea of providing explicit meta-data support for annotation has been introduced first by Microsoft in the common language run-time (CLR). The virtual execution environment is part of the standard Common Language Infrastructure (CLI) [14] and a crucial element of the .NET initiative. The CLR executes code expressed in an intermediate language (IL) like the JVM bytecode, though the code is stored and distributed in the form of an assembly. A single assembly contains the definition of one or more types and may refer to other assemblies. Together with the meta-data, required by the CLR for loading and managing the types contained within an assembly, it is possible to store arbitrary information in the form of *custom attributes*. A *custom attribute* is an instance of a class that inherits from the `Attribute` class. A custom attribute is created by invoking one of its constructors, though all the values used to create it must be computable at compile time. Custom attributes are serialized into an assembly at compile-time and ignored by the execution system. Nevertheless, the reflection API provides a means to retrieve these attributes at run-time. For instance, let `m` be an instance of `MethodInfo` (a reflective descriptor of a method). In C# we can retrieve the custom attributes associated with the method as follows:

```
Attribute[] attr = m.GetCustomAttributes();
```

In the CLR model attributes can be used to decorate essentially all the objects accessible through reflection: assemblies, types (delegates, value types, and classes), fields, properties, and methods.

Problem Description.

The crucial idea behind the custom annotations consists of shifting up data about the code into the executable and to render them available at run-time. Custom annotations are interpreted by programs and are used for program transformation.

Microsoft .NET provides support for implementing web services by means of custom attributes. A custom attribute named `WebMethod` is used to label methods that should be exposed as web services. A minimal web service written in C# that computes the sum of two integers is the following:

```
public class HelloWorldWS {
  [WebMethod]
    public int add(int i, int j) {return i+j;}
}
```

Once compiled, the `HelloWorldWS` class does not provide any web services interface. A different program — actually part of the Internet Information Server — is responsible for looking up reflection information within assemblies and generating a SOAP/WSDL interface to the method `add()` over HTTP.

A significant limit to the annotation model introduced by the CLI is the granularity of annotations: they can only be used on methods and not inside them to annotate code blocks. This limitation is partially bound to the provided reflective model that does not permit to reify blocks of statements. Several programs whose goal consists of administering and manipulating other programs would benefit from a finer grained model for annotations.

In this paper we present [a]C#, an extension to the C# programming language supporting custom annotations on arbitrary code blocks or statements. The language extends the syntax of the C# language to allow a more general form of annotation and provides a run-time library that extends the reflection support with operations for retrieving the information about annotations inside methods.

It might not be evident that this finer grained model for custom annotations of code can be of any use, thus we provide a detailed example of how custom annotations and [a]C# can be used to render parallel a sequential C# program. In particular, we discuss the general operations that can be used on annotated code blocks, which hide from the programmer the complexity of manipulating intermediate language instructions explicitly.

We also present the implementation of the compiler, which is realized as a source to source translator. In particular, it is interesting how the compiler reduces the extended model for custom annotations to the existing one with the help of small modifications of the generated intermediate language.

2 [a]C# IN A NUTSHELL.

[a]C# extends the original language by allowing the use of custom attributes inside a method body.

In the example, we have defined two attributes, `MyAnnotationAttribute` and `AnotherAnnotationAttribute`, that we will use for annotating the method body. They inherit from `ACS.CodeAttribute` rather than from the `System.Attribute` class.

```
class MyAnnotationAttribute : ACS.CodeAttribute {}
```

```
class AnotherAnnotationAttribute : ACS.CodeAttribute {}

class Example {
 public static void Main(string[] args) {
   // Code without annotations
   [MyAnnotation] {
     // Code under the aegis of the MyAnnotation attribute
     [AnotherAnnotation] {
       // Code inside a nested annotation
     }
   }
   [AnotherAnnotation, MyAnnotation] // Single statement
 }
}
```

Inside the method `Main()` we use these two annotations and, as it is usual in C#, we omit the `Attribute` suffix within square brackets. Inside a method body custom attributes can be used before statements or code blocks delimited by braces.

The ability of putting custom annotations into methods is of little use without any means to retrieve the information the programmer has specified using the annotations. How can a tool retrieve annotations from an assembly? The run-time of the language (a library linked to the program resulting from the source to source transformation) provides the method `GetCustomAttribute()` to look at the method code for custom attributes:

```
public class Annotation {
 // ...
 public static AnnotationTree[]
   GetCustomAttributes(MethodInfo m, bool rec)
 // ...
}
```

Although we cannot extend the reflection library, this method should be considered as belonging to the `MethodInfo` class: given a method reification, it returns all the annotations associated with it (if present) by reflective inspection.

The retrieved annotations are stored into a forest. Each tree represents a group of nested annotations rooted at the outermost one, whereas each node represents and stores information about a single annotation. Nested annotated blocks are sub-trees of the node representing the outer annotation sorted as they appear.

Although, to be able to retrieve annotations related to a single method body is often enough, there are still situations where a broader view is necessary. For example, consider a tool whose aim consists of performing the static analysis of a program by means of *pre* and *post conditions*. In this case, the annotation tree of a single method can be of little use because the conditions about the program's
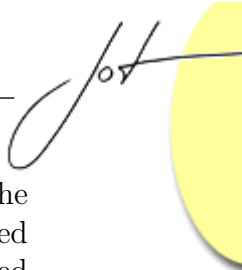
state do not only depend on a method body but also on the methods it calls. The `rec` argument of `GetCustomAttributes()` serves this purpose: if true the returned annotation forest represents the closure of the annotations attached to the specified method body and, recursively, of the annotations attached to body of the methods it calls. In this case, the leaves at a given level represent both nested annotations and annotations attached to the body of methods called in the scope of the annotations in the upper level.

The [a]C# run-time provides operations to manipulate the intermediate language instructions within the scope of annotations. These operations are:

- *Extrusion*: is used to extrude the annotation by generating a new method whose body and arguments are respectively the annotated code and the free variables of the annotation;

- *Injection*: is used to insert code immediately before and after an annotation;

- *Replacement*: is used to replace the annotated code with the specified code.

All these operations can feed an `ILGenerator` that generates the body of a method. A tool can generate new methods, which are a translated version of the original ones, using these high level operations, without the need for explicitly manipulating `IL` instructions.

Code injection simply requires an `IL` stream that corresponds to a method `m` with a signature **void** `m()`. This operation is useful for tools like aspect weavers. The same can be done for replacement. A variant of injection and replacement allows the generated code to access the same variables used by the code block target of the operation.

3   CASE STUDY: TO PARALLELIZE A METHOD.

To render parallel a program provides a good example of application of our model for custom annotations of code blocks. Suppose we are interested in giving hints about how to parallelize the execution of a method. Of course, these hints can be ignored and the method is executed sequentially. On the other hand, specific tools could recognize these annotations and transform the method consequently.

We have chosen this case study because it offers a relevant application domain where the [a]C# annotations schema proves to be useful. In our case study we focus on the code manipulation aspects rather than on the techniques to render it parallel.

The Used Annotations.

With the standard model of custom attributes it is possible to specify information about a method, but there is no easy way to indicate which portions of its body can

be run in parallel: in this case the information about the scope of the annotation is required. We introduce the following annotations:

```
public class Parallel : ACS.CodeAttribute {
  public string Description;
  public Parallel() { Description = string.Empty; }
  public Parallel(string d) { Description = d; }
}
public class Process : ACS.CodeAttribute {
  public string Description;
  public Process() { Description = string.Empty; }
  public Process(string d) { Description = d; }
}
```

These definitions must be known by the programmer who uses them to annotate the code, and by tools that understand them and behave depending on the annotations. Note that we provide a constructor that allows the user to specify some descriptive text to be included with the annotation. Note that we use the names `Parallel` and `Process` for the annotations, but this does not imply by any means that parallelism is achieved through system processes. In general it is the tool that defines the semantics of annotations by manipulating the annotated code.

Annotated Code.

A method annotated with these annotations may look like the following:

```
public void m() {
  Console.WriteLine("Parallelable code sample");
  [Parallel("Begin of a parallel block")] {
    Console.WriteLine("Code exec by the main thread");
    [Process("First process")] { /* Computation here */ }
    [Process] { /* Computation here */ }
  }
  Console.WriteLine("Here is sequential");
}
```

Listing 1: Hints for Parallelizing a Method

We notice that a sequential computation can be annotated by declaring portions of code that can be executed either sequentially or in parallel. If we execute the method m, as specified by the CLI virtual execution system, annotations are ignored and the code is executed sequentially. A tool capable of understanding the annotations `Parallel` and `Process` is required, as in the case of web services implementation based on the `WebMethod` attribute. This tool will be responsible for giving a semantics to the annotated methods.

Giving Semantics to Annotations.

In the domain of parallel execution we can even think of a JIT compiler aware of these annotations leading to better performing code. Nevertheless, an off-line approach can be also followed: a tool can read the method `m` in its binary form from the assembly where it lies; and it can generate a new assembly with a method `m` whose body has been translated so that its execution will be rendered parallel according to the annotations. Our tool will give semantics to the annotations as follows:

- `Parallel`: a block annotated with this attribute denotes a scope within the code, where the blocks annotated with `Process` are executed in parallel. When the execution control reaches the end of the block, all the processes spawned within should have finished their activity.

- `Process`: the code within a `Process` block is executed by a separate execution thread. In our implementation the code is responsible for ensuring that the access to shared variables does not cause race conditions.

The transformation schema we have adopted for transforming an annotated method `m` in its parallel form relies on the asynchronous methods support provided by CLI through delegate objects. Given a delegate object (wrapping a method) it is possible to invoke the corresponding method by calling the method `BeginInvoke()`. A thread is taken from the `ThreadPool` to execute the method referred by the delegate and the result of the invocation is an object representing the running method. The main thread calls the method `BeginInvoke()` for each `Process` block. At the end of the `Parallel` block a `WaitAll()` method call is performed to wait for all the asynchronous methods to complete, then the execution continues with the single thread.

Our tool relies on the operations provided by the [a]C# run-time for implementing this code transformation, as follows:

- for each `Parallel` block (obtained from the `GetCustomAttributes()` method):
  - it extrudes a method for each `Process` block within its scope;
  - it generates a delegate type for the extruded methods;
  - an asynchronous call to a delegate wrapping the extruded method replaces the corresponding `Process` block;

- finally, it injects a call to the method `WaitAll()` with the references to the handlers of each extruded method.

The algorithm shows how the operations defined in [a]C# run-time are useful for transforming the code.

Unbound Process Spawning.

At first sight it may be thought that the annotations we have introduced allow only expressing a fixed number of processes to be spawned by the translated code. This is not true: in our transformation schema, if we annotate the body of a **for** loop the main thread will spawn an asynchronous method for each execution of the body. We consider the classic example of paralleling the rendering of Mandelbrot fractals (see [9]). Our goal is to draw the fractal by using four threads, one for each fourth of the drawing area.

Of course we could repeat four times the code needed to draw a single portion and annotate the four blocks as `Process`. Although this approach works it requires us to replicate a portion of code. Thus, to avoid code replication, we introduce a loop that repeats four times the code that draws a region of the fractal. The resulting code is the following:

```csharp
public void Mandelbrot(Complex z1, Complex z2, int xsteps, int ysteps) {
    // rects contains the upper left corner of each region.
    Complex[] rects = new Complex[] { z1,
            new Complex((z2.Re + z1.Re)/2, z1.Im),
            new Complex(z1.Re, (z2.Im + z1.Im)/2),
            new Complex((z2.Re + z1.Re)/2, (z2.Im + z1.Im)/2)
        };
    double dx = (z2.Re - z1.Re) / xsteps;
    double dy = (z2.Im - z1.Im) / ysteps;

    [Parallel] {
      for (int count = 0; count < 4; count++)
        [Process] {
          for (int i = 0; i < (xsteps / 2); i++)
            for (int j = 0; j < (ysteps / 2); j++) {
              Complex c = new Complex(
                  rects[count].Re + dx * i,
                  rects[count].Im + dy * j), z = c;
              int it = 100;
              while (it-- > 0 && z.SqrModule < 4) z = z * z + c;
              DrawPixel(i, j, xsteps, ysteps, count, it);
            }
        }
    }
}
```

The Mandelbrot method computes the fractal given a region in the complex plane (bound by the two complexes `z1` and `z2`). The algorithm implemented subdivides the complex plane in four regions (whose upper left corners are contained in the `rects` array), and performs the classic Mandelbrot algorithm on each of these. We used the Process annotation on the block of the first statement. Accordingly to
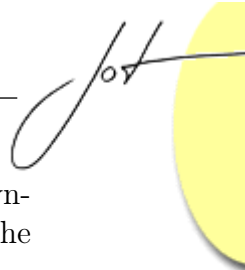
our transformation schema, the block will be replaced by the invocation of an asynchronous method. Note that the code can also be executed as it is; in this case the four regions will be drawn in sequence rather than in parallel.

The spawning of asynchronous methods in the example is known (though not reading the annotation); a variant of the algorithm may determine at run-time the number of regions that should be drawn in parallel.

Final Considerations.

The annotations we used to render parallel sequential methods introduce a form of parallelism with shared memory. The variables, defined outside the scope of the `Process` attribute, are shared among processes (which in fact are threads) though the tool does not provide any form of automatic synchronization. The variables defined inside the scope of the `Process` attribute are local to the process itself.

Without our extensions the standard model for annotations would not have been enough for expressing scopes inside method bodies. However, some may wonder if the employment of different techniques would have led us to the same result.

An alternative approach to compute the Mandelbrot set in parallel could consist of employing a tool performing source to source transformations. The disadvantage of this approach is that, unless the tool implements a full parser for the C# language, a shallow parsing approach would have been more error prone than ours that simply has to detect and retrieve the annotations. Moreover, annotations in the binary code allow different languages to benefit from the tools written to use them. Finally, the decision of transforming the program or of executing it sequentially can be postponed and the transformation can be adapted to the particular machine where the program will run.

4   COMPILER IMPLEMENTATION

The [a]C# compiler has been implemented as a source to source compiler, reducing our enriched model to the standard model for code annotation. We believe that our approach in the realization of the compiler is another contribution of this paper in two ways: first of all we show how the more expressive code annotation schema proposed by [a]C# can be reduced to the existing annotation model combined with a particular coding convention schema; the transformation engine constitutes a second contribution because of its general structure based on a full parser rather than on a shallow system.

Representing Code Annotations

As already discussed, the existing model for custom annotations cannot keep track of the attributes' scope within the body of a method. Nevertheless, the execution environment already provides all the machinery to support annotations.

We have considered extending the virtual machine by modifying Rotor [15]. Although feasible, the problem of this approach is that the code generated by our compiler would have been compatible only with the modified version of the runtime. Thus, we have decided to encode information about ranges of code annotations by inserting placeholders into the bytecode. As shown in [3, 4], we used a call to a dummy method to indicate the beginning and the end of an annotated block. Annotations are lifted onto methods and indices are used to preserve the binding between dummy method calls and the relative annotations.

Consider, for instance, the [a]C# code in Listing 1. The [a]C# compiler transforms it into the C# equivalent method:

```
[Parallel("Parallelable code sample", ACSIndex=1)]
[Process("First process", ACSIndex=2)]
[Process(ACSIndex=3)]
public void m() {
  Console.WriteLine("Parallelable code sample");
  Annotation.Begin(1); { // [Parallel]
     Console.WriteLine("Code exec by the main thread");
     Annotation.Begin(2); /* [Process("First process")] */ { ··· }
     Annotation.End(2);
     Annotation.Begin(3); /* [Process] */ { ··· }
     Annotation.End(3);
  } Annotation.End(1);
}
```

We rely on a language feature provided by custom attributes in C#: the constructor of a custom attribute allows specifying named arguments after the standard arguments. Each named argument corresponds to the assignment to a class property. In the previous example the second `Process` attribute is instantiated calling the parameterless constructor, and then the `ACSIndex` property of the instance is set to the value 3.

Classes defining code annotations must inherit from `ACS.CodeAttributes` class. Through this class we are able to inject the `ACSIndex` property into all the code annotations, allowing the transformation system to preserve the association between the attribute lifted to the method and its scope. The definition of the base class for code annotations is:

```
[AttributeUsage(AttributeTargets.Method,
          AllowMultiple=true, Inherited=true)]
```

```csharp
public class CodeAttribute : Attribute {
  private int idx;
  internal ILCursor.State beg;
  internal long end;

  public int ACSIndex {
    get { return idx; }
    set { idx = value; }
  }
  public ILCursor ILInstructions {
    get { return ILCursor.RestoreCursor(beg); }
  }
  public long EndPosition { get { return end; } }
}
```

The class `CodeAttribute` inherits from `Attribute`. This allows to use its instances as custom attributes in C#. The data about the scope of the annotation are stored within the compiled IL.

The language run-time support is responsible for retrieving the information stored within the executable file in the form of custom attributes plus the `Begin`/`End` method invocation pairs.
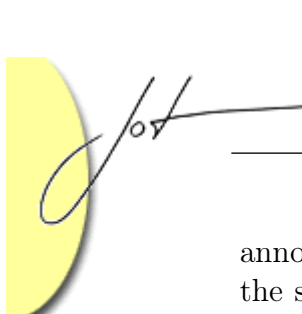
A finite state automaton reads the method body looking for method calls to the static methods `Begin()` and `End()` of the `Annotation` class. When the appropriate `call` instruction is found the automata looks for the instruction used to load the integer arguments on the arguments stack and used to relate the method call to the scope defined by an attribute.

The invocation of the `GetCustomAttributes()` method causes the language run-time to analyze the method body and fill the reflection objects used to describe the annotation tree. The `AnnotationTree` class is used to retain not only the attribute object, but also the information needed to locate the begin and the end of the intermediate language instructions within the annotation's scope.

The indices used as arguments to `Begin()` and `End()` methods are used to associate the scope with the corresponding custom attribute. It is worth noting that the annotations generated by the [a]C# compiler can be consumed by all the languages targeting the CLI execution environment.

Annotations and Run-time Execution

Our translation schema introduces method calls into annotated programs. It is natural to ask whether these calls may introduce performance hits with respect to their non-annotated versions. One of the most important assumptions of code

annotations is that the execution of an annotated program should run as fast as the same program without the annotations. Besides, we rely on fake method calls to preserve the scope of annotations. Therefore, it is natural to wonder if these IL instructions may introduce inefficiencies when the container method is executed.

It is worth noting that the called methods are static and empty. A JIT compiler can easily avoid generating the call instruction by simple inlining strategies. Nevertheless, they might generate a call instruction in the code generated by the JIT compiler. As a matter of fact all the non trivial JIT compilers perform inlining and are capable of removing the method call at JIT time. This is the behavior of the Microsoft .NET JIT, and of the Mono JIT. The Rotor JIT is very trivial and emits the call instruction.

Another issue we may think of is: does the compiler eliminate the method call to optimize the generated code? The answer is negative: the compiler links the method that is already compiled in a different assembly. Thus, it does not know the actual definition of the method demanding this optimization at later stages in the execution pipeline. Moreover, it cannot change the position of the method call inside the method body because it cannot establish whether the static method will perform side effects that may lead to a different semantics. Besides, the JIT compiler will have full knowledge of the code to be executed and will perform these optimizations if possible.

We conclude that it is reliable to insert method calls and look for them in the binary output to find the boundaries of custom annotations. Moreover, in any reasonable implementation of the CLI standard the annotations would not affect the performance of annotated programs at all. Other compilers may support the same mechanism by simply adopting the same conventions when generating code and linking the [a]C# run-time library.

CoCo/R and the C# Transformation System

The [a]C# compiler transforms an annotated program into a standard C# program using the transformation schema previously illustrated. A few transformations are necessary but they require information about the program structure: attributes must be extended with the `ACSIndex` value, though the existing arguments should be preserved. Moreover, attributes should be lifted on the method they are used. A shallow approach to program transformation is, in this case, risky and error prone. We decided to rely on a full parser of the C# 2.0 [1] syntax. Fortunately, the CoCo/R project [18] produced a parser generator for C# and a full annotated grammar for the language.

The C# annotated grammar has been modified in order to obtain a parser that parses an C# program and rewrites it. The buffer where the output is stored provides operations to insert marks within the code, and then to insert text in a marked site. Thus, we obtained a transformation system for C# that performs full syntax parsing to implement code transformation patterns.
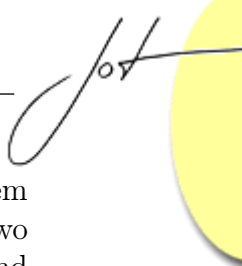
The [a]C# compiler has been developed by changing this transformation system (that implements the identity transformation). As an example, we include two productions of the annotated grammar (semantic actions are delimited by "(." and ".)"):

```
PRODUCTIONS
  (*-----------------------------------------------------------------------*)
  ClassBody = "{"            (. ob.IndentMore(); ob.WriteLine(" {"); .)
    { { AttributeSection } (. Modifiers m = new Modifiers();
                              annotation = 0; ob.Mark("ACS");
                              ob.WriteLine(); .)
      { MemberModifier<m>   (. ob.Write(" "); .)
      } ClassMemberDecl<m>
    }
    "}"                      (. ob.IndentLess(); ob.WriteLine();
                              ob.Write("}"); .)

  .
  (*-----------------------------------------------------------------------*)
  EmbeddedStatement          (. bool isPointer; MarkedBuffer obbak; .)
   = Block                   (. ob.WriteLine(); .)
   | ";"                     (. ob.WriteLine(";"); .) /* empty statement: */
                               ...
   | "while" "("             (. ob.Write("while ("); .)
     Expr                    (. ob.Write(")"); .)
     ")" EmbeddedStatement
                               ...
   |                         (. obbak = ob; ob = new MarkedBuffer();
                               inACS = true; .)
     AttributeSection        (. inACS = false;
                               obbak.WriteLine("ACS", ob.ToString());
                               ob = obbak;
                               int myann = annotation++;
                               ob.WriteLine("ACS.Annotation.Begin("+myann+");"); .)
     EmbeddedStatement       (. ob.WriteLine();
                               ob.WriteLine("ACS.Annotation.End("+myann +");"); .)
  .
  (*-----------------------------------------------------------------------*)
```

The first production defines the body of a class; we mark the output buffer ob with a label called ACS each time we encounter a member of the class. This mark is used to insert the attributes found while parsing the method body.

The second production defines a statement of the language; it has been extended with a case for statements with attribute. In the associated action we see that we inject the attribute definition where the ACS mark lies in the output buffer.

We derived the transformation system by customizing a generic C#-to-C# rewriting system based on the CoCo/R parser. The transformations needed by the [a]C# compilation strategy cannot be achieved without full understanding of the source code: the compiler needs to have information about types in order to ensure that code annotations are of types derived from CodeAttribute and that the attribute lifting process preserves the semantics of the original program. The whole annotated
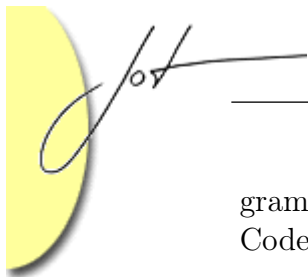
grammar and the whole source code of the [a]C# compiler is available as part of the
CodeBricks project[1].

Programming Tools

A well-known problem with source to source tools is that the programmer manipu-
lates a source file different from the one the compiler deals with. Fortunately, C#
re-introduced (similar to the C/C++ preprocessor directive) the #line directive.
Thus, we annotate the lines of the generated C# file with the reference to the line in
the original source. This is important because in the process of parsing and rewriting
the source code we loose the original formatting of the code.

As a consequence, the whole programming infrastructure, ranging from the com-
piler to the debugging tools, is capable of supporting the real source code rather
than the one generated by the translator.

The command line version of the [a]C# compiler is just a front-end to the stan-
dard C# compiler: it preprocesses all the files with .acs extension and then passes
all the file names of the generated files to the C# compiler. The front-end allows
mixing .acs and .cs files.

We have also integrated the [a]C# compiler in Visual Studio.NET 2003. The
Visual Studio programming environment includes the notion of a custom tool. A
custom tool is a preprocessor that the programming environment runs before start-
ing the compilation process. A custom tool is a COM component implementing a
set of interfaces that the editor uses to manage the code generator. We developed
the custom tool as a .NET library exposed as a COM component using the inter-
operability facilities of the execution environment. We have also added wizards to
the programming environment so that the creation of attributes and [a]C# files is
supported by the programming environment.

5   RELATED WORK.

Code annotation, in several different forms, is not a new idea and is largely used
in several contexts. Some of the most interesting applications are related to code
instrumentation, analysis and documentation. Mainly, they have been used to en-
hance the flexibility and the efficiency of the compiling step and to support new
language features, see [10, 11, 12].

Code generation is another of the most diffuse application for code annotations.
The XDoclet[2] tool [17] for Java has been successfully used for performing code gen-
eration tasks. The tool relies on comments similar to those used by Javadoc to

---

[1]See at http://www.robotics4.net/Software/ACS.aspx.
[2]See at http://xdoclet.sourceforge.net/xdoclet/index.html.

annotate programs, and then the `ant`[3] building tool is used for controlling the code generation. This approach to code annotation is based on source code manipulation: XDoclet comments are used by Java modules to generate Java source code. On the other hand, [a]C# provides operations to retrieve annotations at run-time on executable files without any need for a whole compilation infrastructure to be available at run-time.

Program manipulation with bytecode transformation is a technique that has been employed in several applications [13, 16]. There are also attempts to raise the abstraction level of this bytecode manipulation. The Javassist [6] library attempts to provide the programmer with operations that do not require knowledge on the underlying bytecode. However, the Javassist approach still focuses on the manipulation of single instructions like method calls, field accesses and so on. We are trying to build meta-programming abstractions that have coarser grain though focus more on the behavioral aspect of the code. Operations such as extrusion, injection and replacement can be type checked and provide the programmer with operations that recall the Lego bricks: "take this block of code and put it here". As a matter of fact the CodeBricks library [2, 4] performs similar tasks using entire methods as the unit for code manipulation.

Code annotation is becoming more and more relevant also in the novel aspect-oriented technology. Alice [7, 8] is an aspect-oriented based approach that exploits annotations to provide additional meta-information about components they are associated with. The annotations identify components' constraints and requirements that the weaver has to consider. In [5], the annotations are used to mark the join points inside the bytecode with the high-level specification to simplify the join point selection. Both examples could benefit from an annotation model that supports block-level annotations (as the one provided by [a]C#).
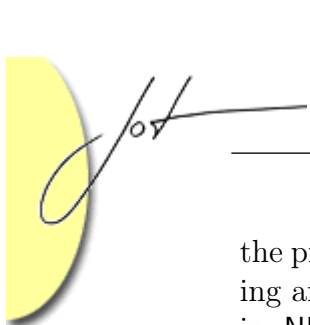
6   CONCLUSIONS AND FUTURE WORK

In this paper we have presented [a]C#, an extension of the C# language that allows custom annotations inside method bodies. The run-time library of the language extends the .NET reflective information by providing operators to retrieve custom annotations and to manipulate the annotated code blocks in their binary form.

We have presented a case study in the domain of code parallelism. The aim of the exercise was to show how a well-known problem can be faced by writing an external tool that manipulates a program rather than extending a programming language. We have also shown how the operations provided by the language are used to transform a method into its parallel version.

The compiler has been implemented as a source to source transformation. Nevertheless, we avoided the shallow approach by adopting a full C# parser that rewrites

---

[3]See at http://ant.apache.org.

the program after parsing. We have been able to transform the program during parsing and reduce the extended model for code annotations to the one already present in .NET. We have encoded the annotations' scope by inserting dummy method calls at the beginning and at the end of each block.

The compiler is still in its early stage, and the run-time of the language should still evolve. The operations for manipulating bytecode still expose some of the complexity of the underlying execution environment. Moreover, we are considering the possibility of integrating the CodeBricks library [4] into the language run-time. In this way we could expose annotated code blocks as code bricks that can then be mixed to work as basics for the operations provided by the library.

We believe that code annotations can be useful to develop software that meets the always increasing demand for flexibility and dynamic adaptation. We are exploring the possibility of using code annotations to describe the behavioral aspect of components. In particular, in the embedded device domain, devices may give the code to access them (as it is in Jini) annotated with a description of the operations performed by methods.

References

[1] Tom Archer and Andrew Whitechapel. *Inside C#*. Microsoft Press, second edition, 2002.

[2] Giuseppe Attardi and Antonio Cisternino. Multistage Programming Support in CLI. *IEE Proceedings Software*, 150(5):275–281, October 2003.

[3] Giuseppe Attardi, Antonio Cisternino, and Diego Colombo. CIL + Metadata > Executable Program. *Journal of Object Technology. Special issue: .NET: The Programmer's Perspective: ECOOP Workshop 2003*, 3(2):19–26, 2004.

[4] Giuseppe Attardi, Antonio Cisternino, and Andrew Kennedy. Code Bricks: Code Fragments as Building Blocks. In *Proceedings of 2003 SIGPLAN Workshop on Partial Evaluation and Semantic-Based Program Manipulation (PEPM'03)*, pages 66–74, San Diego, CA, USA, 2003.

[5] Walter Cazzola, Sonia Pini, and Massimo Ancona. AOP for Software Evolution: A Design Oriented Approach. In *Proceedings of the 10th Annual ACM Symposium on Applied Computing (SAC'05)*, pages 1356–1360, Santa Fe, New Mexico, USA, on 13th-17th of March 2005. ACM Press.

[6] Shigeru Chiba and Muga Nishizawa. An Easy-to-Use Toolkit for Efficient Java Bytecode Translators. In Frank Pfenning and Yannis Smaragdakis, editors, *Proceedings of Generative Programming and Component Engineering (GPCE'03)*, LNCS 2830, pages 364–376, Erfurt, Germany, September 2003. Springer.
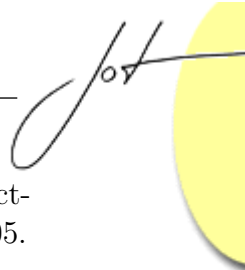
[7] Michael Eichberg. Component-Based Software Development with Aspect-Oriented Programming. *Journal of Object Technology*, 4(3):21–26, April 2005.

[8] Michael Eichberg and Mira Mezini. Alice: Modularization of Middleware Using Aspect-Oriented Programming. In Thomas Gschwind and Cecilia Mascolo, editors, *Proceedings of the 4th International Workshop on Software Engineering and Middleware (SEM04)*, LNCS 3437, pages 47–63, Linz, Austria, September 2004. Springer.

[9] Benoit Gennart and Roger D. Hersch. Computer-Aided Synthesis of Parallel Image Processing Applications. In *Proceedings of the Conference on Parallel and Distributed Methods for Image Processing*, pages 48–61, Denver, USA, 1999.

[10] Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers. Annotation-Directed Run-Time Specialization in C. In *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM'97)*, pages 163–178, Amsterdam, The Netherlands, June 1997. ACM Press.

[11] Andreas Hartmann, Wolfram Amme, Jeffery von Ronne, and Michael Franz. Code Annotation for Safe and Efficient Dynamic Object Resolution. *Electronic Notes in Theoretical Computer Sciences*, 82(2), 2003.

[12] Raimund Kirner and Peter Puschner. Classification of Code Annotations and Discussion of Compiler-Support for Worst-Case Execution Time Analysis. In *Proceedings of the 5th Euromicro International Workshop on Worst-Case Execution Time Analysis (WCET'05)*, Palma, Spain, July 2005.

[13] Hidehiko Masuhara and Akinori Yonezawa. Run-time Bytecode Specialization: A Portable Approach to Generating Optimized Specialized Code. In *Proceedings of Programs as Data Objects, Second Symposium, PADO'01*, 2001.

[14] James Miller. *Common Language Infrastructure Annotated Standard*. Addison-Wesley, November 2003.

[15] David Stutz, Ted Neward, and Geoff Shilling. *Shared Source CLI Essentials*. O'Reilly, March 2003.

[16] Éric Tanter, Marc Ségura-Devillechaise, Jacques Noyé, and José Piquer. Altering Java Semantics via Bytecode Manipulation. In Don S. Batory, Charles Consel, and Walid Taha, editors, *Proceedings of Generative Programming and Component Engineering (GPCE'02)*, LNCS 2487, pages 283–298, Pittsburgh, PA, USA, October 2002. Springer.

[17] Craig Walls and Norman Richards. *XDoclet in Action*. Manning Publications, December 2003.

[18] Albrecht Wöß, Markus Löberbauer, and Hanspeter Mössenböck. Compiler Generation Tools for C#. *IEE Proceedings Software*, 150(5):323–327, October 2003.

ABOUT THE AUTHORS

Walter Cazzola (Ph.D.) is currently an assistant professor at the Department of Informatics and Communication (DICo) of the Università degli Studi di Milano, Italy. His research interests include reflection, aspect-oriented programming, programming methodologies and languages. He has written and has served as reviewer of several technical papers about reflection and aspect-oriented programming.

Antonio Cisternino is research fellow at the Dipartimento di Informatica of Università di Pisa. His current research is on run-time code generation and multi-stage programming on execution environments like JVM and CLI.

Diego Colombo is PhD student at IMT Lucca. He is interested in robotics, computer vision, 3D graphics and Game Design.