Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 253 (2009) 97–111
www.elsevier.com/locate/entcs

Towards an XML-based Bytecode Level Transformation Framework

Arno Puder[1] and Jessica Lee[2]

San Francisco State University
Computer Science Department
1600 Holloway Avenue
San Francisco, CA 94132

Abstract

Virtual machines (VMs) facilitate the deployment of applications in heterogeneous environments. Popular VMs such as Sun Microsystem's Java VM or Microsoft's Common Language Runtime (CLR) use a stack-based machine architecture to execute bytecode instructions. The focus of this paper is to explore the ability of XML to serve as a generic framework to represent bytecode instructions from different VMs. With an XML representation, supporting technologies such as XSL stylesheets and XPath expressions can be used to provide various transformations of bytecode instructions. We demonstrate the flexibility and power of this approach by showing examples of cross-compilation for various CLR bytecode instructions to the JVM as well as API mappings of the underlying runtime libraries.

Keywords: Virtual machine, Java VM, Microsoft CLR, XML

1 Introduction

Virtual machines are abstract computing machines that offer a homogeneous computing platform in a heterogeneous environment. Instead of compiling source code to machine language, an intermediate language called *bytecode* is produced. It cannot directly run on a physical machine and requires a virtual machine that loads and executes the application. While often confronted with the argument of being slower in execution, they offer a number of advantages over machine compiled languages. Specifically, they ease the development and deployment of applications in heterogeneous environments, without the need to recompile the application for a specific platform. Two virtual machines that are widely used today are the Java Virtual Machine (JVM) from Sun Microsystems [9] and Microsoft's Common Language Runtime (CLR) as part of their .NET framework [2]. Implementations can

[1] Email: arno@sfsu.edu
[2] Email: jeslee@sfsu.edu

doi:10.1016/j.entcs.2009.11.017

be found on devices like cell phones, personal computers, or chip-cards. Both virtual machines rely on a stack-based execution model, but the CLR features a wider range of data types and bytecode instructions than the JVM.

Programs to manipulate virtual machine bytecode are becoming more prevalent. BAT2XML [3] uses XML to represent Java bytecode in order to take advantage of supporting XML technologies for processing and manipulation. One of the features of BAT2XML allows for the easy injection and extraction of Java bytecode. This means that the bytecode can be analyzed and then optimized. BAT2XML is only able to represent Java bytecode, however, and has no provisions for CLR bytecode.

Aspect-Oriented Programming (AOP, [7]) provides another example of the need to manipulate bytecode. AOP is concerned with separating the basic business logic of a system from what it deems to be *crosscutting concerns* such as logging and authorization. These concerns are shared by modules all throughout a system. Rather than being added at the source code level, they are separated out into aspects and added in at compile time by an *aspect weaver*. This "weaving" is done on a bytecode level.

Cross-compilation is yet another example of a situation where byte code is directly processed. It would be of obvious economic interest to be able to map bytecode instructions from one VM to the other. By doing so, developers for one VM could deploy their applications on the other VM. The IKVM project [4] offers support to execute JVM applications on a CLR platform. The JaCIL project [5] cross-compiles .NET executables to run on a JVM. However, neither of these approaches can provide a generic representation of bytecode for both the CLR and the JVM.

The approach taken in this paper is to make extensive use of XML technologies in order to provide a framework for generic bytecode manipulation. We use XML in order to provide a representation of bytecode that is a superset of stack-based machine languages. This allows for the easy manipulation of bytecode, and we demonstrate this with a cross-compilation example that uses XSL stylesheets [10] for translating CLR bytecode instructions to the JVM. XSL stylesheets allow for a declarative methodology when performing this transformation.

Ultimately, this paper is a showcase for the power of XML technologies and declarative programming for a non-trivial application. The outline of this paper is as follows: In Section 2 we present XMLVM, our XML-based representation of bytecode instructions. Section 3 gives an overview of the JVM and the CLR while focusing on their differences. Based on XMLVM, we show in Section 4 how to cross-compile bytecode instructions using XSL stylesheets. Section 5 presents conclusions and outlook for future work.

2 XMLVM

Cross-compilation requires access to the bytecode instructions for both the CLR and the JVM. From an engineering perspective, this can be accomplished by various libraries that allow the inspection and construction of executables. For Java class files, the Byte Code Engineering Library (BCEL, [1]) allows such manipulations. For

CLR executables, the .NET framework offers a similar library based on a reflection API for low-level bytecode manipulations. It should be noted that BCEL can only be accessed using Java while the .NET API is only offered for languages supported by .NET. Currently, there is no single library that allows JVM and CLR bytecode manipulations from one high-level programming language (e.g., Java or C#). Any kind of cross-platform bytecode manipulations are therefore difficult to realize. For that reason we have chosen to create an abstraction for bytecode instructions using XML [11] and limit the use of platform-specific libraries only during reading and writing JVM and CLR executables.

We use XML to represent the contents of a Java class file as well as the contents of a CLR executable. Since the resulting XML document is structured according to the semantics of a program for a generalized virtual machine, we call it XMLVM. Another way to look at XMLVM is that it defines an assembly language for those virtual machines using XML as the syntax. The benefit of using XML is that it creates an abstraction in the sense that it hides the complexities of bytecode manipulation libraries. It thereby allows us to focus on the bytecode manipulations itself by exploiting powerful XML technologies such as XSL and XPath. In addition, the declarative nature of XSL will result in compact specifications for mapping CLR bytecode instructions to the JVM.

The following template shows the general structure of an XMLVM translation unit used for both JVM and CLR programs:

```
<xmlvm xmlns:clr="http://xmlvm.org/clr"
       xmlns:jvm="http://xmlvm.org/jvm"
       xmlns="http://xmlvm.org">
    <class ...>
        <field .../>
        <method ...>
            <signature>...</signature>
            <code>...</code>
        </method>
    </class>
</xmlvm>
```

An XMLVM program consists of several classes, each contained in a separate translation unit. Each class can have one or more fields and methods. The attributes of the XML tags, which are not shown in the template above, give more details such as identifier names or modifiers. A method is defined through a signature and the actual implementation, denoted by the tags <signature> and <code> respectively. We make use of XML namespaces to indicate the semantics of the various tags used in an XMLVM program. The tags shown in the template above are located in the default namespace and represent common features between the CLR and the JVM. Those features specific to the particular VM—such as different byte code instructions—are located in their respective namespace. Consider the following simple C# program that will serve as an example when discussing the mapping of numerical operators:

```
// C#
using System;

class AddTest {
```

```
5
6     public static void Main() {
7         int a = 11;
8         int b = 22;
9         Console.WriteLine(a + b);
10    }
11 }
```

Class `AddTest` has one public static method called `Main`. The method adds two integer values and prints the sum to the console. The following XML shows a simplified representation of class `AddTest` in XMLVM:

```
1 <xmlvm xmlns:clr="http://xmlvm.org/clr"
2        xmlns="http://xmlvm.org">
3   <class name="AddTest">
4     <method name="Main" isStatic="true" isPublic="true">
5       <signature>
6         <return type="void" />
7       </signature>
8       <code>
9         <clr:var index="0" type="int" />
10        <clr:var index="1" type="int" />
11        <clr:ldc type="int" value="11" />
12        <clr:stloc index="0" />
13        <clr:ldc type="int" value="22" />
14        <clr:stloc index="1" />
15        <clr:ldloc index="0" />
16        <clr:ldloc index="1" />
17        <clr:add />
18        <clr:call has-this="false" class-type="System.Console" method="WriteLine">
19          <signature>
20            <return type="void" />
21            <parameter type="int" />
22          </signature>
23        </clr:call>
24        <clr:return />
25      </code>
26    </method>
27  </class>
28 </xmlvm>
```

It should be emphasized again that the above XMLVM program is essentially an XML representation of the contents of the `AddTest.exe` executable generated by a C# compiler. The top-level tags are identical to the XML template shown earlier. The <`method`> tag has attributes for each of the modifiers `public` and `static`. A method has access to its own stack and local variables as well as the global heap. If a method has actual parameters, they are automatically stored in the local variables upon entering the method.

The most interesting part of the above XMLVM program is the actual implementation of method `Main`, which lies in between the tags <`code`> and </`code`>. Since the bytecode instructions belong to the CLR, the respective XMLVM instructions are placed in the XML namespace denoted by the prefix `clr`. The <`clr:var`> (*variable*) tag declares a variable with respective type that can be addressed by a given index. Instruction <`clr:ldc`> (*load constant*) pushes a constant referred to by attribute `value` onto the stack.

The <`clr:stloc`> (*store location*) instruction pops off the top of the stack and saves it in the local variable referred to by attribute `index`. The <`clr:ldloc`> (*load location*) instruction does the inverse by pushing the content of a variable onto the stack. The instruction <`clr:add`> (*addition*) pops the last two values off

the stack and pushes their sum back onto the stack. The <clr:call> instruction invokes the method System.Console.WriteLine(). The false value of attribute has-this indicates that WriteLine() is a static method, because it does not require a this reference. Note that the actual parameters have been removed by the <clr:call> instruction after the invocation. The <clr:return> instruction finally leaves the method Main and returns to the caller.

The aforementioned <clr:add> instruction gives no indication as to the type of the operands. In this particular example, the <clr:add> instruction will compute the sum of two integers, since the two top elements of the stack are of type integer. The same <clr:add> instruction would have been used if two floating point values had been added. The CLR states that the virtual machine has to determine the correct type through some mechanism. This could be accomplished by either a static data flow analysis of the program or by maintaining a type-stack at runtime.

The <clr:add> instruction does not check for overflow. If an overflow occurs, only the least significant bytes of that type are considered for the sum of the arguments. If the C# program above had computed the sum using the expression checked((int) (a + b)), it would have resulted in the byte code instruction <clr:add_ovf> (*add overflow*) that raises a runtime exception when an overflow occurs. Note that for the <clr:add_ovf> instruction the VM also has to determine the correct type of the arguments similar to <clr:add>.

3 Overview of the JVM and CLR

An exhaustive comparison between the JVM and the CLR is outside of the scope of this paper, as our main interest lies in the exploration of bytecode manipulation with XML technologies. In the following we focus on two key features of the CLR for which there is no corresponding support in the JVM. Later in this paper we will present how these features can be mapped to the JVM. Specifically, we will demonstrate how both the CLR's type-agnostic instructions (as used with primitive types) and value types can be mapped to the JVM. Other features such as delegates and generics are left for future work. The work involved with this mapping will allow us to demonstrate the power and flexibility of using XML, in conjunction with XSL and XPath, to perform cross-compilation. The features discussed in the following already allow for non-trivial applications to be cross-compiled using a declarative approach. In the following sections we discuss primitive types (Section 3.1) and value types (Section 3.2).

3.1 Numerical Primitive Types

As with any high-level programming language, the languages based on the JVM and the CLR both define a set of primitive types such as bytes, integers, or doubles. Furthermore, both execution platforms define various bytecode instructions such as addition or subtraction that operate on these primitive types. We first discuss the different data models of the JVM and the CLR and then introduce the bytecode instructions.

The JVM supports numerical types of various sizes and precisions. In the following we focus on integer types. Based on the Java programming language, the JVM supports four different integer types, ranging from 8 to 64 bit precision. One interesting fact is that the JVM (and therefore Java) only supports signed integers. I.e., there is no built-in support for unsigned integer types. As with the JVM, the CLR also supports integer types of various precisions. One important difference however is that the CLR supports both signed and unsigned integer types.

Based on the integer types, both virtual machines offer various byte code instructions that operate on those types. In the following we introduce the bytecode instructions for adding two integer values. The bytecode instructions for subtraction, multiplication, and division follow the same pattern. The JVM features the following two different bytecode instructions for adding integers:

- <`jvm:iadd`>: adds two signed 32 bit integers of type `int`.
- <`jvm:ladd`>: adds two signed 64 bit integers of type `long`.

Interestingly, there are no special instructions for adding 8 bit and 16 bit integer values. As noted in section 3.11.1 of the JVM specification, "encoding types into opcodes places pressure on the design of [the VM's] instruction set" [9]. As a consequence, the JVM designers wanted the <`jvm:iadd`> instruction (along with other 32 bit integer-based instructions) to be able to work with both bytes and shorts. This is accomplished by sign-extending values of these types to 32 bit signed integers when loaded onto the stack.

Despite the fact that an overflow may occur, execution of an <`jvm:iadd`> or <`jvm:ladd`> instruction never throws a runtime exception. The result is the 32 low-order bits of the true mathematical result in a sufficiently wide two's-complement format, represented as a value of type `int`. If an overflow occurs, then the sign of the result may not be the same as the sign of the mathematical sum of the two values.

Whereas there are only two bytecode instructions for adding integers offered by the JVM, the CLR supports three different op-codes for adding integers:

- <`clr:add`>: adds two signed integers without overflow check.
- <`clr:add_ovf`>: adds two signed integer values with overflow check. Throws `OverflowException` if an overflow occurs.
- <`clr:add_ovf_un`>: adds two unsigned integer values with overflow check. Throws `OverflowException` if an overflow occurs.

It is interesting to note that the description for the add instruction only specifies the addition of two *signed* integers. There is no mention of unsigned integers. How can it be possible, then, that the CLR supports the basic addition of unsigned integers? It turns out that the CLR imposes a constraint somewhat similar to that of the JVM for items placed upon its evaluation stack. Only signed 32 bit and 64 bit integers can be loaded onto the stack. Unsigned types are simply sign-extended and treated as signed types when loaded onto the stack. The CLR then takes advantage of the fact that, for some operations, signed and unsigned integers do not need to be

treated differently. In those cases where they need to be treated differently, special instructions such as <clr:add_ovf_un> are used [2].

Based on this examination, we can see that there are several significant differences between the JVM and CLR with respect to their support of integer types. This complicates the efforts involved with cross-compilation. To translate bytecode from the CLR to the JVM, for example, the cross-compilation process must at least provide support for unsigned integers, use the appropriate bytecode instructions for different data types, and check for overflow by throwing a runtime exception when an overflow occurs.

3.2 Value Types

The CLR introduces the notion of *value types*. Value types are similar to classes, but their instances are allocated on the stack. Instances of classes (i.e., objects) are usually allocated on the heap and are garbage collected when not used anymore. Garbage collection introduces significant runtime overhead. However, since value types are allocated on the stack, they will be automatically reclaimed when the method where they were defined is exited. For this reason, the underlying behavior of value types in the CLR differs from that of classes. This is demonstrated by the following C# program:

```
1 // C#
2 using System;
3
4 public struct Person {
5     public string Name;
6
7     public Person(string name) {
8         Name = name;
9     }
10 }
11
12 class ValueTypeTest {
13     static void Main() {
14         Person p1 = new Person("Bob");
15         Person p2 = p1;
16         p2.Name = "Alice";
17         assert(p1.Name == "Bob");
18         assert(!p1.Equals(p2));
19     }
20 }
```

In C#, a `struct` defines a value type (line 4). Although a value type is also instantiated via the `new` operator (line 14), it will effectively be allocated on the stack. As a consequence, variable `p2` in the example above will copy the value type (line 15). If `Person` were a proper class, `p1` and `p2` would be referencing the same object. But as can be seen by the assertions in lines 17 and 18, `p1` and `p2` are separate copies. Value types can be converted to and from proper garbage collected objects. Converting a value type to an object is called *boxing*, while the reverse mapping is called *unboxing*. Apart from the boxing and unboxing operation, the CLR introduces no special bytecode instructions for handling value types. The same bytecode instructions are used for objects and value types, but their semantics are different. The JVM offers no support for value types and therefore this requires special handling during cross-compilation.

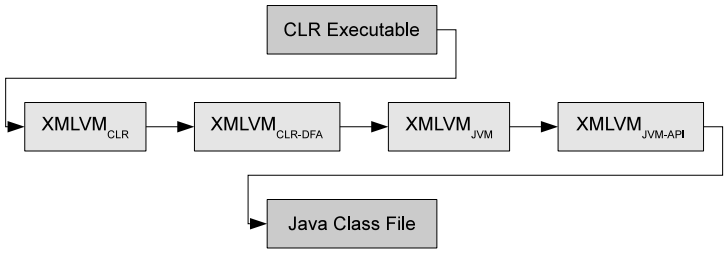

Fig. 1. XMLVM Toolchain.

4 CLR to JVM Transformation

The preceding section outlined some differences between the CLR and the JVM. The CLR is a superset of the JVM in many ways, which is underlined by the fact that the CLR supports more primitive types as well as features additional bytecode instructions not found in the JVM. Therefore, it is relatively straightforward to map a JVM program to the CLR. There already exists at least one project that implements this direction of the transformation [4].

The reverse direction of the transformation poses more challenges, and in the following we discuss some techniques to accomplish this with the use of various XML technologies. The basis for this study is the use of XMLVM, which was introduced in Section 2. Figure 1 summarizes the overall workflow of our toolchain. The transformation begins with a CLR executable that is first translated to an XML representation we call XMLVM$_{CLR}$, whose format was described earlier. The next step consists of a data flow analysis of the XMLVM program which we refer to as XMLVM$_{CLR-DFA}$. Section 4.1 details how the data flow analysis can be done via XSL stylesheets. After the data flow analysis, the CLR bytecode instructions can be transformed to the JVM (referred to XMLVM$_{JVM}$ in Figure 1), the details of which will be given in Section 4.2. After cross-compiling the bytecode instructions, the API of external libraries need to be mapped which is described in Section 4.3. Finally, the resulting XMLVM$_{JVM-API}$ program can be translated to a binary Java class file that can be executed on a standard JVM.

4.1 Data Flow Analysis

Up to this point we have shown how XML can be used to represent any program that can be executed on the JVM or the CLR. As outlined earlier, the CLR only features un-typed instructions; this is illustrated by our examination of integer addition. As a consequence, it will become necessary to know the type of the operands during the cross-compilation process. Without this knowledge, it would be impossible to map the CLR-add instruction to one of the typed add-instructions supported by the JVM. As a prerequisite, one has to determine on which specific argument types the un-typed instructions operate. This can be accomplished by a *data flow analysis*. This task is similar to what a bytecode verifier has to do when loading a program into the virtual machine [6].

During a data flow analysis, all the execution paths through a program are

traced, but instead of pushing and popping specific arguments onto the stack, only the types of those arguments are stored on the stack. For this reason it is also called a *type stack* (vs. an argument stack). With this analysis, it is possible to determine the type of the arguments that will be stored on the stack at any point during the execution of the program.

As a first step towards a data flow analysis, we take advantage of XML's extensibility and introduce new markup of an XMLVM program that can capture the effects an individual bytecode instruction has on the type stack. For each instruction, we need to record the state of the type stack before and after execution of that instruction. Those two states will be marked up with the tags <`stack-pre`> and <`stack-post`> respectively and appended as children to the XML tag of the bytecode instruction. The elements contained on a stack are denoted by the <`elem`> tag. We refer to the resulting XMLVM program that contains these type-stack annotations as $\text{XMLVM}_{CLR-DFA}$.

We use XSL templates to generate the type stack transition for each instruction. XSL templates specify translations to apply to an XML document when an XML node is found that matches the rule specified in the `match` attribute. As a specific example, consider the aforementioned <`clr:ldloc`> instruction that pushes the content of a local variable onto the stack. In terms of changes to the type stack, this instruction will push an argument of a certain type (determined by the type of the variable whose value is to be pushed onto the stack). The following is an example of the XSL template for <`clr:ldloc`>:

```
1  <xsl:template match="clr:ldloc[preceding-sibling::*[1]/dfa:stack-post]">
2      <clr:ldloc>
3          <xsl:copy-of select="@*"/>
4          <stack-pre>
5              <xsl:copy-of select="preceding-sibling::*[1]/stack-post/*"/>
6          </stack-pre>
7          <stack-post>
8              <xsl:copy-of select="preceding-sibling::*[1]/stack-post/*"/>
9              <elem>
10                 <xsl:variable name="idx" select="@index"/>
11                 <xsl:attribute name="type">
12                     <xsl:value-of select="../clr:var[@index = $idx]/@type"/>
13                 </xsl:attribute>
14             </elem>
15         </stack-post>
16     </clr:ldloc>
17 </xsl:template>
```

The DFA for this instruction can only be computed once the <`stack-post`> of the preceding instruction has been determined. This condition is given by the XPath expression of the `match` attribute (line 1). The first <`copy-of`> copies all attributes to the result tree via the `@*` expression (line 3). Next, the template defines the type stack before the instruction executes via the <`stack-pre`> tag. The state of the type stack at this point is identical with the <`stack-post`> of the previous instruction. Therefore, the XPath expression of the second <`copy-of`> selects the <`stack-post`> of the first preceding sibling of the current instruction (lines 4–6). The above template finally defines the type stack after executing the <`clr:ldloc`> instruction via the <`stack-post`> tag (lines 7–15). In case of this instruction, one new element is added to the top of the type stack whose type is determined by the

type of the variable whose value is pushed by <clr:ldloc>. The <value-of> in the template above (line 12) uses an XPath expression to reference that type. The following excerpt shows the resulting XMLVM$_{CLR-DFA}$ program for AddTest with the computed type stack transitions:

```
1 <!-- ... -->
2 <clr:ldloc index="0">
3     <stack-pre/>
4     <stack-post>
5         <elem type="int"/>
6     </stack-post>
7 </clr:ldloc>
8 <clr:ldloc index="1">
9     <stack-pre>
10        <elem type="int"/>
11    </stack-pre>
12    <stack-post>
13        <elem type="int"/>
14        <elem type="int"/>
15    </stack-post>
16 </clr:ldloc>
17 <clr:add>
18    <stack-pre>
19        <elem type="int"/>
20        <elem type="int"/>
21    </stack-pre>
22    <stack-post>
23        <elem type="int"/>
24    </stack-post>
25 </clr:add>
26 <!-- ... -->
```

In the XMLVM$_{CLR-DFA}$ excerpt above it can be seen that the data flow analysis added tags for each CLR instruction that reflects the content of the type stack before and after the execution of that instruction. As shown in lines 17 to 25, the CLR instruction <clr:add> pops off two integers and pushes an integer back onto the type stack. It should be noted that the data flow analysis done here is much less complex than what a bytecode verifier typically is required to check. In particular, since we are only interested in primitive types, it is not necessary to compute the LUB (least upper bound) of object types of different execution paths [8].

4.2 Generating JVM Bytecode

The next step of the XMLVM toolchain consists in translating the CLR instructions to JVM bytecode. In some cases this mapping is trivial, as there is a one-to-one correspondence between CLR and JVM instructions. One example is the <clr:ldnull> instruction, which pushes a null reference onto the execution stack. This can be directly mapped to the <jvm:aconst_null> instruction. In other cases, the mapping has to rely on the data flow analysis as introduced in the previous section. In the following sections we demonstrate how to map CLR bytecode instructions for numerical operations and value types to semantically equivalent JVM bytecode instructions.

4.2.1 Mapping Numerical Operations

As explained earlier, the CLR only features un-typed instructions, whereas the JVM only has typed instructions. The <clr:add> instruction of the XMLVM$_{CLR}$ program needs to be mapped to one of the typed addition operators of the JVM.

This can be accomplished with the help of the data flow analysis. The following two XSL templates demonstrate the mapping of the CLR instruction <clr:add> for integer types:

```
1 <xsl:template match="clr:add[stack-post/elem[last()][@type = 'int']]">
2     <jvm:iadd/>
3 </xsl:template>
4
5 <xsl:template match="clr:add[stack-post/elem[last()][@type = 'long']]">
6     <jvm:ladd/>
7 </xsl:template>
```

The XPath expression in the `match`-statements of the XSL templates check the top of the type stack and map the <clr:add> CLR instruction to either the JVM <jvm:iadd> instruction (line 2) or the JVM <jvm:ladd> instruction (line 6) depending whether the top of the type stack is of type `int` or `long`. This is an example of how un-typed CLR instructions can be mapped to type-specific JVM instructions via declarative XSL templates. The following XMLVM excerpt shows the JVM bytecode instructions that result from transforming the CLR implementation of the `AddTest` class introduced earlier:

```
1 <code>
2     <jvm:ldc type="int" value="11"/>
3     <jvm:istore index="0"/>
4     <jvm:ldc type="int" value="22"/>
5     <jvm:istore index="1"/>
6     <jvm:iload index="0"/>
7     <jvm:iload index="1"/>
8     <jvm:iadd/>
9     <jvm:invokestatic class-type="System.Console" method="WriteLine">
10        <signature>
11            <return type="void"/>
12            <parameter type="int"/>
13        </signature>
14    </jvm:invokestatic>
15    <jvm:return/>
16 </code>
```

In some cases it is not possible to map one CLR bytecode instruction to exactly one JVM instruction. The instructions that check for overflow (both <clr:add_ovf> and <clr:add_ovf_un>) during addition are two such cases. In order to map those CLR instructions to the JVM, it is necessary to introduce a compatibility library that mimics the semantics of the original CLR instruction. As an example, consider the CLR instruction <clr:add_ovf> discussed earlier. Since the JVM does not offer a single bytecode instruction with the same semantics, the <clr:add_ovf> instruction is mapped via the following stylesheet to an invocation of the static method `MathLib.add_ovf(int, int)`:

```
1 <xsl:template match="clr:add_ovf[stack-post/elem[last()][@type = 'int']]">
2     <jvm:invokestatic class-type="MathLib" method="add_ovf">
3         <signature>
4             <return type="int"/>
5             <parameter type="int"/>
6             <parameter type="int"/>
7         </signature>
8     </jvm:invokestatic>
9 </xsl:template>
```

This above XSL template demonstrates that bytecode instructions can be very easily inlined. The JVM tag <jvm:invokestatic> serves the same purpose as the CLR tag <clr:call has-this="false"> for static methods. Whenever the <clr:add_ovf> instruction is encountered in a CLR program, an invocation to a compatibility library is made that mimics the semantics of that instruction. The following Java class MathLib shows one possible implementation of the <clr:add_ovf> bytecode instruction for integers for which no direct correspondence exists in the JVM. Note that the signature of this method is consistent with the <clr:add_ovf> instruction:

```
1 // Java
2 public class MathLib {
3     static public int add_ovf(int x, int y) {
4         int z = x + y;
5         if (z == ((long) x + (long) y))
6             return z;
7         else
8             throw new OverflowException();
9     }
10 }
```

4.2.2 Mapping Value Types

Value types have been introduced in the CLR as light-weight objects for which the JVM offers no equivalent bytecode operations. The following XMLVM$_{CLR}$ shows the CLR bytecode instructions for lines 14-15 of the ValueTypeTest.Main() method introduced in Section 3.2:

```
1 <clr:var index="0" isValueType="true" type="Person" />
2 <clr:var index="1" isValueType="true" type="Person" />
3 <clr:ldloca index="0" />
4 <clr:ldc type="String" value="Bob" />
5 <clr:call has-this="true" class-type="Person" method=".ctor">
6     <vm:signature>
7         <vm:return type="void" />
8         <vm:parameter type="String" />
9     </vm:signature>
10 </clr:call>
11 <clr:ldloc index="0" />
12 <clr:stloc index="1" />
```

The <clr:var> declarations of type Person will allocate sufficient memory on the stack to hold instances of that value type. Since value types are allocated on the stack, they are not created via <clr:newobj>. The <clr:ldloca> (*load location address*) pushes the address where the value type is allocated onto the stack. The following <clr:call> instruction then calls the constructor (.ctor) of the value type. The combination of <clr:ldloc> and <clr:stloc> copies a value type. Note that those two byte code instructions also work with regular objects, but since they are applied to value types in this example, a deep copy is performed. Analyzing the bytecode instructions created by a C# compiler, several observations can be made:

- Value types are not allocated via the instruction <clr:newobj> that is typically used to instantiate a new object on the heap.
- Value types are allocated on the stack. The <clr:var> variable declaration

implicitly allocates memory for the new value type on the stack.

- Value types are manipulated via the same bytecode instructions as regular objects. The <clr:stloc> instruction in this case does a deep copy of a value type.

Since the JVM has no support for value types, they have to be simulated while retaining the semantics as defined by the CLR. The easiest approach is to convert value types to heap-allocated objects. If value types are mapped to objects, they need to be allocated via <clr:newobj>. Special care has to be taken when copying value types. The <clr:ldloc> and <clr:stloc> copy references for objects, while performing a deep copy for value types. In this case, the data flow analysis again helps to map these bytecode instructions to proper JVM instructions. Since a deep copy cannot be performed with a single bytecode instruction, we generate a call to a compatibility library that implements this behavior. Each value type inherits directly or indirectly from class `System.ValueType`. We add a static method `__COPY` to this class that performs the deep copy using the Java reflection API. Here is the declaration of this method:

```
1 // Java
2 package System;
3
4 public class ValueType extends System.Object {
5     static public void __COPY(ValueType from, ValueType to)
6     { ... }
7 }
```

With the assistance of this helper method, the bytecode instruction <clr:stloc> will be mapped to a static invocation of this method whenever the instruction is handling a value type:

```
1 <jvm:aload index="1" type="Person"/>
2 <jvm:invokestatic class-type="System.ValueType" method="__COPY">
3     <vm:signature>
4         <vm:return type="void"/>
5         <vm:parameter type="System.ValueType"/>
6         <vm:parameter type="System.ValueType"/>
7     </vm:signature>
8 </jvm:invokestatic>
```

The <clr:stloc> instruction expects the value type to be stored on the stack. Since we treat all value types as objects, this value type will be represented as a reference. The destination (also represented as a reference) is pushed onto the stack via <jvm:aload> before making a call into the compatibility library. The signature of method `__COPY` is chosen such that it matches the stack layout at this point in time.

4.3 API Transformation

Cross-compiling CLR bytecode instructions to JVM bytecode instructions does not solve the problem of external libraries referenced by the application. E.g., if a C# application uses WinForms to create a button on a user interface, it will reference class `System.Windows.Forms.Button`. Cross-compiling bytecode instructions would result in a Java class file having an external reference to this class that does

not exist in the Java runtime library. One solution is to create a set of compatibility classes in Java with the exact API as their CLR counterparts.

In some cases it is possible to map the API without the need of compatibility classes as can be seen with the .NET class `System.String`. Its behavior is almost identical to that of the Java class `java.lang.String`. In principle, every reference of `System.String` can be replaced with `java.lang.String` in the cross-compiled program. There are some important differences, however, in their respective implementations. One such difference is the way the length of a string is determined. Class `System.String` in .NET defines the read-only property `Length` for that purpose:

```
1 // C#
2 namespace System {
3     class String {
4         public int Length {get;}
5         // ...
6     }
7 }
```

When used in a C# program, the resulting bytecode calls an instance method of name `get_Length()`. This method has the same behavior as `java.lang.String.length()` and it can therefore be replaced. This is accomplished by the following XSL template:

```
1 <xsl:template match="jvm:invokevirtual[@class-type = 'System.String' and
2                                        @method = 'get_Length']">
3     <jvm:invokevirtual class-type="java.lang.String" method="length">
4         <vm:signature>
5             <vm:return type="int"/>
6         </vm:signature>
7     </jvm:invokevirtual>
8 </xsl:template>
```

Changing class names and method names is the essence of API mapping. When used together with API wrapping, the cross-compiled Java application behaves identically to its original CLR version. The above XSL template demonstrates the full potential of declarative XPath expression to filter out the desired API.

5 Conclusion and Outlook

In this paper we have demonstrated the feasibility of using XML technologies to perform bytecode manipulations—such as with the cross-compilation of CLR bytecode instructions to the JVM. This allows .NET developers to deploy their applications on a standard JVM. The CLR offers a wider range of features than the JVM, thereby making the cross-compilation non-trivial. We see our work as a showcase for the power of XML technologies. In particular, XSL stylesheets have proven to be a powerful abstraction for bytecode manipulations. XSL is a declarative, Turing complete language that allows us to focus on performing bytecode transformations without having to deal with byte code manipulation libraries such as BCEL or the .NET reflection API.

In this paper we have shown how to map integer operations and value types to the

JVM. Other features of the CLR remain for future work. In particular the support for generics will require significant work. Other future research will investigate the capabilities of XMLVM to represent bytecode programs that use instructions from different VMs. In particular, we plan to look at the possibility of weaving byte code instructions from different VMs into one XMLVM program in the context of AOP. This would make it possible to weave a C# aspect into a Java application and vice-versa. The cross-compilation would have to cope with mixed bytecode instructions from different VMs. XMLVM would be particularly well-suited to serve as an abstraction for this heterogeneous mix of bytecode instructions.

References

[1] Dahm, M., *Byte code engineering*, Java Informations Tage, 1999, pp. 267–277.

[2] ECMA, "Common Language Infrastructure (CLI)," 4th edition (2006).

[3] Eichberg, M., *BAT2XML: XML-based Java Bytecode Representation*, Electronic Notes in Theoretical Computer Science, 2005.

[4] Frijters, J., "IKVM.NET: A JVM for the Microsoft .NET Framework," http://www.ikvm.net.

[5] Goo, A., "JaCIL: A CLI to JVM Compiler," Http://sourceforge.net/projects/jacil/.

[6] Gosling, J., *Java intermediate bytecodes*, Proc. ACM SIGPLAN Workshop on Intermediate Representations (1995), pp. 111–118.

[7] Laddad, R., "AspectJ in Action: Practical Aspect-Oriented Programming," Manning Publications, 2003.

[8] Leroy, X., *Java bytecode verification: algorithms and formalizations*, Journal of Automated Reasoning **30** (2003), pp. 235–269.

[9] Lindholm, T. and F. Yellin, "The Java Virtual Machine Specification," Addison-Wesley Pub Co, 1999, second edition.

[10] W3C, "XSL Transformations," (1999), http://www.w3.org/TR/xslt.

[11] W3C, "eXtensible Markup Language (XML)," (2006), http://www.w3.org/XML/.