Automating the Ada Binding Process for Java - How Far Can We Go?

David E. Emery[1], Robert F. Mathis[2], and Karl A. Nyberg[3]

[1] The MITRE Corporation 1820 Dolley Madison Blvd., MS W538 McLean, VA 22102-3481 emery@mitre.org
[2] Pithecanthropus Consulting, Inc., 4719 Reed Rd., #305 Columbus, OH 43220 bob@pithecanthropus.com
[3] Grebyn Corporation, P. O. Box 47 Sterling, VA 20167-0047 karl@grebyn.com

Abstract. This paper describes an automated approach for generating Ada bindings from Java class files. We start with the set of Java features that require a visible Ada binding, and an Ada compiler's definition of how to interface Ada and Java. We discuss how to obtain the Java definitions from the class file and then translate them into an Ada binding (using the GNAT binding approach). While it is possible to generate a technically complete Ada binding from the information in a Java class file (within the constraints of necessary support from an Ada compiler). However, we show that such a binding has significant limitations from a practical usability perspective.

1 Introduction

The Java promise of "write once, run anywhere" [Kramer] (but see [Wragg]) has generated a lot of excitement and interest in the computing community. This approach to target-independent computing is achieved by generating intermediate code, which is then interpreted by a target-dependent interpreter, called the Java Virtual Machine (JVM).

The JVM [Lindholm], [Meyer], [Venners] includes a definition of the format of a Java Class File (CFF). This file, in Ada terms, contains both the "programming library information" and also the "object file" produced by many conventional Ada compilation systems. Each class has a separate class file. To execute a program contained in a specific class file, the JVM loads (on demand) the class files referred to by the program and interprets the object code accordingly. (This is analogous to the elaboration closure required for Ada packages.) Java comes with a significant set of library routines, and many software vendors have produced Java APIs to their products, where all of these services are described by an appropriate CFF. A CFF serves as both the 'object file' for the class and the 'library file' for separate compilation checking purposes.

Consider the situation where a software vendor provides a Java binding to his product, but provides no Ada binding. The vendor supplies appropriate Java

class files for his Java binding, but provides no source code. If we can generate an Ada binding from the Java class file, we can provide access to this product for Ada programmers. This paper describes how we can generate an Ada binding, given a Java class file, and shows the limitations of the Java class file definition for generating such bindings.

The JVM does not place any restrictions on the source language compiler that produces Java class files. Several other languages, in particular Ada, have compilers that target the JVM. [Taft], [Aonix], [GNAT-JVM]. Thus an Ada programmer can write code targeted to the JVM, but this is not particularly useful unless that programmer can gain access to existing Java libraries. To do this, we need to be able to develop an Ada interface to Java classes, and then develop bindings for the various Java classes of interest. This paper concentrates on the second step.

For this paper, we use the term "Java" to refer to the programming language, including class libraries implemented in that programming language. Thus we make a distinction between "Java the language" that is akin to "Ada the language", and the JVM (which does not imply/require that it execute programs compiled from "Java the language"). Note that our dependence on the JVM format, rather than Java source code, means that our technique will generate an Ada binding for any source language that can be compiled to produce Java class files. However, our focus is on class files where the source language was Java.

2 The Java Model

2.1 Java Virtual Machine

The Java Virtual Machine is an abstract stack-based computational machine with an instruction set generally implemented as an interpreter. (Additional speedups, called Just-In-Time compilers, which actually translate the instructions to concrete instructions on the execution target, are still run under the control of an interpreter.) A JVM interpreter then interprets the byte codes of the class files and updates its internal state according to the execution of the instructions.

2.2 Java Class Files

Java class files are organized collections of variable length records containing information about the class they contain. Those records have an indication of their length and further structured records within them.

At the outer most level a Java class file roughly has the structure:

```
file header
constant pool
class descriptor
flags
interfaces
fields
methods
attributes
```

The file header has identifying information and the Java version that produced it. The constant pool is similar to the symbol table in a traditional load module, but it is much more comprehensive since other structures within this class file reference it. Much more local information is available. It is in the constant pool that many strings used in the program are recorded: internal and external field, method, and class names; field and method signatures; and actual text strings used within the program.

The class descriptor has the name of the superclass, the name of this class, and any imported interfaces. These names are all stored by providing indexes into the constant pool. The flags provide access information to the class regarding the availability of the methods of the class to other entities (public, final, superclass, abstract, etc.)

The fields are the local variables of the class. They have various attributes including visibility, types, and values among other things. Static fields are shared across all instances of the class, while each class object has a separate copy of the instance (non-static) variables.

The methods have signatures and reference various attributes, such as executable byte code, exceptions that can be thrown, source file location and line number information, referenced out of the attributes section.

The attributes contain information like the source file name and other vendor specific information. This is an extensible section, but our work does not depend on any attributes not guaranteed to be present in all Java class files.

Because Java is dynamically linked, the class file must contain a lot of symbolic and typing information that would have been eliminated in more static systems. The information that enables the JVM to dynamically link classes is the information we use to generate Ada bindings.

3 Binding Technology

There are two problems to be solved in producing a binding for an API implemented in a given programming language. The first problem is the "pragma

interface problem", where the solution to this problem ensures that a program written in the source language (i.e., Ada) can access facilities defined by a program in the target language (i.e., C or Java). Solutions to this problem require compiler and linguistic support. Interfacing to C, for example, requires access to C functions and static objects. The details for this interface include topics such as producing values of an arbitrary C type, "by-value" vs "by-reference" parameter modes, passing C struct values (versus struct * values), and handling C functions that both return a value and modify their parameters. In all cases, one issue that must be resolved is the mapping of identifiers from the target language to Ada. (Problems can occur when the target language identifier is an Ada reserved word or when the identifier does not match Ada syntax, including case-sensitive identifiers.)

For Java, we have a similar, but expanded, set of interface issues. The list for Java includes: deriving (inheriting) from a Java class, implementing a Java interface, Java vs Ada exceptions, circular class references and synchronized methods. We are adopting the approach introduced in [GNAT-JVM] and more completely defined in [GNAT].

The second fundamental problem in binding is to develop a technique or pattern for mapping collections of target language features into the appropriate/best source language representation. Of course, the first requirement on the mapping is that it be feasible, using the solution to the "pragma interface problem" described earlier. The binding solution must take into consideration a somewhat different set of issues, including packaging and aggregation, error handling, binding-wide type models and visibility, and documentation.

Normally, bindings are programming language "syntactic transforms". They start with the expression of the API in the target languages (e.g., C header files or Java class files), and apply a set of transformations (either automated via a bindings-generator tool such as [c2ada], or manually, as in [POSIX].) The target language syntax (and semantics) defines the information that can be used by the binding generator.

Our approach differs from this "classical" approach, since we are not working at the target programming language (Java) syntax level. Instead, we are working with another representation of the target language, the Java Class File. Our approach to working with the Class File is to first define the "classical" Java-to-Ada syntactic transformations, i.e., a Java Class File maps to an Ada package. Then, we obtain the equivalent information from the Java Class File, and transform it into Ada. Thus, instead of parsing Java source code and walking the syntax tree, we instead parse and navigate the Java Class File, searching for specific Java syntactic constructs. The primary advantage of this approach is

that we do not require Java source code for the API. All we require is the Java Class Files for that API, which must be accessible for the Java Virtual Machine to use the API. This means that we can generate an Ada binding in situations where the source code is not available, such as reverse-engineering situations or cases where the source code is proprietary.

4 Java Interface Issues

The goal for the binding is to first, extract the 'binding entity' from the Java class file, and second, generate the appropriate Ada text. This section briefly summarizes the GNAT approach as described in [GNAT], with some additional decisions made for our binding.

4.1 Classes Map to Packages

We map each (non-nested) class to a separate package. The package name hierarchy maps well to Java's class heirarchy, i.e., the class java.lang.String maps to the package Java.Lang.String; there is a parent package called "Lang" that itself is a child package of a package named "Java". Nested classes are defined within the package that maps to the non-nested top-level class.

Each class that has instance variables or methods defines a tagged type. An Ada compiler that implements an interface to Java must define the root Object class, along with the rules for how other Java and Ada classes/types are derived from Object. Our binding follows the GNAT approach, and the type representing a class is a child of Java.Lang.Java_Object. See [GNAT] for a description of this approach.

Appropriate Ada95 pragmas (e.g., pragma Convention, pragma Interface, pragma Import) must be placed as defined by the Ada compiler.

4.2 Primitive Types and Variables

The mappings from the basic data types (section 4.3.2 of [Lindholm]) to Ada types are straightforward as described in [GNAT]. (Note that Java has no primitive Enumeration type.)

```
boolean is mapped into Ada's Boolean
char    is mapped into Ada's Wide_Character
byte    is mapped into Ada's Short_Short_Integer
short   is mapped into Ada's Short_Integer
int     is mapped into Ada's Integer
```

```
long    is mapped into Ada's Long_Integer
float   is mapped into Ada's Float
double  is mapped into Ada's Long_Float
```

Mapping for arrays of elementary types are nearly as straightforward.

Static instance variables map to objects within the class package. Non-static instance variables are fields of the record type defined in the class. Constructors are defined as functions. Java interfaces are implemented as record fields that point to the interface object.

4.3 Method Modifiers

Java defines the following set of method modifiers. These keywords define specialized semantics for class methods. By default, a method without any modifiers is public, not protected, not abstract, not static and not final.

```
public
protected
private
abstract
static
final
synchronized
native
```

Public methods are primitive operations in the visible part of the class's package, while protected and private methods are located in the private part of the class's package. Abstract methods match the Ada semantics. Final methods have no direct analog in Ada, and are not treated specially by the binding. Static methods do not have the implicit "self" parameter, and are therefore not necessarily primitive operations of the type defined by the package. Native methods are not currently supported by our binding, but we expect that a native method is no different, from a binding perspective, than any other method.

Synchronized methods require special treatment. Our binding tool marks a class having synchronized methods, and we expect that a type with synchronized methods will need to be based on some combination of tagged and protected types that need to be defined by the Ada compiler vendor.

4.4 Exceptions

Exceptions are defined in the [GNAT] approach as a straightforward binding, consisting of a package containing an object derived from the parent java.lang.Exception_Class.obj

with a null record extension, an Ada exception declaration, and the appropriate operations. The GNAT compiler then recognizes this set of declarations and performs the correct mapping to a Java exception.

5 The Binding Implementation

This section describes the implementation of the binding process.

5.1 Parsing the Java Class File

Parsing the Java Class File is rather a straightforward process. The abstract description given in [Lindholm] is sufficiently detailed to permit the direct generation of supporting record types and a recursive descent parser to navigate class files.

5.2 Generating the Package Template

The first task is to generate the package template. A pass through the methods for the class is performed to look for any method that has the synchronized modifier. Code generation for a class with synchronized methods must implement the underlying compiler's approach for dealing with such methods, i.e., including some sort of protected object in the object declaration. Mapping to the Ada package mechanism is straightforward as synchronized methods in Java and protected objects in Ada both commonly have underlying monitors to ensure sequential access.

5.3 Generating the Fields

Each of the non-static fields is placed into the package specification. As the list of fields is traversed, the field name and signature are obtained and mapped into a meaningful name for the Ada compiler.

5.4 Generating Simple Methods

For the simple (public) methods, the list of methods is traversed to find those public methods. Each method's name and parameter list are again mapped into meaningful names for the Ada compiler. All parameters are either mode "in" or an accesstype. Although the parameter's types are available in the underlying class file, the name of the parameters are not always available. This has lead to an instantiation of the parameter list by a simple positional enumeration of their location within the list.

6 Results / Analysis / Discussion

This paper has shown that we can go a long way toward automating the generation of Ada interface specifications for Java class files. We have applied our tool to both the JDK 1.1.4 release, comprising 1611 Java classes and the JDK 1.1.5 release with 1626 Java classes. At the time this paper was written, the tool completely parses all classes, and implements public and static methods and variables, generating the corresponding Ada construct.

One major problem is that the class file does not contain all of the information needed to generate a good binding. The primary limitation is the loss of formal parameter names. The representation of a method descriptor in the Java CFF includes only type information for each formal parameter. The parameter name itself is missing. As an extension to the CFF, it is possible to recompile the Java source using the JDK's "-g" flag. This will retain parameter information for some methods (those that are not abstract) [Taft2]. Unfortunately, we cannot guarantee that any arbitrary Java Class File was generated by the JDK compiler using the -g flag.

In our view, this makes the resulting binding unusable. Consider the following methods from the class Java.Lang.String:

```
public boolean regionMatches (boolean ignoreCase,
        int toffset,
        String other,
        int oofset,
        int len);

public boolean regionMatches (int toffset,
        String other,
        int oofset,
        int len);
```

Here is our generated Ada binding:

```
function regionMatches (this: Java.Lang.String;
        arg_1 : Interfaces.boolean;
        arg_2 : integer;
        arg_3 : Java.Lang.String;
        arg_4 : integer;
        arg_5 : integer) return boolean;

function regionMatches (this: Java.Lang.String;
```

```
        arg_1 : integer;
        arg_2 : Java.Lang.string;
        arg_3 : integer;
        arg_4 : integer)  return boolean;
```

For comparison, here is the same method, from the Intermetrics/Aonix binding:

```
function Region_Matches (This : access String_Obj;
        Toffset : Integer;
        Other : String_Ptr;
        Ooffset : Integer;
        Len : Integer) return Boolean;

function Region_Matches (This : access String_Obj;
        IgnoreCase : Boolean;
        Toffset : Integer;
        Other : String_Ptr;
        Ooffset : Integer;
        Len : Integer) return Boolean;

function RegionMatches (This : access String_Obj;
        Toffset : Integer;
        Other : String_Ptr;
        Ooffset : Integer;
        Len : Integer) return Boolean renames Region_Matches;

function RegionMatches (This : access String_Obj;
        IgnoreCase : Boolean;
        Toffset : Integer;
        Other : String_Ptr;
        Ooffset : Integer;
        Len : Integer) return Boolean renames Region_Matches;
```

Note that the Intermetrics/Aonix binding uses a renames clause that maps the "run-on" Java name to a more Ada-Like name using Underscores. Additionally, since Java is case sensitive and Ada is not, it is necessary to find a mechanism to distinguish between, for example, mappings of AphID and aphid.

The class String points out several other interesting aspects of Java binding. As mentioned in [GNAT-JVM] the class String refers to the class Object, and vice versa. (String extends Object, while Object contains an operation (toString) that returns a value of the class String.) This circular reference affects both the

contents of each individual class, as well as the overall approach to packaging. The Intermetrics/Aonix binding removes this circularity by manual analysis, while the GNAT proposal allows a special pragma that will permit a "forward reference" from one class to another.

Our tool depends on the ability to provide the GNAT-style forward references (see [GNAT] and [GNAT-JVM]. This simplifies the package structure, and the "withing". Each class maps to a separate package, and the context clause for that package is constructed from the "extends" and "implements" part of the Java class declaration. For example, consider the following extract from the DateFormat class:

```
public abstract class java.text.DateFormat extends java.text.Format
        implements java.lang.Cloneable {
                ...
        protected Calendar calendar;
        ...
}
```

The generated Ada binding for this class starts out:

```
with Java.Lang.Object; with Java.Lang.Cloneable;
package Java.Text.DateFormat is
        ...

pragma Import_Access (java.lang.Calendar.ptr);

        type DateFormat is new Java.Lang.Object with record
                Calendar : Calendar.ptr;
                ...
end Java.Text.DateFormat;
```

This approach avoids the problem of determining the circular references, and the even more difficult problem of hand-manipulating the package structure to eliminate such circular references.

The algorithms described in this paper were implemented using Ada 95, consisted of approximately five thousand lines (including blanks and comments) and were ported to a variety of operating platforms (Linux, Solaris and SunOS).

7 Summary and Conclusions

We have demonstrated how an Ada binding can be automatically generated from the information contained in the Java Virtual Machine (JVM) Class File For-

mat (CFF). The resulting binding is functionally complete, but has substantial limitations in usability. The primary limitation is that the JVM class file does not retain the names of formal parameters.

It would be possible to provide formal parameter names through an extension to the JVM CFF. The definition of the CFF allows for implementation-defined attributes, and formal parameter names could be an implementation-defined attribute for methods of a class. But this would not meet our goals of using only the portable aspects of the CFF. A longer-range alternative is for the JVM CFF to define such an attribute. This information would be very useful for a Java debugger, so it is possible, once the issues surrounding Java standardization are settled, that the JVM CFF specification is updated to include such attributes.

The Java language also contributes to usability restrictions on the part of an Ada binding. The Java language permits two classes to be mutually defined. Thus class files do not exhibit the same degree of ordering (they do not form a lattice) as is required by Ada packages, or C/C++ "include" files. Since the Java Language does not provide for user-defined scalar types, nor does it provide an enumeration type, there is no information in the JVM CFF to identify scalar values other than using the predefined Java types such as Integer.

Some of the limitations of the JVM CFF could be mitigated by using Java source code, rather than JVM CFF, as the input to a binding generator tool. This approach requires source code to the Java class, which may not always be available or convenient. Another alternative would be to "revert" to manually developing a more user-friendly binding, that builds on the machine-generated binding derived from the JVM CFF. This approach is time-consuming and there is no guarantee that two bindings authors will generate the same Ada binding from the given Java source. But the human analyst may be able to synthesize information that is not directly available from the Java source or the CFF, such as scalar type information or an appropriate "withing structure".

The most significant restriction with our approach is the loss of formal parameter names. This could be easily rectified through the addition of appropriate attribute values to the JVM CFF. With the addition of formal parameter names, machine-generated Ada bindings to Java class files would not be perfect, but should prove to be usable for Ada developers. Without this information, bindings generated from the Java CFF should be considered only when the Java source code is unavailable.

Another opportunity, as of yet uninvestigated, is to look at the possibility of mapping to a more object oriented binding directly utilizing tagged types in Ada rather than having everything be a subclass of Object (as is done in Java). This might provide additional benefit to the Ada programmer, albeit perhaps at

a cost to the bindings implementer.

8 Acknowledgements

The authors would like to acknowledge the contributions of Tucker Taft for the Ada to Java idea and responding to various email questions, the GNAT team for information on their binding approach and Olimpia Velez for last-minute editing and corrections.

References

[AdaBindings] Ada 95 Bindings Report, DASW01-94-C-0054, Task Order T-S5-306, Defense Information Systems Agency Center for Software, 15 August 1995.

[Aonix] "Read me" file for Aonix Ada 95 to JDK 1.1 binding, available from ftp://ftp.aonix.com/pub/web/ada/jdk_1.1.zip

[AppletWriter] Applet Writer's User Guide, available from the AppletMagic home page, http://www.intermetrics.com/appletmagic/download/appletwriters.guide.txt.

[c2ada] Automated C to Ada binding tool. Available from http://www.inmet.com/ mg/c2ada.

[Emery] Emery, David and Nyberg, Karl; "Observations on Portable Ada Systems", in Ada: the design choice, Proceedings of the Ada-Europe International Conference, Madrid, Spain, 13-15 June 1989. Cambridge University Press. Also available as MITRE Technical Paper MTP-282, February, 1989, Bedford, MA.

[GNAT] Java to Ada Interfacing, Appendix B of GNAT Ada mapping to JAVA, Ada Core Technologies, in preparation, private communication.

[GNAT-JVM] Comar, Cyrille; Dismukes, Gary and Gasperoni, Franco; "Targeting GNAT to the Java Virtual Machine", in Proceedings of Tri-Ada 1997, ACM SIGAda, St. Louis, 1997.

[Harold] Harold, Elliotte Rusty Java Secrets, IDG Books Worldwide, Foster City, CA, 1997.

[Intermetrics] Release notes for release 2.0.1 of AppletMagic, available from the AppletMagic home page, http://www.intermetrics.com/appletmagic/api/index.html

[JNI] The Java Native Interface - available at URL http://www.javasoft.com/products/jdk/1.1/docs/guide/jni/spec

[Kramer] Kramer, Douglas; The Java Platform, available at http://java.sun.com/docs/white/platform/.

[Lindholm] Lindholm, Tim and Yellin, Frank; The Java Virtual Machine Specification, Addison-Wesley, Reading, MA, 1997.

[Meyer] Meyer, Jon and Downing, Troy; Java Virtual Machine, O'Reilly & Associates, Sebastopol, CA, 1997.

[POSIX] IEEE Standard IEEE STD 1003.5-1992, POSIX System Interfaces Ada Binding, IEEE, Piscataway, NJ, 1992.

[Taft] Taft, S. Tucker, "Programming the Internet in Ada 95", in Proceedings of the Ada-Europe International Conference, 1996. Cambridge University Press; also available as: http://www.inmet.com/štt/adajava_paper/.

[Taft2] Taft, S. Tucker, private communication.

[Venners] Venners, Bill, Inside the Java Virtual Machine McGraw-Hill, New York, 1998.

[Wragg] Wragg, David; Drossopoulou, Sophia and Eisenbach, Susan; "Java Binary Compatibility is Almost Correct", available as http://outoften.doc.ic.ac.uk/projects/slurp/papers.html#bincomp

This article was processed using the LaTeX macro package with LLNCS style