LEVERAGING EXISTING PLASMA SIMULATION CODES

Anna Malinova, Vasil Yordanov, Jan van Dijk

Abstract: This paper describes the process of wrapping existing scientific codes in the domain of plasma physics simulations through the use of the Sun's Java Native Interface. We have created a Java front-end for a particular functionality, offered by legacy native libraries, in order to achieve reusability and interoperability without having to rewrite these libraries. The technique, introduced in this paper, includes two approaches – the one-to-one mapping for wrapping a number of native functions, and using peer classes for wrapping native data structures.

Keywords: wrapping, legacy code, Java Native Interface, peer classes.

ACM Classification Keywords: D.2.13 Reusable Software

Conference: The paper is selected from Sixth International Conference on Information Research and Applications – i.Tech 2008, Varna, Bulgaria, June-July 2008

Introduction

Contemporary physics simulations evolve over the years and become composite and increasingly complex. A large portion of the system may consist of legacy codes, which are mostly written in languages like FORTRAN, C, and C++. It is ineffective and unreliable to rewrite the entire software system with new design rules, or in new programming languages. Reusing is perhaps the best strategy to handle complexities in software development.

We use the Sun's Java Native Interface (JNI) to leverage existing native libraries. By native code, we mean non-Java code, typically C or C++. As part of the of the Java virtual machine implementation, the JNI is a two-way interface that allows Java applications to invoke native code and vice versa [JNI, 2003]. Thus JNI allows programmers to take advantage of the power of the Java platform, without having to abandon their investments in legacy scientific codes [Malinova, 2006].

In this paper we present creating Java front-ends for some basic functionality of the PLASIMO simulation software. PLASIMO is a framework for modeling low-temperature plasma sources. It has been developed at Eindhoven University of Technology in the department of Applied Physics [Plasimo, 2008]. PLASIMO is written in C++, so in our work we have used the JNI to produce a class library that wraps a set of the PLASIMO's functions and classes. We first discuss the most straightforward way to write wrapper classes – the *one-to-one* mapping. We then introduce how we wrap native data structures using *peer classes*. At the end, we discuss such issues like exception handling and reflection support provided by the JNI.

Methodology

Java applications call native methods in the same way they call methods implemented in the Java programming language. Behind the scenes, however, native methods are implemented in another language and reside in native libraries. We use the JNI to write native methods that allow the Java code to call functions implemented in PLASIMO's native libraries. The JNI allows interaction to occur in both directions: the Java code can invoke native methods; as well the native methods can create, update and inspect Java objects and call their methods (see Figure 1).

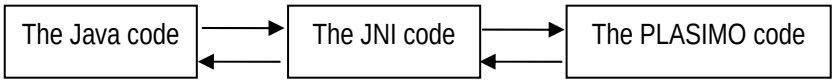

Figure 1. The Java Native Interface as link between Java and PLASIMO

Implementation of a call from Java to a PLASIMO library includes several basic steps, such as [JNI, 2003]:

- Define a Java wrapper class with the native method declaration. This declaration includes the keyword `native` to signify to the Java compiler that it will be implemented externally. The created Java wrapper loads the native library that contains the native code, and invokes the native methods.
- Generate a header file for use by the native (C/C++) code.
- Create the native C/C++ implementation - the native code implements the function definition contained in the generated header file and implements the needed logic as well.
- Compile the native implementation into a native library – create dynamic (.dll) or shared object library (.so) so it can be loaded at runtime.

One-to-one mapping

This section discusses the most straightforward way to write wrapper classes – the one-to-one mapping. This approach requires us to write one stub function for each native function that we want to wrap. Figure 2 presents the one-to-one mapping of the functions belonging to the PLASIMO's `plRuntime` namespace - it contains functions and classes for runtime configuration and library loading/unloading.

```
public class JRuntime {
        private static native void _configure(JNode node);

        public static void configure(JNode node) {
                _configure(node);
                .........
        }
        .........
}
```

a) native function declaration

```
JNIEXPORT void JNICALL Java_JRuntime__1configure
    (JNIEnv *env, jclass clsptr, jobject jnode) {
        ...........
        // invoke plRuntime::Configure
        ...........
}
```

b) native stub function

```
namespace plRuntime
{
        ...........
    void Configure( const plNode & node) {....}
}
```

c) native method definition

Figure 2. One-to-one mapping approach to write wrapper classes.

Each native function (for example, _configure) maps to a single native *stub* function (for example, `Java_JRuntime__1configure`), which in turn maps to a single native method definition (for example, `plRuntime::Configure`). Here the stub function serves two purposes:

- The stub adapts the native function's argument passing convention to what is expected by the Java virtual machine. The virtual machine expects the native method implementation to follow a given naming convention and to accept two additional arguments. The first parameter, the `JNIEnv` interface pointer, points to a location that contains a pointer to a function table. Each entry in the function table points to a JNI function. Native methods always access data structures in the Java virtual machine through one of the JNI functions. The second argument differs depending on whether the native method is a static or an instance method. The second argument to an instance native method is a reference to the object on which the method is invoked, similar to the "this" pointer in C++. The second argument to a static native method is a reference to the class in which the method is defined. Our example, `Java_JRuntime__1configure`, implements a static native method. Thus the `jclass` parameter is a reference to the `JRuntime` class.
- The stub converts between Java programming language types and native types. The JNI defines a set of C and C++ types that correspond to types in the Java programming language. The mapping of primitive types is straightforward. For example, the Java type `int` maps to the C/C++ type `jint` (defined in `jni.h`). The JNI passes objects to native methods as opaque references. Hence, the native code must manipulate the underlying objects via the appropriate JNI functions.

The result is that the created stub function makes calls to the PLASIMO functions and possibly back to the Java methods, and returns results to Java.

Peer classes

One–to–one mapping addresses the problem of wrapping native functions. The examples in the previous section have covered calling standalone C++ functions that return result or modify parameters passed into the function. However, if we create an instance of a C++ class in one native method, another problem encounters: how can C++ classes be used from a Java program and keep objects around while the program is running? One way to handle this situation is to define a Java class called "peer class" that corresponds to the C++ class. Peer classes are classes that directly correspond to native data structures. Each instance of the peer class corresponds to a C++ object, tracking the state of the C++ object.

We have created a number of peer classes that correspond to some basic PLASIMO classes (see Figure 3). The native methods are called within the peer classes, and are the link between the peer classes and the C++ classes.

The JPeer class is an abstract Java class that all peer classes extend, as presented in Figures 4 and Figure 5. The JPeer class provides some common functionality and contains a 64-bit field that refers to the corresponding C++ instance of a PLASIMO class (see Figure 4). Subclasses of JPeer assign specific meaning to that field. If we are on a platform with 32-bit pointers, we can simply store this pointer in an int; if we are on a platform that uses 64-bit pointers, we store it in a long.

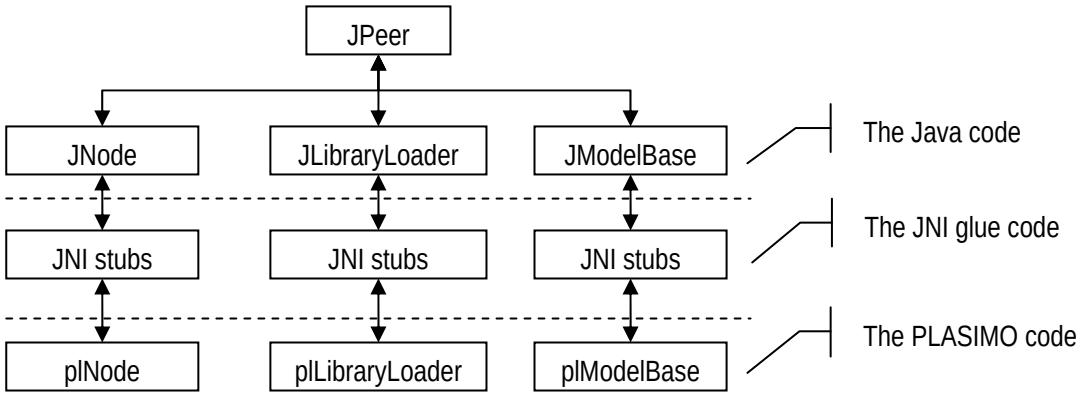

Figure 3. The Java peer classes corresponding to the C++ PLASIMO classes.

The Java peer classes will have the same methods as the C++ classes they represent (it is not necessary all the methods to be wrapped), but the implementation of these methods will be to call the C++ equivalents. For example, take the JNode peer class that wraps the native plNode class (see Figure 5). The Plsimo's plNode is a tree with a variable branching factor. It consists of two parts: a data part of type plLeaf and a variable-sized vector of pointers to children, also of type plNode. Two kinds of plNode-instances occur, sections and data lines: nodes without children are data lines; nodes with one or more children are sections. The children are also plNode's instances, also these can be either sections or data lines. plNode offers a variety of members for accessing its data items, or for section nodes, those of its children.

```
abstract class JPeer {
  private long pointer;

  public long getPointer() {
        return pointer;
  }
  public void setPointer(long ptr) {
        pointer = ptr;
  }

  public abstract void destroy();

  public void finalize(){
        destroy();
  }
}
```

Figure 4. The abstract peer class.

As can be seen in Figure 5, the JNode's constructor calls the native method `create` passing as arguments string values representing a section name and a file name. Then inside the stub function, implementing the JNode's `create` method, an instance of the C++ plNode class is created (see Figure 6).

```
public class JNode extends JPeer{
   private native long create(String secname);
   private native long create(String secname, String filename);
   private native long create(String secname, BufferedReader reader);
   private native void destroy(long p);
   private native void write(String filename, boolean recreate, long p);
   private native void write(PrintWriter writer, boolean recreate, long p);
   private native void read (String filename, long p);
   private native long mount (long tree, long p);
   private native long getSection(String secname, long p);
   .....
   public JNode(String secname, String filename) throws JException {
        long jnodePtr = create(secname, filename);
        setPointer(jnodePtr);
   }
   public void destroy() {
        long jnodePtr = getPointer();
        if(jnodePtr != 0) {
             destroy(jnodePtr);
             setPointer(0);
        }
   }
   .....
}
```

Figure 5. The JNode peer class.

The pointer to that instance is saved in the 64-bit field defined in the peer class JNode. In other words, the `create` native method returns this value, and the value is saved in objects of the peer class. When other native methods, such as `destroy`, `write`, etc., are called, this value is retrieved and passed as an argument to the method. The value is then casted into a C++ pointer.

```
JNIEXPORT jlong JNICALL Java_JNode_create__Ljava_lang_String_2Ljava_lang_String_2
  (JNIEnv * env, jobject obj, jstring secname, jstring filename) {
    const char * str;
    str = env->GetStringUTFChars(secname, NULL);
    if(str == NULL) { return 0; }
    const char * c_filename = env->GetStringUTFChars(filename, NULL);
    if(c_filename == NULL) {
        return 0;
    }
    plNode *ptr=0;
    try{
        ptr = new plNode(str, c_filename);
        return (jlong)ptr;
    }
    catch(plParserException& plex){
        std::string msg = "Plasimo exception: " + (std::string)plex.what();
        JNI_throwException(env, "JException", msg);
        return 0;
    }
    .....
}
```

Figure 6. The stub function implementing a JNode's `create` method.

An important point about peer classes concerns freeing native data structures. Instances of the Java peer classes are garbage collected but not the instances of the C++ classes. C++ has no garbage collection, so it is necessary to think about how objects are destroyed when they are no longer in use. When objects are dynamically created with the `new` operator, the `delete` operator must be explicitly called. In the peer class, this behavior is modeled using the `destroy` method. It is declared as abstract method of the JPeer class. All peer classes that extend the abstract JPeer class provide implementations of the `destroy` method (see Figure 5).

JPeer also defines a `finalize` method that calls `destroy`. When the garbage collector is ready to release the storage used for a Java object, it will first call `finalize` and clean up the memory allocated inside the non-Java code. Since neither garbage collection nor finalization is guaranteed, one cannot rely on `destroy` being called in a timely way. If the Java virtual machine is not close to running out of memory, then it might not waste time recovering memory through garbage collection. That is why we explicitly call `destroy` to invoke the destructor for the C++ class, and avoid memory leaks.

Use the reflection support to call Java from C++

In the previous examples we have discussed calling C++ from the Java code: how can C++ classes be used from a Java program, and keep objects around while the program is running. But since JNI is a two-way interface, we can also call Java from within the native code. JNI allows accessing Java class fields, calling methods, invoking constructors. This involves using the JNI functions that provide reflection support since the reflection allows us to discover at run time the name of arbitrary class objects and the set of fields and methods defined in the class. Although it is possible to call the corresponding Java API to carry out reflective operations, the JNI provides functions to make the frequent reflective operations from native code more efficient, such as: `GetSuperClass`, `GetObjectClass`, `IsInstanceOf`, etc.

In Figure 7 it is shown how to call the JNode's method `getPointer` from a native stub function. Calling a Java method (instance or static) from within the native code involves the following three steps: retrieve the class

reference; retrieve the method identifier; call the method. The `getPointer` method is an instance method defined in JPeer – the JNode's superclass, so we have to obtain the method identifier from a reference to the superclass.

```
JNIEXPORT void JNICALL Java_JRuntime__1configure
    (JNIEnv *env, jclass clsptr, jobject jnode) {

    jclass cls = env->GetObjectClass(jnode);
    jclass super = env->GetSuperclass(cls);
    jmethodID mid = env->GetMethodID(super, "getPointer", "()J");
    if (mid == NULL) {
        return;
    }

    jlong jl = env->CallNonvirtualLongMethod(jnode, super, mid);
    .....
}
```

Figure 7. Use the JNI functions providing reflection support

Exception handling

When the native code detects the exception thrown by the PLASIMO code, it throws a Java exception - in our case an instance of the JException class, as it is shown in Figure 8. This can be seen also in Figure 6.

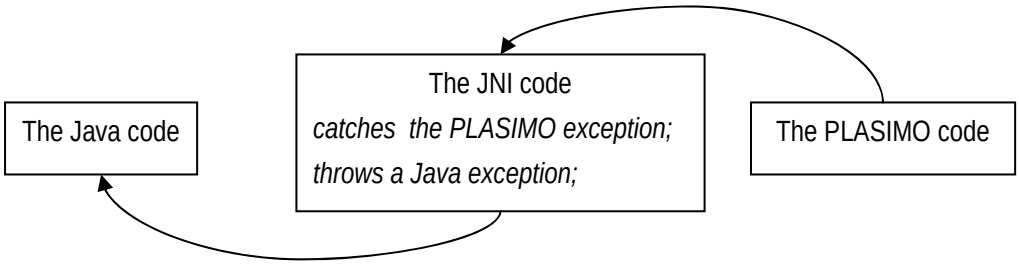

Figure 8. Handling PLASIMO exceptions inside the JNI stub functions.

More interesting is the case when the native code issues a callback to a Java method that itself throws an exception. When the control is returned to the native method, the native code can detect this exception by calling the `ExceptionOccurred` function – the JNI function that performs checks for a pending exception in the current thread. Then the native code can clear the exception by calling `ExceptionClear` and then execute its own exception handling code.

Conclusions

The technique of using the Java Native Interface allows us to leverage code in existing native libraries, particularly the PLASIMO simulation software. Once we have Java-based components to work with, they can easily be transformed into Web services [Mahmoud, 2005]. External client applications will interact with these Web service wrappers, rather than the actual scientific codes. This will alleviate the scientific collaboration with other development teams and will allow us to build connected applications.

Apart from avoiding rewriting PLASIMO in Java, using the existing PLASIMO libraries trough JNI has another advantage. Java has been designed to be portable – compiled programs run on any machine with a Java Virtual

Machine. To accomplish this, the Java compiler compiles to machine-independent byte code that is interpreted at run time by the virtual machine. Although the Java interpreter is efficient, the plasma simulations are supposed to involve cpu-intensive operations on large amounts of data and by leaving these operations out of Java into the PLASIMO compiled libraries, we may expect better performance.

Acknowledgments

This work is supported by the NSF of the Bulgarian Ministry of Education and Science, Project VU-MI-205/2006 and the NPD-07M07 project of the University of Plovdiv "Paisii Hilendarski", Bulgaria.

Bibliography

[Agrawal, 2006] S. Agrawal, V. Chenthamaraksan. Handling events from native objects in Java code, 2006, http://www-128.ibm.com/developerworks/java/library/j-jniobs

[JNI, 2003] Java Native Interface 5.0 Specification (2003). http://java.sun.com/j2se/1.5.0/docs/guide/jni/index.html

[Malinova, 2006] A. Malinova, S. Gocheva-Ilieva, I. Iliev. Wrapping legacy codes for Numerical simulation applications, Proceedings of the III International Bulgarian-Turkish Conference Computer Science, Istanbul, Turkey, October 12-15, Part II, pp. 202-207, 2006.

[Mahmoud, 2005] Q. Mahmoud. Service-Oriented Architecture (SOA) and Web Services: The Road to Enterprise Application Integration (EAI), 2005, http://java.sun.com/developer/technivcalArticles/WebServices/soa/index.html

[Plasimo, 2008] PLASIMO simulation software, http://plasimo.phys.tue.nl .

[Pont, 2004] M. Pont. Calling NAG Library Routines from Java. NAG Technical Report TR 1/04, 2004.

[Pont, 2003] M. Pont. Calling C library routines from Java, Dr.Dobb's Journal, 2003, http://www.ddj.com/architect/184405381

[Smith, 2003] J.Smith. Fast math with JNI, JavaWorld.com, 2003.

[Stearns, 2003] B. Stearns, Java Tutorial: Java Native Interface. Sun Microsystems Inc., 2003. http://java.sun.com/docs/books/tutorial/native1.1 .

Authors' Information

Anna Malinova – Department of Computer Technologies, University of Plovdiv "Paisii Hilendarski", 24 Tzar Assen St., Plovdiv-4000, Bulgaria; e-mail: malinova@uni-plovdiv.bg

Vasil Yordanov – Faculty of Physics, Sofia University, BG-1164 Sofia, Bulgaria; e-mail: v_yordanov78@yahoo.com

Jan van Dijk – Department of Applied Physics, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands; email: j.v.dijk@tue.nl