

Wrapping Legacy Codes for Grid-Based Applications

Yan Huang, Ian Taylor, and David W. Walker
Department of Computer Science
Cardiff University, PO Box 916
Cardiff CF24 3XF, UK

Robert Davies
Department of Physics and Astronomy
Cardiff University, PO Box 913
Cardiff CF24 3YB, UK

Abstract

This paper describes a process for the semi-automatic conversion of numerical and scientific routines written in the C programming language into Triana-based computational services that can be used within a distributed service-oriented architecture such as that being adopted for Grid computing. This process involves two separate but related tools, JACAW and MEDLI. JACAW is a wrapper tool based on the Java Native Interface (JNI) that can automatically generate the Java interface and related files for any C routine, or library of C routines. The MEDLI tool can then be used to assist the user in describing the mapping between the Triana and C data types involved in calling a particular routine. In this paper we describe both JACAW and MEDLI, and demonstrate how they are used in practice to convert legacy code into Grid services.

1. Introduction

This paper describes a wrapping and data mapping technique for converting existing “legacy” code, particularly libraries of scientific and mathematical software written in the C language, for use as composable computational services within a Grid environment based around Triana. Triana is a visual programming environment that allows users to create applications from code components supplied with the system by dragging components, dropping them into a workspace, and connecting them together to build a workflow graph [9]. Triana was initially developed by scientists in GEO600¹, a major gravitational wave detection experiment, to help in the flexible analysis of massive data sets.

Java’s object-oriented features, platform independence, and numerous APIs for tasks such as network programming, XML processing, and GUI building, make it a powerful and increasingly popular language for developing Grid-based e-Science applications. However, the task of manually con-

verting the large body of existing high quality, validated code to Grid-enabled, Java-based services is both daunting and expensive. The main aim of the work described in this paper is to make this conversion task as automated as possible in order to facilitate the use of such services in composing Grid-based scientific applications.

The wrapping and data mapping functions are carried out by two separate tools. The Java-C Automatic Wrapper (JACAW) tool automatically wraps C routines as Java code using the Java Native Interface (JNI). This approach can be applied to individual C routines, or to whole libraries, and is based on the routine interfaces given in the C header files. JACAW allows users with no knowledge of JNI to quickly and easily build a Java wrapper for C routines. The approach taken is similar to that of Getov and co-workers in the Java-C Interface (JCI) project [4, 6]. However, JACAW extends the JCI functionality to deal with arrays of structures, and provides a graphical user interface through which users can wrap legacy C libraries.

The MEDIation of Data and Legacy code Interface (MEDLI) tool permits third-party Java software to be used in Java environments. In the case of Triana, MEDLI wraps code so it conforms to the Triana component model and uses Triana data types for its input and output. MEDLI can be applied to the wrapped C routines produced by JACAW, and so also allows C legacy code to be accessed as native Triana components.

This paper is organised as follows. In Section 2, a brief overview is given of other approaches to addressing the problem of integrating legacy code into a Grid environment. Section 3 discusses the main features of JACAW and describes the main features of its implementation. MEDLI is discussed in Section 4. An example of the use of JACAW and MEDLI within a Triana-based Grid environment is given in Section 5. Some concluding remarks are presented in Section 6.

¹See <http://www.geo600.uni-hanover.de/>.

2. Related Approaches to the Legacy Code Problem

Two main approaches have been adopted in applying Java to numerical computing. In the first approach, scientific packages previously written in C, C++, or FORTRAN are completely rewritten in Java. Examples include commercial packages such as JNL from Visual Numerics², and packages from research projects such as JAMA³. Work on the Numerically Intensive Java (NINJA) project [7] supports the view that there are no fundamental technical reasons why Java should not be used for high performance numerical computing. NINJA uses language and compiler techniques to address Java performance problems, and this type of approach is essential if Java is to be adopted throughout the high performance computing community.

In the second approach, legacy packages are retained and JNI [5] is used to integrate native methods (e.g., C code) with Java. This may not always be an optimal or elegant solution, but it is necessary when large scientific libraries are not immediately available in Java. Wrapping legacy C code can also result in better performance than pure Java code, though this benefit will presumably diminish as a variety of compiler and runtime techniques continue to close the performance gap between C and Java code. One example of the wrapping approach is a Java interface to MPI [1, 6]. The Janet (JAvA NAtive EXTEnSiOnS) project makes use of Java language extensions and a preprocessing tool to develop Java interfaces to native code by automatically generating JNI code [3]. This approach requires the source code, but this has the advantage of allowing a high degree of control over the low-level behaviour of the native code. The Jaguar project [10] avoids the use of JNI in accessing native code by extending the Java runtime environment to enable direct Java access to operating system and hardware resources. This avoids the need to copy data between the Java and native code, and leads to efficient code, but the approach is architecture specific. SWIG [2] connects programs written in C (and C++) with a variety of high-level programming languages (including Java). It processes an interface file that defines all of the variables and functions that need to be accessed from Java (or any other language), and generates the JNI interface to the C code for Java. However, SWIG is not completely automatic (the interface file needs to be written) and furthermore, in some cases sufficient knowledge of variables and functions may not be possible without the source code (which is often not available).

²Java Numerical Library: see <http://www.vni.com/products/wpd/jnl>.

³JAvA MAtRiX package: see <http://math.nist.gov/javanumerics/jama/>.

3. An Introduction to JACAW

JACAW can be used to wrap existing legacy code in C as a Java code that calls the original code through the Java Native Interface. JACAW can be applied automatically to wrap entire software libraries thereby saving time and substantially reducing the likelihood of introducing coding errors. JACAW is based on the Java Native Interface (JNI) which is an API that allows Java code to interact with code written in another language. JACAW shields the user from the details of JNI, and does not require the user to have any knowledge of how to use it. JACAW takes the C header files as input and automatically creates the corresponding Java and C files needed to make native calls. JACAW also automatically compiles the Java files, creates the header files, and builds a shared library of the JNI-enabled C routines. The user can initiate and control all these actions through a simple GUI, although when used in conjunction with MEDLI this GUI is essentially replaced by the MEDLI interface. Details of the JACAW GUI, its support for multidimensional arrays and C structures, and performance issues are discussed in a separate paper.

As an example of what JACAW does, suppose the C header file `util.h` contains only two routines, the first of which generates a pseudo-random number, and the second sorts an array of integers:

```
signed int rand();
void bubblesort(int, int[]);
```

JACAW builds a class for each routine making it clearer and more convenient for the user to call the routine from Java. In the second routine there are two inputs, and no return value — the second parameter `int[]` provides both the input (the array to be sorted) and output (the sorted array) for the routine. The Java class files generated automatically by JACAW for these two routines are called `utilJ_randM.java` and `utilJ_bubblesortM.java`. The class name is constructed from the name of the C header file that defines the interface of the method and the method name. The capital 'M' appended to the method name indicates that this class is working as a method. This distinguishes it from a class that is working for struct ('S'), enum('E') and union('U') types defined in the native code. The code generated by JACAW for the `utilJ_bubblesortM.java` class is shown in Fig. 1.

The instance variables of this class are `arg0`, `arg1`, and `needCopy`, where `argn` corresponds to the `n`th parameter of the `bubblesort()` routine. The Boolean variable `needCopy` gives the user the option of copying back the array that is input to the native method when the method completes. The default value of `needCopy` is "true", but if the array does not need to be accessed in the Java code before it is next passed to a C routine then setting `needCopy` to "false" can reduce

```

public class utilJ_bubblesortM{
    boolean needCopy = true;
    int arg0;
    int[] arg1;

    public utilJ_bubblesortM(int arg0, int[] arg1){
        this.arg0 = arg0;
        this.arg1 = arg1;
    }
    public void nativeCall(){
        utilJ.bubblesort(this.arg0, this.arg1,
            this.needCopy);
    }
    public int getArg0(){
        return this.arg0;
    }
    public int[] getArg1(){
        return this.arg1;
    }
    public void setNeedCopy(Booleam needCopy){
        this.needCopy = needCopy;
    }
}

```

Figure 1. utilJ_bubblesortM.java: Java interface for the native routine bubblesort() generated by JACAW.

the overhead in some cases by preventing JNI from copying the value back to Java when it releases the reference to the array. This does not apply to the bubblesort() code in Fig. 1. Upon return from the bubblesort() native method the user can retrieve the sorted array by using the getArg1() method.

JACAW generates the Java class utilJ.java that loads and links the native methods declared in the header file util.h (see Fig. 2). JACAW also generates the C functions that implement the native methods and stores them in the file utilJCall.c (see Fig. 3). On compilation these are placed in the “util” library referred to in Fig. 2.

```

public class utilJ{
    public native static int rand();
    public native static void bubblesort(
        int arg0, int[] arg1, boolean needCopy);
    static {
        System.loadLibrary("util");
    }
}

```

Figure 2. utilJ.java: The class that loads and links the native methods.

The file utilJCall.c makes clear the role of the needCopy argument that is passed to the native method — the value of needCopy determines the release mode of the primitive array. If needCopy is true, then the mode is set to 0 and JNI has to copy back the contents of the array and free the

```

#include <jni.h>
#include "util.h"
JNIEXPORT jint JNICALL
Java_utilJ_rand(JNIEnv * env, jclass cls) {
    /* This is a native call for
       method: signed int rand() */
    signed int ret;
    ret = rand();
    return ret;
}
JNIEXPORT void JNICALL
Java_utilJ_bubblesort(
    JNIEnv * env, jclass cls, jint arg0,
    jintArray arg1, jboolean needCopy) {
    /* This is a native call for
       method: void bubblesort(int, int [])*/
    jint* arg1P;
    int mode;
    arg1P = (*env)->GetIntArrayElements(
        env, arg1, NULL);
    bubblesort(arg0, arg1P);
    if (needCopy == JNI_TRUE) mode = 0;
    else mode = JNI_ABORT;
    (*env)->ReleaseIntArrayElements(
        env, arg1, arg1P, mode);
}

```

Figure 3. utilJCall.c: This code implements the native methods.

array buffer on exit. Otherwise, the mode will be set to JNI_ABORT that will free the buffer without copying back the possibly modified array. Figure 4 summarises the functions of JACAW and the files it generates.

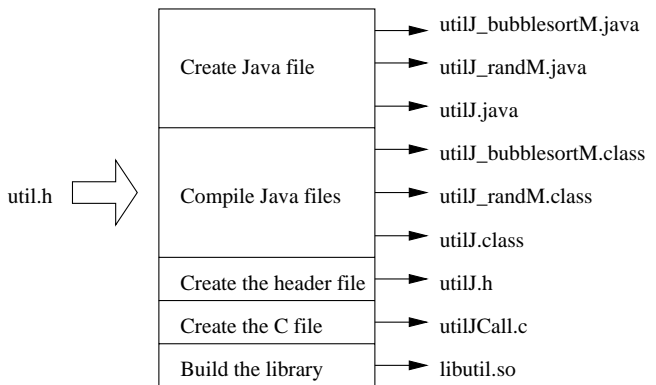


Figure 4. The inputs and outputs of JACAW

Once the library and the Java stub have been created the native routine can be called from Java. The Java interface created by JACAW is similar to that of original C code. To call a C routine, such as rand() or bubblesort(), from Java the user needs to do the following:

1. Initiate the class corresponding to the wrapped routine.

2. Set the needCopy variable, if necessary, using the setNeedCopy() method. The default value is true so this step can usually be omitted.
3. Make the native call.
4. Get the return value, if any, using the getReturnValue() method.
5. Retrieve any arguments that may have been modified by the call.

An example of the code a programmer might write in order to use the wrapped rand() and bubblesort() routines generated by JACAW is shown in Fig. 5. In this example lines 1-7 create and initialise an array of 100 random integers, and lines 8-10 sort the array. Line 3 initialises the class utilJ_randM. The constructor in this case has no arguments because the original C routine, rand(), has none. Within the for loop, in lines 5 and 6, the wrapped version of rand() is called through the nativeCall() method, and the getReturnValue() method is then used to retrieve the random integer generated, which is then stored in the ivalues array. Line 8 initialises the class utilJ_bubblesortM and it should be noted that the arguments passed to the constructor are simply the arguments of the original bubblesort() routine. The wrapped version of the bubblesort() routine is called in line 9, and in line 10 the getArg1() method is used to copy back the argument at position 1 (i.e., the modified array) into the ivalues array.

```

1. int[] ivalues = new int [100];
2. int len = ivalues.length();
3. utilJ_randM rand = new utilJ_randM();
4. for (i=0;i<len;i++){
5.     rand.nativeCall();
6.     ivalues[i] = rand.getReturnValue();
7. }
8. utilJ_bubblesortM bubblesort =
   new utilJ_bubblesortM(len, ivalues);
9. bubblesort.nativeCall();
10. ivalues = bubblesort.getArg1();

```

Figure 5. An example of wrapped routines, rand() and bubblesort(). The line numbers on the left are for reference only and do not form part of the code.

4 MEDLI

MEDLI is a graphical environment for mediating the conversion of data types between the classes of a Java application and the inputs and outputs of third-party Java software. This is accomplished through the use of MEDLI's

wizard, which allows the user to specify the mediation (i.e., conversion) of the data types between the Java input classes and the target method and then from the return object of the target method to the Java output class. It also facilitates the mediation between the input and output classes for variables that are not required in the target routine. Thus, one of MEDLI's principal functions is as a graphical "port builder" that allows the user to group the inputs and outputs of a target routine into sub-sets, termed ports, that correspond to the data types used by the application from which the target method is invoked. MEDLI makes extensive use of the Java introspection mechanism and so requires the classes of the application to provide public get and set methods for their instance variables, as is the case for Java beans and Triana classes. Furthermore, the class of which the target method is a member must have an empty constructor in the current version of MEDLI.

By building ports for target routines it becomes possible to use third-party software in an application as if it were custom-coded for the application. In particular, MEDLI provides a means to incorporate third-party software into Triana toolboxes so that it can be used in exactly the same way as other Triana components.

JACAW can be used to wrap C routines as Java classes, which can then have ports built for them using MEDLI. In this way, third-party C routines can also be incorporated into Java-based software environments such as Triana.

4.1. MEDLI and Triana

Java requires support for the integration of legacy software in order to be widely adopted by the scientific and engineering communities in Grid-enabled applications. Triana supports Java-based distributed computing and the development of MEDLI has been inspired by the need to provide pluggable code components for use within Triana. Triana can be used as a workflow composition interface for Grid applications. The basic building blocks of a Triana workflow graph are Triana components. These are usually written to a certain data type specification, but MEDLI allows third-party Java (and C) code to be connected to a Triana component's input and output ports and used in a Triana workflow.

Triana has many pre-defined data types within its system. In the example discussed in Section 5, a Fourier spectrum is produced from an input waveform. In this case, the FFT component takes a Triana SampleSet object as its input and outputs a Triana ComplexSpectrum object and the units can be connected because their data types match, i.e., one of the accepted data types for the FFT is a SampleSet. However, if we wanted to plug in a third-party Java FFT function then the process is more complicated. The data will need to be mediated between the arrays variables of the SampleSet

data type to the FFT function and the return argument will result in the construction of a ComplexSpectrum data type object. MEDLI provides this functionality.

4.2. Using the MEDLI Mediation Wizard

The use of MEDLI is divided into three phases:

1. MEDLI displays the instance variables of the classes in the arguments passed to the target routine. The user then graphically maps some or all of these variables to the instance variables of a selected Java class (such as a Triana class) through its get and set methods, thereby creating a port for the target routine. This process of port creation is repeated until all instance variables passed through the target routine arguments have been mapped to a port.
2. Next the user mediates the data returned from the target function to one or more Java output classes, thereby creating one or more output port(s). This process is very similar to the mapping of the instance variables of the target routine arguments.
3. Finally, the user mediates any other data between the input Java class and the output Java class(es) in phase 2. This is data that is not needed by the target routine, and hence is not passed to it in its arguments, but which is needed in the output port(s) created in phase 2.

MEDLI uses a file browser interface to read in the Java classes used to create the ports, and to read in the target routines. Introspection is then used to discover the instance variables of the target and the get and set methods of the Java classes. MEDLI actually creates a wrapper for the target Java code. Within this wrapper MEDLI generates code similar to:

```
target_variable = javaObject.getX();
```

when the user maps a target instance variable to the get method of a Java class. This type of mapping is applied to the inputs of the target routine. After all the inputs to the target routine have been set, the target routine itself is called. Similarly, when the user maps a target instance variable to the set method of a Java class MEDLI generates:

```
javaObject.setX(target_variable);
```

This type of mapping is applied to the outputs of the target routine, and the code generated by MEDLI for this appears after the call to the target routine is called.

A port for which all variables are inputs is tagged as an input port, and a port for which all variables are outputs is tagged as an output port; otherwise, it is tagged as an inout port. This information can then be used by visual programming environments such as Triana.

4.3. Native Calls and MEDLI

MEDLI also supports the mediation of data between Java and C functions by integrating JACAW and a compilation wizard.

MEDLI uses JACAW to create JNI bindings to the C function that gives it the Java binding it needs for the mediation wizard. MEDLI's compilation wizard then helps the user to compile the C library in a flexible and intuitive way. The compilation wizard allows the user to specify the compilation process in a compiler-independent way. The compilation wizard has a database of several compilers and translates the GUI options specified by the user into the necessary command-line options of the compiler available on a particular platform.

The current version of MEDLI builds ports for target Java methods with empty constructors. However, when JACAW wraps a C routine it creates a single class for that routine and passes the arguments via the constructor, as discussed in Section 3. Thus MEDLI's behaviour is slightly different when building ports for a Java class generated by JACAW. In this case MEDLI generates the code to invoke the constructor and the native method call, and then uses the `getArg()` and `getReturnValue()` methods to return the results of the native call.

5. Examples of the Use of JACAW and MEDLI

To demonstrate in more detail how JACAW and MEDLI are used to integrate third-party software into a component-based software environment, such as Triana, two examples will now be considered. The first example shows how MEDLI mediates the data passed between Triana and the following third-party fast Fourier transform (FFT) Java method:

```
class FFT extends java.lang.Object {
    public static double[][] realFFT1D (
        double[] data){
        ...transform code...
    }
}
```

In the above example, the `realFFT1D()` method evaluates the FFT of a real input array — the `double[]` array that is its input argument. The transformed data is returned as a 2-D array of doubles, of size `data.length()` by 2, holding the real and imaginary parts of the transform.

In Triana, the FFT method takes as its input a `SampleSet` object and outputs a `ComplexSpectrum` object. The corresponding classes have the simplified forms shown in Fig. 6.

In this case MEDLI would integrate the `realFFT1D()` method by generating the code shown in Fig. 7. In line 5, the `getData()` method is used to pass the data array in the `SampleSet` to the `realFFT1D()` method. In lines 7 and 8, the

```

class SampleSet extends VectorType
    implements Signal{
    public double data[];
    public double samplingFrequency;
    private double acquisitionTime;
    ... public get and set methods for
        all instance variables...
}
class ComplexSpectrum extends VectorType
    implements Spectral{
    public double real[];
    public double imag[];
    public double samplingFrequency;
    private double acquisitionTime;
    ... public get and set methods for
        all instance variables...
}

```

Figure 6. The Triana SampleSet and ComplexSpectrum data types.

output array is split into an array containing the real part of the transform and an array containing the imaginary part. These arrays are then inserted into the ComplexSpectrum object that is returned by the MEDLI code. Finally, in lines 9 and 10, the values of the samplingFrequency and acquisitionTime variables are set in the return object. These are not required by the target method and so are copied directly from the input SampleSet to the output ComplexSpectrum.

```

1. import triana.types.SampleSet;
2. import triana.types.ComplexSpectrum;

3. public class MEDLI_realFFT1D {
4.     public static ComplexSpectrum
       realFFT1D(SampleSet input){
5.         double[][] output =
           FFT.realFFT1D (input.getData());
6.         ComplexSpectrum retVal =
           new ComplexSpectrum();
7.         retVal.setReal(output[0]);
8.         retVal.setImag(output[1]);
9.         retVal.setSamplingFrequency(
           input.getSamplingFrequency());
10.        return retVal;
11.    }
12. }

```

Figure 7. An example of code generated by MEDLI for integrating a third-party FFT method as a Triana component.

The second example considers the integration of a C function for the in-place evaluation of a real 1-D FFT. Suppose the header file fft.h contains the following:

```
void realft (int n, double[] data)
```

This routine calculates the FFT of a set of $2n$ real values stored in the input array data. On return the array data contains the real and imaginary parts of the transform packed as follows. The real-valued zeroth and n th elements of the complex transform are stored in data[0] and data[1]. The remaining $n - 1$ complex values are stored in data[2],data[3],...,data[2n-1] with the real parts being stored in the even positions and the imaginary parts in the odd positions. Only $n + 1$ complex transform values are stored as the remaining values are all complex conjugates of the values stored.

JACAW would wrap the realft() routine in a way that is very similar to the bubblesort() example in Section 3, and the JACAW-generated code would be used in an application as follows:

```

double[] data = new double [128];
int len = data.length();
...set value of elements of data array...
fftJ_realftM realft = new fftJ_realftM(len, data);
realft.nativeCall();
data = realft.getArg1();

```

The JACAW interface closely follows that of the original C routine, and hence it is the user's responsibility to be aware of how the output data is stored. However, the current version of MEDLI cannot deal with packed data arrays of the sort output by realft(). The ability to described parameterised data types, such as array sections, may be added to MEDLI in the future using an approach similar to that used in MPI for defining derived data types [8].

6 Concluding Remarks

JACAW provides a fast and convenient way of enabling legacy C routines to be called from Java applications. MEDLI provides a graphical interface for building ports for third-party Java software so it can be integrated into software environments such as Triana. By using JACAW and MEDLI together, C routines can be used as pluggable components in Triana as if they were custom-designed Triana components.

Future versions of MEDLI will permit more general data mappings between the input and output Java objects and the data types of the target method. In particular, the ability to map from and to parameterised array sections would be useful.

Triana was originally designed as a visual programming environment for signal processing applications in which Triana components are connected to build applications that can then be run on stand-alone systems. Triana is currently being further developed for use within a distributed environment where the components making up an application may execute on different machines and are accessible as Grid

services. Thus, future work on MEDLI will develop a wizard for deploying a software component as a Grid service.

JACAW will also be extended in the future to permit wrapped code to be accessed as a Grid service. This will involve generating a description of the service in an XML-based format such as Web Services Description Language (WSDL), and publishing the service in a UDDI and/or Jini registry.

MEDLI is available from the GridOneD web site⁴ and JACAW is available for download from the JACAW web site⁵.

References

- [1] M. Baker, B. Carpenter, G. Fox, S. Ko, , and X.-Y. Li. mpi-java: A java interface to mpi, September 1998. Presented at the First UK Workshop on Java for High Performance Network Computing.
- [2] D. Beazley. Swig: An easy to use tool for integrating scripting languages with c and c++. Presented at the 4th Tcl/Tk Workshop, Monterey, California, July 6-10, 1996.
- [3] M. Bubak, D. Kurzyniec, P. Luszczek, and V. Sunderam. Creating java to native code interfaces with janet. *Scientific Programming*, 9:39–50, 2001.
- [4] V. Getov, P. Gray, S. Mintchev, and V. Sunderam. Multi-language programming environments for high performance java computing. *Scientific Programming*, 7(2):139–146, 1999.
- [5] R. Gordon. *Essential JNI: Java Native Interface*. Prentice Hall PTR, 1998.
- [6] S. Mintchev and V. Getov. *Towards Portable Message Passing in Java: Binding MPI*, volume 1332 of *Lecture Notes in Computer Science*, pages 135–142. Springer Verlag, 1997.
- [7] J. Moreira, S. Midkiff, M. Gupta, P. Atrigas, P. Wu, and G. Almasi. The ninja project: Making java work for high performance numerical computing. *Commun. ACM*, 44(10):102–109, October 2001.
- [8] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI – The Complete Reference: Volume 1, The MPI Core*. The MIT Press, Boston, Massachusetts, second edition, 1999.
- [9] I. Taylor and B. Schutz. Triana – a quicklook data analysis system for gravitational wave detectors. In *Proceedings of the Gravitational Wave Data Analysis 2 Conference*, November 1997.
- [10] M. Welsh and D. Culler. Jaguar: Enabling efficient communication and i/o in java. *Concurrency: Practise and Experience*, 12(7):519–538, May 2000.

⁴See <http://www.gridoned.org/>.

⁵See <http://www.cs.cf.ac.uk/User/Yan.Huang/research/JACAW/JACAW.htm>