

# Updates to the R-CDK Interface

Recently the R-CDK interface was upgraded to use the Java/R Interface (JRI). This article discusses the reasons for the update and some aspects of the design of the new system.

Rajarshi Guha

## Introduction

One of the goals of the CDK was to provide an interface to mathematical and statistical programming environments to perform analyses of chemical information. To that end, an interface between the CDK and the R statistical environment[1] was developed. The interface was based on the SJava[2] package for R and is described in detail in Refs. [3] and [4].

Although the SJava based interface provides access to all the functionality of R there are a number of drawbacks. With version 0.68 (which was used by the CDK) the most fundamental problem faced when using SJava is the installation. A number of people have reported problems with installing the package on Linux and the Windows version does not appear to be supported. However, as of writing v0.69 of the SJava has been released which should alleviate many problems.

Another aspect of the interface was the issue of maintainability. The design of the interface was based on a two-way communication; that is, Java methods would call R functions and pass them Java objects. On the R side these would be parsed using SJava functions and calculations would be performed. Depending on the nature of the final R object these would then pass through *converters* which would instantiate a Java class that would wrap the fields of the R object in question. On the Java side a number of classes were required to interact with a given R model object: a frontend class to build the model, a *fit* class that would wrap the fitted R model and finally a *prediction* class that would wrap the results of a prediction using the R model. Clearly, this involves a large amount of work to keep up with any changes in the underlying R functions.

Given these problems it was decided that the R-CDK interface would be migrated away from SJava to the JRI library[5]. The JRI library is a relatively recent development and is used as the basis for a number of well known R packages. Though the functionality provided by the library is quite stable, the API is expected to change in subsequent releases. Our goal was to improve the usability of the R-CDK interface, so we decided to use JRI because our design encapsulates the JRI API and hence does not require the user of the CDK to interact directly with it.

## Requirements

To compile the R-CDK interface requires that the JRI jar be available on the CLASSPATH. Since this is included in the CDK distribution this is generally not a problem. However the JRI library interacts with R using JNI and thus to actually use the R-CDK interface the native library (`libjri.so` on Linux) must be available. An important aspect is that the versions of the JRI jar and that of the native library match. Currently the CDK includes version 0.3 of JRI and so the user should have version 0.3 of the native library available.

On Linux the user must set two environment variables to allow the R-CDK interface to work:

- `R_HOME` should point to the installation directory of R
- `LD_LIBRARY_PATH` should include the directories that contain `libjri.so` and `libR.so`

Currently the code only checks for a mismatch of the JRI jar and native library.

## Design

### Interfacing R objects

Much of the design of the JRI-based interface is identical to that of the SJava based interface. Using R from CDK requires that the R engine be instantiated once in a given Java thread due to the lack of multi-threading in R. All of the modeling classes are subclasses of an abstract base class that ensures that the R engine is initialized once. In contrast to the base class for the SJava-based interface, the base class for the new interface provides a number of utility methods such as `removeObject`, `checkIsOfClass` and so on. Certain methods are declared to be abstract since the implementation of these methods depends on the nature of the model being built. These include `build`, `predict` and `loadModel`.

The development of a class that provides an interface to an R model, such as a linear regression model, involves checking that user specified parameters are valid for the model in question, checking that a model loaded from disk is indeed of the correct class and so on. In general an R model object is simply a `list`. The JRI library provides a class, **RList**, that is a Java representation of an R `list` object. Thus, the very minimum that is required is that a class representing an R model should return an object of class **RList** representing the model that was built. However, it is useful to be able to quickly access components of the model. For example in the case of a linear regression model, R allows us to extract the coefficients, rank, residuals and so on. Thus,

it is advisable to provide so-called “getter” methods that return these components from the model in terms of native Java types.

Within the R engine all objects are of a fundamental type denoted as REXP. However, an object will also have a class and mode (and possibly a set of attributes) which allows us to identify what type of an object it is, such as a numeric value or a matrix. The JRI library provides a class, **REXP**, which represents an R REXP object. The class also provides a number of utility methods that convert the REXP object to a native Java type, assuming that such a conversion is valid.

Given the R model object in terms of a Java **RList** class one can write getter methods that simply extract a component from the list and return it as a native Java type. Clearly this is a tedious task when the model object has more than one or two components. To alleviate this problem, the CDK distribution comes with an R script called `stubs.R` which given an R model object will generate a set of getter methods for each component in the model object. These getter methods will simply extract the corresponding component from the **RList** object and convert it to the native Java type. Note that the code generator ignores R objects of class ‘call’ and ‘formula’. There are some drawbacks to this approach. For example an ‘nnet’ object has a component called ‘n’ which is a 3-element vector containing the number of neurons in the input, hidden and output layers. On the R side these have a mode of `double` rather than `integer`. As a result the code generator creates a getter for this component that returns a `double[]` rather than `int[]`. In general this is not a significant problem but is something that the user should be aware of.

## Wrapper functions

An integral component of the SJava based interface was the idea of R wrapper functions. The goal of these functions was to perform checks on parameters and perform other conversions and then call the actual modeling function. This concept remains in the JRI based interface. One difference is how the parameters for an R model are passed to R from Java.

Parameters are stored in a **HashMap**. In the original SJava based design the call to the `eval()` of the **R evaluator** class allowed us to pass the parameter map to the wrapper function. As a result the wrapper function needed access to SJava on the R side to understand the **HashMap** object.

Related to this aspect, when a R function wanted to return a complex R object (such as of class ‘lm’) a *converter* was required to be registered with SJava. The converter would then detect when an object of a certain class was being returned to Java and call the appropriate R function which would take the R object and instantiate a Java class designed to represent

the model object on the Java side.

Clearly the original design was complex and difficult to maintain. Given that JRI allows easy access to R objects of type REXP and conversion of such objects to more specific R classes (such as a list or a factor) we can avoid the use of converters. The process of providing parameter values to the R wrapper functions has also been changed. Before executing an R wrapper function the user should call the `loadParametersIntoRSession()` method of the **RModel** class. This utility method iterates over all the parameters and assigns them to a list object in the R session. The list is given a unique name which is returned to the caller. The variable naming scheme is based on the use of the current time, a random number and an optional string prefix. The identifier is generated by the following code:

```
public String
  getUniqueVariableName(String prefix)
{
  if (prefix == null || prefix.equals(""))
    prefix = "var";
  Random rnd = new Random();
  long uid =
    ((System.currentTimeMillis() >>> 16)
     << 16) + rnd.nextLong();
  return prefix +
    String.valueOf(Math.abs(uid)).trim();
}
```

The caller then can invoke the R wrapper function via JRI and pass it the name of the R variable in which the model will be stored on the R side and the name of the list object containing the parameters. As a result, all R wrapper functions should be structured in the following manner:

```
aWrapper <- function(modelName, param-
List) {
  attach(parameterList)
  ...
  detach(parameterList)
  # return the result
}
```

One advantage of this approach is that use of the R-CDK interface no longer requires any packages to be installed on the R side to interface with Java.

In general, each R model class in the CDK provides access to the various components of the R model object by ordinary Java methods. However each model is stored in the R session in a uniquely named variable which can be accessed by using `getModelName()` of the **RModel** class. With the name of the variable one is free to directly access the model object within the R session.

Rajarshi Guha  
 School of Informatics  
 Indiana University  
 rguha@indiana.edu

## Bibliography

- [1] R Development Core Team. *R: A language and environment for statistical computing*. R Foundation for Statistical Computing, Vienna, Austria, 2004. ISBN 3-900051-07-0.
- [2] <http://www.omegahat.org/RSJava/>.
- [3] R. Guha. Using R to provide statistical functionality for QSAR modeling in CDK. *CDK News*, 2(1):7–13, 2005.
- [4] C. Steinbeck, C. Hoppe, S. Kuhn, M. Floris, R. Guha, and E. Willighagen. Recent developments of the chemistry development kit (CDK) - an open-source java library for chemo- and bioinformatics. *Curr. Pharm. Des.*, 12(17):2110–2120, 2006.
- [5] <http://www.rosuda.org/JRI/>.