Chapter 13

The XR Structure for Interfaces

13.1 Introduction

This chapter is a reference for the XR interface structure, the basis for interfaces from R such as those to Python and Julia described in the following chapters. The chapter provides details on the structure and its implementation. This section introduces the motivating goals for the design and for the integration of the interface packages into a project for extending R.

I expect that the main use of interfaces will be in writing application packages, where one or more server languages provide useful software for the application. Most readers of this book will, I hope, be involved in developing application packages or other extensions of R.

The chapters on individual interface packages will repeat the information needed to use the XR structure, along with specifics for the language. You can use the interface package without understanding the structure, but as always in this book I feel that understanding helps. That is particularly true with inter-language interfaces, where there are multiple levels on which programming occurs and many options for how to implement solutions.

Goals

The interface structure aims to support the use of software in one of a general family of languages. For implementing an application using such an interface from R and especially for the users of those applications, the design has three main goals: convenience, generality and consistency.

convenience: The application should be able to create functions, classes and other software interfacing to the server language in a natural R form, with

the maximum of detail handled automatically. The *user* of the application should see natural R functions and objects, with the interface causing no inconvenience.

generality: There should be no intrinsic limitations on the server language functions and objects used. The interface should be able to accomodate arbitrary classes of objects both in R and in the server language, and to communicate these between languages when that is needed.

consistency: The basic structure of the interface should be independent of the individual server language where possible, but with suitable extensions and variations to suit that language. This aids both the application programmer and end user by providing a single natural organization.

To support these goals, the XR structure uses a more advanced form of interface than most of the earlier packages discussed in Chapter 12, with facilities that take advantage of the FUNCTION and OBJECT principles in R and of some of the more modern features of most server languages.

Levels of programming

Extending R using an interface based on this structure involves packages at three levels, with deeper involvement for fewer participants at the deeper levels:

1. application packages, which will import an interface package for one or more server languages in order to use specific software written in those languages;

2. interface packages for individual server languages, providing facilities and programming tools for the application;

3. the XR package, providing the common structure imported and used by the specific interface packages.

The interface software presented by the application packages to their users will nearly all be in the form of proxies and other specialized software that utilizes the facilities of the specific interface packages at the second level. The interface packages will extend classes and use functions from XR. The application package will use language-specific objects with methods and functions sharing the common structure described in this chapter.

The ordinary R user of an application package will be largely or entirely unaffected by the underlying dependance on software from another language. R programming will continue to be functional and object-based, using standard R expressions.

Programming in an application package will use the interface in the same style, regardless of the server language. Differences will arise when languages differ in a relevant way, but otherwise programming will follow the structure described in this chapter, perhaps simplified by convenience functions for the individual server language that still follow the common structure.

The next higher level of involvement would be developing an interface package for a new language or one using a different strategy for a current language, by specializing the XR structure. Each server language package extends classes and provides methods for generic functions defined in XR. The extended classes override methods for particular server language computations, such as evaluating user expressions or returning descriptions of server language classes. The programming required is typically a small fraction of that needed to define an interface from scratch.

In terms of who-has-to-know-what, the result is a desirable "pyramid" shape. A few people will be involved in creating or modifying interfaces to specific languages. They will need to know a moderate number of details about the XR structure.

A considerably larger number will be involved in writing or modifying application packages that make use of such interfaces. I expect many readers of this book to be among them.

A much larger group again will be users of the application packages. A goal of the XR design is that they will typically be able to ignore the interface nature of the software entirely when using and programming with the application package.

As mentioned, however, I believe that *understanding* the XR design will be helpful, even when explicit use of it is not a programming requirement for your project.

13.2 The XR Interface Structure

The "XR structure" is based on two characteristics assumed for server languages:

1. objects exist that are organized into classes or a similar structure; and
2. expressions are evaluated as function and/or method calls to which the objects are arguments, and which may return objects as their value.

Python, Java, JavaScript, Julia, Perl and other languages follow versions of this paradigm, as does R itself.

The design of the interface structure has these characteristics:

- R programming with the interface uses evaluator objects, from R reference classes, to evaluate expressions and carry out other computations.

- Evaluator objects have methods for evaluation of expressions, data conversion and commonly required programming steps, such as importing modules.
- In addition, there are some generic functions to carry out computations for the evaluator, mostly for data conversion.
- Specialization to particular languages is through a few methods for the evaluator objects and for the generic functions.
- Functional and object-oriented capabilities in the server language map into corresponding proxy functions and classes in R.
- The potential software in the server language is unrestricted in the sense that any function or method in the server language is a candidate, regardless of the class or data type of the arguments or of the value returned, through the use of proxies.
- To communicate to or return from the server an object belonging to any class in R, there is a formalism for representing an object from an arbitrary R class as a named list, or "dictionary" as we will call it. Where it makes sense, a similar formalism can represent an arbitrary server language object.

There will be some limitations on implementing the design, depending on the particular language, but the overall structure seems to apply widely to the languages following this paradigm that are most likely to be attractive for interfaces from R.

Section 13.3 describes the class of evaluator objects, the common superclass of the specific interfaces. This class has methods for the essential interface computations; typically, evaluating an expression, calling a function or invoking a method in the server language and eventually getting information back to R about the results.

Section 13.4 outlines the facilities that will be provided by interface packages to support application programming and some steps needed in organizing the application package. The programming for an application will emphasize creating proxy functions and classes. There will also likely be some facilities for data conversion keyed to the application (e.g., for graphical and other summaries of results).

Section 13.5 discusses the specialization of the general class to specific server languages. A few methods need to be implemented for the specific server language in order to execute basic computations and retrieve language-specific information. Other operations have default definitions but will often be redefined by methods special to the language (to import modules or get information about functions or classes, for example).

The goal of allowing arbitrary computations in the server language is supported by the definition of *proxy objects* in R. Computed results other than simple types of data will be assigned in the server rather than being converted, with the interface call returning to R a key to identify the result, along with the class and size of the corresponding server-language object. Section 13.6 discusses proxy objects.

Proxy functions and proxy classes provide a transparent link to analogous server language computations. An R function that is a proxy will be called by users like any other function, but its sole action will be to pass the call on to its mirror function in the server language. By default, it will use the current evaluator corresponding to its interface class.

A proxy class is similarly an R reference class whose fields and methods are proxies for corresponding server language constructs. Once a proxy for a server class exists, proxy objects returned by the interface with the matching server class will automatically be promoted to objects from the proxy class. Field and method computations on these objects will map into the analogous server computations. The proxy class object contains a reference to the interface evaluator that created the object, so computations with the object do not need to specify an evaluator, even if it is not the default evaluator for that language.

Many server languages will have the ability to return a description of existing functions and classes, as does R. If so, the interface can use that information in constructing the R proxy function or class.

The net result is that typical application package use will create a proxy class object by calling a normal R function and the object created will then be treated as a normal reference class object. Section 13.7 discusses proxy functions and classes.

The goal of representing arbitrary R objects is implemented by a convention for representing the structure of objects in R as a named list containing parts of the class definition plus the representation of the data in the individual object (the slots, attributes or other pieces of the object). Building on a mechanism that can send simple objects between the languages, the representation allows the return from the server of enough information to construct an arbitrary R object. Similarly, an existing R object can be represented in this form in order to send it to server language software that will then extract the relevant information. If the server language supports it, a similar representation of a server language object could be generated (Julia can do this in a very R-like way).

The conversion mechanism can be specialized at several levels by defining methods for generic functions involved in the conversion. This can accomodate features of the server language (for example, exploiting some common data structures in R and Julia). It could also provide for alternative methods for large objects, or use a common intermediate form (for example, XML objects). Section 13.8 describes the conversion mechanism.

13.3 Evaluator Objects and Methods

The XR interfaces operate through an *evaluator object*, an R reference class object with methods to carry out computations in the server language, to convert objects between R and the server language and to obtain information such as the definition of classes or functions in the server language. Actual evaluator objects will come from a class defined for the particular language. That class inherits from the `"Interface"` class described in this section.

The examples in this and the following sections will illustrate direct use of evaluator object methods. These are the base for the interface computations. Keep in mind, however, that this is the low level of interface programming, relevant for specialized needs and for implementing higher-level features like proxy classes.

Users of application packages will likely never see this level. Even the implementers of the application packages will encounter direct methods only when needing to specialize the match between the server language tools and the functions or classes that make sense for the R application. Otherwise, software that creates proxy functions and classes will provide a simpler programming mechanism.

The XR package maintains a table of currently active evaluators, stored by the interface class name. Computations for proxy functions and classes can obtain the current evaluator for their interface class, so that no evaluator needs to be supplied by the user in normal usage. If no evaluator has been started, the request will create one. Users can have as many evaluator objects as they want from any class, but typical applications will only require one. If the evaluator is needed explicitly, a convenience function for the particular interface class will return it (see Section 13.4).

The evaluator methods for computation have a common structure that is central to the goal of unrestricted server language computations. It can be examined by looking at the basic computational method, `$Eval()`.

```
ev$Eval(expr, ...)
```

The `expr` argument is a character string specifying an expression in the server language. The expression is evaluated and the value returned to R:

```
> ev$Eval("1+1")
[1] 2
```

More general computations can incorporate data into the expression, supplied in the `"..."` arguments. Each argument will be converted to a character string. The requirement is that when that string is parsed and evaluated in the server language, the result is an object or data structure that is "equivalent" to the R object.

```
> ev$Eval("1+%s", pi)
[1] 4.141593
```

The result of evaluating the expression in the server is an (arbitrary) object or data item in that language. The result can always be assigned there, with a proxy object returned from the call to the `$Eval()` method. As a result, the arguments substituted into an expression as above can be the result of an earlier interface computation, whether it was returned as a proxy object or converted.

The default strategy is to get back converted versions of simple results (typically, scalars); all others are assigned and returned as a proxy. An optional argument, `.get=`, controls the strategy; the default corresponds to `.get=NA`, while supplying `.get=TRUE` will always attempt to get an equivalent R object.

```
> yp <- ev$Eval("%s.lines", speech)
> yp
R Object of class "list_Python", for Python proxy object
Server Class: list; size: 9
> y <- ev$Eval("%s.lines", speech, .get = TRUE)
> class(y); length(y)
[1] "list"
[1] 9
```

(For `speech`, see page 316.) The conversion strategy for sending and getting arbitrary objects is discussed in Section 13.8.

All the proxy functions and methods created by the interface will also have a `.get=` argument to allow results to be converted when that is desired. As the discussion of the strategy will explain, essentially any object in each language will have *some* conversion to the other that attempts to include all the relevant information in the original object.

This structure allows essentially arbitrary computations in the server language. Suppose, for example, an argument to a server language function requires some object that is not a simple conversion of an R object. One would first evaluate a server language expression to compute that object. This evaluation would return a proxy object reference to R that can then be passed back as the argument in question.

The arguments are incorporated into `expr` via C-style string fields, `"%s"`, and are processed in the interface by using a method, `$AsServerObject()`, that returns a character string asserted to evaluate to the appropriate server language object when parsed and evaluated as part of the expression.

For example, the Python function `parse()` in the `"xml.etree.ElementTree"` module parses an XML object in a specified file. A call to this function could be generated by:

```
> x <- ev$Eval("xml.etree.ElementTree.parse(%s)", file)
```

If the argument is a proxy object, a server language expression will be substituted that retrieves the corresponding server language object. In most implementations, this is just the "key" in the proxy object that is the name under which the server object was assigned.

For all non-proxy objects, the server language expression represents the R object as a server language expression that will be evaluated. When the object is a scalar number, character string or logical value, the expression is usually the constant representing that value in the server language. In the example, `file` could be a quoted character string that would be converted into a similar quoted string in Python.

Scalars at the simple end and representation of arbitrary R objects on the general end are the two boundaries for conversion. In between there may be a variety of special cases, depending on the class of the object and/or on the particular server language interface. The conversion strategy in both directions is discussed in Section 13.8.

The same technique for substituting arguments is used for function calls, method invocation and general commands in the server language. Given the name of a function in the server language, a call to it can be carried out by the `$Call()` method. In the example above:

```
x <- ev$Call("xml.etree.ElementTree.parse", file)
```

In general, `ev$Call(fun, ...)` calls the server function specified by `fun` with arguments corresponding to `...` and returns the value of the call.

There is an analogous interface method, `$MethodCall()`, to invoke a method in the server language from arguments specifying the object, the name of the method and any arguments to that method. To invoke the Python method `findtext()`, with the argument `"TITLE"`, on the object returned above:

```
ev$MethodCall(x, "findtext", "TITLE")
```

Both `$Call()` and `$MethodCall()` have a `.get=` argument to control whether results are kept as proxies or converted. Remember that all three evaluation methods are the base layer, but proxy functions and classes will hide this layer from the end user.

In R every expression has a value, but some languages execute commands that are not expressions, and will throw an exception if treated as producing a value. The evaluator method

```
ev$Command(expr, ...)
```

is like `$Eval()`, but without a return value.

To get a converted version of a previously computed proxy object:

```
ev$Get(expr)
```

To send a converted version of an object to the server and get a proxy back:

```
ev$Send(object)
```

It's generally a useful strategy to explicitly send objects of substantial size or complexity, rather than include them repeatedly as arguments in evaluation.

Some of the methods will not be meaningful for some languages. A language where all programming is essentially via the OOP paradigm (Java, for example), will have little need for `$Call()`. Invoking an encapsulated-style method will not be meaningful for languages not implementing this version of OOP (Julia, for example).

Packages implementing the XR structure for a specific server language will complete and customize the general structure, as will be discussed in Section 13.5 and illustrated by the packages in Chapters 14 and 15.

The XR structure is designed to make the specializations simple, while keeping to our goals for the resulting interface. Evaluator objects for a particular language will come from a subclass of `"Interface"` that provides methods for the actual communication with the language. Specialized data conversion will be implemented by methods for functions `asServerObject()` and `asRObject()`.

The table of evaluators

To free application programming from managing evaluator objects explicitly, XR maintains a table of interface evaluators currently active. The table stores objects by the class of the evaluator. Each specific server interface defines such a class, with methods that extend or override the `"Interface"` class in XR.

Access to the table is through the `getInterface()` function, typically:

```
getInterface(Class)
```

which returns the current evaluator for the particular class; if no such evaluator is in the table, one will be initialized. With no arguments, the current evaluator (the last one started for any class) is returned. The package implementing a particular server language interface will provide a convenience function that calls `getInterface()` with the right class.

The XRPython package, for example, includes a function, `RPython()`, which will return a reference to an evaluator from class `"PythonInterface"`, the current existing evaluator if there is one or else a newly created one:

```
ev <- RPython()
```

The call to `getInterface()`, and the calls to the specific functions that use it, have additional optional arguments. Use of these will allow the creation of multiple evaluators possibly with different parameters.

Connected interfaces are more likely to use multiple evaluators; embedded interfaces will usually be calling one internal evaluator for the language. With connected interfaces using general socket connections, evaluators on remote servers might be spawned to handle large problems or to use locally available server language modules. The `getInterface()` function has optional `"..."` arguments that will be passed on to the initializer method for the interface class. A special argument `.makeNew=` controls whether to generate a new evaluator. By default, a new evaluator will be started if none exists or if `...` arguments are supplied.

In addition, an argument `.select=` can be provided. It should be a function of one argument. That function will be called with a list of the current evaluators of the specified class; it is expected to return an evaluator if one satisfies its requirements, and `NULL` otherwise. A `.select` function could be used, for example, to require an evaluator with a particular host for its connection. It could also be used to cycle through evaluators for distributed computations (that's why it gets the whole list of evaluators as argument).

13.4 Application Programming

Evaluator objects provide the essential interface computations. Their methods, fields and other properties allow us to create structure for interfaces and to specialize that to different languages.

For applications, on the other hand, dealing with the evaluator objects explicitly is usually not important. Just having an evaluator there for a particular language is what's needed. Application programming can provide most computations using a "current evaluator" for the particular language, at least by default, freeing the application from generating and managing the evaluator object explicitly.

For most applications, the default evaluator for a particular language is sufficient. A function that uses the evaluator explicitly typically has a formal argument for the evaluator, with the current evaluator for the particular server language as the default. Proxy functions, for example, use this mechanism to hide the evaluator from the user.

The programming required for an application can largely be supplied by three techniques provided by the interface package:

- importing or sourcing server language code;

- defining proxy functions or classes;
- specializing data conversion for particular classes in R or the server language.

We'll deal with the first of these below, the other two in Sections 13.7 and 13.8.

Aside from possible special data conversion requirements, the R programming for the application package typically only requires calling a few functions provided by the interface package, with straightforward arguments.

Server language programming

Keeping the R side use of the application simple does sometimes depend on a willingness to do some server language programming for the application package. Proxy functions and classes are very simple to specify, but they assume that the server language function or class matches the needs in R. If there is a mismatch with the application's natural computations, the application will need to do some customizing.

The R side could do most or all the customizing, but often it is more natural to augment the server language side. In the shakespeare package, for example, the data structure parsed is in a tree format not natural for examining the underlying text. The package defines a few Python classes that extract sensible data forms and makes proxies for these in R.

During the development of application-specific server language functions or classes, you are likely to feel the need to experiment interactively with the server language code. Making changes or testing different strategies by reinstalling the application package each time is not fun. Usually, the best strategy is to import the relevant server language modules or code into a good interactive development environment for that language. Server languages likely to be attractive for an interface nearly all have such environments, often several (for example, ipython with notebooks for Python; IJulia, Juno for Julia; or the Jupyter multi-language environment).

The XR structure does have an alternative, the `$Shell()` method. This starts a simple interactive shell that parses and evaluates expressions in the server language. Expressions have to be on one line, unless each line except the last ends in the back-slash continuation character, `"\"`. The shell will have an exit command, typically the corresponding command or function in the server language.

One advantage of `$Shell()` is that expressions are evaluated in the same context or namespace used by `$Eval()` or `$Command()`.

Organizing the application package

The source code for an application package using an interface needs to provide software in R and usually also in one or more server lanaguages. The application package may construct special objects like proxy functions and classes, whose definition itself involves computing with the interface. If you want to distribute the application package, some special organizational structure and steps may be desirable.

The `"inst"` subdirectory should contain all miscellaneous software in a package; that is, anything other than things like R and C-style source, documentation and other items explicitly recognized by the installation procedure. Section 7.2 discussed the procedure by which the files in this directory and its subdirectories are installed with the package.

The XR structure assumes an organization in which the software in a particular server language is in a subdirectory of `"inst"` with the same name as the language: `"python"`, `"julia"`, etc. This convention provides defaults to methods in the interface.

The evaluator will need to have access to the server language software in the application package. For many languages, access requires two steps: updating some form of search path to include directories associated with the package; and importing individual objects or modules into the current evaluator.

The server language will usually have an internal list of directories in which it expects to find software. When an evaluator starts up, the interface package's initialization method adds its directory of server language code to the path. The application package needs to arrange for the same if it has server language functions. A directory is added to the search path of an evaluator by calling the method

```
ev$AddToPath(directory)
```

An application will nearly always want the search path to have this addition for *any* evaluator from this language. Instead of using the method directly, applications will call a language-specific function, such as

```
pythonAddToPath(directory)
```

The result is to add the directory to a table kept by the XR package. When an evaluator is generated, in this case by `RPython()`, all the directories specified in calls to `pythonAddToPath()` will be added.

The functional versions of `AddToPath()` have an extra benefit. If called at installation time from the source of a package, a load action is added that will repeat the call at load time. The search path is augmented both during installation (for proxy class definition, for example) and when the package is loaded.

If the call is from a function in the application package's source code *and* the subdirectory of server language code has been organized by the convention above, the call needs no arguments. Otherwise, arguments `directory` and `package` can be supplied, the latter if the directory is not in this package. In particular, if you need to search a directory not associated with any R package, specify an empty `package` argument; e.g.,

```
juliaAddToPath("/usr/local/juliaStuff", package = "")
```

If the package is empty and the `directory` is not given as an absolute path it is interpreted in the usual R way, relative to the working directory.

The directories added by the interface so far are stored in the field `"serverPath"` of the evaluator:

```
> ev$serverPath
[1] "/Users/jmc/Rlib/XRPython/python"
> ev$AddToPath(package = "shakespeare")
> ev$serverPath
[1] "/Users/jmc/Rlib/XRPython/python"
[2] "/Users/jmc/Rlib/shakespeare/python"
```

The method checks for duplicates, so adding a path twice has no effect.

Within the directories being searched for server language source, the code will be organized into structures, often referred to as *modules*. Details, such as what a module means and the relation between modules and files, tend to be quite language-specific. In some cases there are multiple forms of access (definitely so in R) depending on questions such as whether the object is available by simple name or only fully qualified (the `"package::object"` form in R).

The evaluator method for importing server language code is:

```
ev$Import(module, ...)
```

The first argument is the name of the module from which functions or other objects are to be imported into the evaluator. The remaining arguments modify what is imported and perhaps other options as well, and will vary among interface packages in their interpretation.

As with the search path, applications are likely to want the modules imported for any suitable evaluator. Functional versions of the method, for example,

```
juliaImport(module, ...)
```

will do this, again by using a table of import expressions in the XR package. The current evaluator for the language (if one has been started) and any evaluator started after the call to the function will include the requested import. Repeated instances of the exactly identical import request will only be executed once.

13.5 Specializing to the Server Language

In the XR structure, the interface to a particular language will be provided by a package specific to that language (the examples in this book are the XRPython and XRJulia packages for Python and Julia). End users or, more often, implementers of application packages will use functions in those packages to compute via an evaluator object or to define proxy functions and classes. The evaluator objects will be from a subclass of `"Interface"`, for example `"PythonInterface"` in XRPython.

The initialization method for the interface class will establish communication with the server language. The main distinction will be whether the interface is embedded or connected, as described in Section 12.3. An embedded interface will call a C-level entry to the server language to initialize the server language evaluator. A connected interface will create a connection, usually to an external process running the server language. The XRPython and XRJulia interfaces are respectively examples of the embedded and connected approaches.

Once an evaluator exists, the specialization of computations using it depends mostly on `$ServerExpression()` and `$ServerEval()`, two language-dependent methods used in the key step of the `$Eval()` method:

```
value <- ServerEval(ServerExpression(expr, ...), key, .get)
```

The `expr` argument is a string representing a computation in the server language, possibly written by a programmer but more typically generated from one of the other evaluator methods. The string will contain `"%s"` fields, one for each of the `"..."` arguments, which will be replaced by the server language expression for the corresponding argument.

The call to `$ServerExpression()` returns the resulting string. This is passed to `$ServerEval()`, which uses the server side of the interface to parse and evaluate the string. Creating an interface to a particular language turns on customizing these two steps, the conversion to a server language expression and the evaluation of that expression.

Each of the `"..."` arguments to `$ServerExpression()` is converted to a string by calling the generic function:

```
asServerObject(object, prototype)
```

The `object` argument is the element of `"..."`; the `prototype` argument is an object representing the target class for conversion. When `asServerObject()` is called from `$ServerExpression()`, `prototype` is an object that identifies the language.

A package specializing XR to a particular language typically defines methods to specialize `asServerObject()`. Section 13.8 goes into details.

The package specializes the evaluation of a string containing a server language expression by the definition of the reference class method:

```
$ServerEval(expr, key, keepValue)
```

The implementation must satisfy these requirements:

1. The `expr` argument is always a single string, intended to be a syntactically valid expression. The server side of the interface will parse and evaluate that string.

2. The argument `key` is also a string. If it is non-empty, the value of `expr` should be computed and returned in some form. If `key` is `""`, the expression is evaluated, but its value if any is ignored.

3. The `keepValue` argument is always a single logical value. In R this can be `TRUE`, `FALSE` or `NA`:

   `TRUE`: Always assign the value in the server and return an R proxy object.

   `FALSE`: Always convert the value to an equivalent R object and return that.

   `NA`: Return the converted value if simple; otherwise, return a proxy.

4. Regardless of the previous requirements, if the parse or evaluation throws an exception (a *condition* in R terminology), return a corresponding condition object to R.

Most server languages have tools for parsing, evaluation and exception handling; if so, a server language function can be defined to implement `$ServerEval()`. The XRPython package, for example, has a Python function `value_for_R()` with analogous arguments.

If the `key` argument is not the empty string, it can be used to assign the value; in particular, the key is guaranteed to be unique. The XR design does not require that the server return a proxy with the *same* key as passed in; interfaces may do some caching to avoid having multiple references to the same object.

The `keepValue` argument may be `NA`. Server languages do not usually have an `NA` for logicals, but often have some equivalent of `NULL` (e.g., `None` in Python or `nothing` in Julia), which will be the argument value in this case. The server language side of the interface will then use information about the computed result to decide between proxy and conversion.

When `keepValue` is `NA`, the strategy for current interfaces is to convert scalars (single numbers or character strings, typically) while returning other objects as proxies. Any data type known not to have an R equivalent will be kept as a proxy.

Although we use the term "expression" here, not all languages are like R in that everything evaluates to an object. Languages may have "statements" that are syntactically correct and can be executed, but have no value and cannot be

nested in other expressions. Empty keys accomodate the evaluation of general statements.

Whether a return value is expected or not, parsing and evaluating the expression may cause the server to throw an exception (a parse or evaluation error, for example). The interface structure in XR accommodates exceptions with some special classes and methods. If an exception occurs, the `$ServerEval()` method is expected to always return an R object from class `"InterfaceCondition"` or a subclass, even if the value was to be ignored otherwise.

The `$ServerEval()` method for a particular server language interface is conceptually a "two-liner":

1. Send the expression as a string to the server language side, along with `key` and `keepNA` in some form, and get the result back as a string;

2. Convert the string result to an R object.

The XR structure has the goal of generality in both languages: being able to evaluate an arbitrary expression in the server and return the result to R; and being able to return an object from an arbitrary R class. These correspond to a general capability in each of the two steps above.

The generality of the first step is provided by proxies in R for objects, functions and classes, discussed in Sections 13.6 and 13.7. To return a general object in the second step, the conversion first interprets the string as an object from one of a few basic classes and then converts that, using a general representation for R objects.

The conversion computation in the second step is shared between code in the server language and in R. As described in Section 13.8, conversion techniques include an explicit representation for an R object from an arbitrary class and, if possible, a similar representation for an arbitrary server language object. This representation opens the way to further customization of the returned value in R.

For example, basic R vectors have a corresponding, but different, representation in Julia. The simple string representation used for returning results, however, has only an untyped list-like vector. To ensure that a vector object of the right type is returned, the server language code generates the explicit representation for an object from class `"vector_R"`:

```
vector_R <- setClass("vector_R",
   slots = c(data = "vector", type = "character",
             missing = "integer"))
```

For this object, slot `"data"` may be a list of the individual elements. A Julia method generates the representation, the corresponding string is turned into an object of class `"vector_R"`, and a method for the generic function `asRObject()` turns that into the desired vector.

Both R and Julia support computations with complex-valued vectors/arrays, but the intermediate representation has nothing corresponding. To return an array of complex data of type `"Array{Complex{Float64},1}"` from Julia, the server side constructs a dictionary describing a `"vector_R"` object. Its `"data"` slot is a list of the values, formatted as strings; its `"type"` slot is `"complex"`. The `"vector_R"` method for `asRObject()` interprets this and produces the intended `"complex"` vector.

From the application's perspective, the interface should just do the right thing:

```
> x <- ev$Send(1:3 + 1i)
> x
Julia proxy object
Server Class: Array{Complex{Float64},1}; size: 3
> y <- ev$Eval("%s * 0.5",x)
> ev$Get(y)
[1] 0.5+0.5i 1.0+0.5i 1.5+0.5i
```

Section 13.8 describes the conversion of data in both directions, to a string for `$ServerExpression()` and from a string for the value returned by `$ServerEval()`. Customization in both directions uses generic functions with methods specialized to the class of the object and/or to the particular interface.

Interface classes will have some additional methods for computations other than basic evaluation.

`$ServerRemove(key)`: Delete the reference in the server implementation previously assigned as `key`. The goal is to recover memory in the server language, but how this happens or even whether it happens does not affect the interface itself. If no special method is defined, the default version does nothing.

`$ServerClassDef(ClassName, ...)`: For the server language class or data type of the specified name, return an object listing the fields and methods. What is known about the class and what additional `"..."` arguments are needed will vary with the language. If there is no reflectance information in the language, the method returns `NULL`. This is the information used in defining proxy classes.

`$ServerFunctionDef(what, ...)`: Return a proxy function for `what`; that is, an R function object that when called will evaluate a call to the specified server language function. Where possible, the method will use the server object defining the target function, along with knowledge of the calling conventions of the language.

`$ServerSerialize(key, file)`; `$ServerUnserialize(file)`: Serialize the proxy object stored under the key, writing the resulting byte string to the specified `file`; unserialize the contents of the file, returning a proxy object. Used to save proxy objects.

`$ServerGenerator(class, module)`: Return the server language expression for the generator function corresponding to the given class and module. The default is to use the class name as the name of the generator and to prepend the module with `"."` as the separator.

`$ServerAddToPath(directory, pos)`: Add the `directory` to the language's search path form importing functions, classes, etc. If the `pos` argument is `NA` (the default), append the directory to the path list; otherwise `pos` is the desired position of the new directory, interpreting the position according to the particular server language (but you really need to know what you're doing if you mess with this).

Server language methods for class metadata and for serializing are important for those computations. Nearly all the languages in Table 12.1 have both.

The methods described so far are all used internally, not called by end users or application packages. Some more directly user-visible operations may also have language-specific methods.

The `$Import()` method in the XR package only takes a single module name as argument. R and many other languages provide for importing only some objects from a module. The `$Import()` methods for the interfaces in Chapters 14 and 15 take a module as first argument and additional argument(s), interpreted according to the import semantics of Python or Julia.

Server languages may or may not require the imported name to be preceded by the module name in later expressions. If the function `parse()` has been imported from module JSON, is it defined simply as `"parse"` in the namespace or must it be called as `"JSON.parse"`? The first case implies that a definition of `parse()` previously existing in the server namespace for the interface will either be masked by the imported version or will mask it. The `$Import()` methods in our packages will behave according to the particular language's protocol. This may include options to use either the simple or fully qualified form.

The `$Source()` method is also likely to be replaced by a language-dependent version that will be more efficient, particularly for large files. In Python, modules are interpreted as files, so `$Source()` is a form of `import` command. In Julia, modules and files are distinct; the language has a function, `include()`, that implements `$Source()`.

13.6 Proxy Objects

One of the design goals for the XR structure is to support arbitrary server language computations. This requires that any data or object in the server language can be supplied as an argument to `$Eval()` and the other methods described in Section 13.3. Also, the computation can return an arbitrary server language object as its value. *Proxy objects* are essential to provide this generality; that is, objects in R that are essentially a reference to objects in the server language.

If a server language evaluation produces a result that will not be converted to an R object, it assigns the server language object, using a key supplied by the R side of the evaluator. The server language implementation guarantees to retain the object and to find it when the corresponding key is sent from R. On the R side the key is embedded in a proxy object.

If the R proxy object is used as an argument to a subsequent interface method, `$ServerExpression()` substitutes an expression that retrieves the previously assigned object, usually just the key as a name. The assumption is that the object has been assigned in a workspace or module. An interface package that wanted to use some other mechanism for storing the objects (e.g., an explicit table) could do so by providing methods for `asServerObject()`.

The XR interfaces return proxy objects from class `"AssignedProxy"`, a subclass of `"character"` with the data part being the key for the object. The keys themselves are generated by the `$ProxyName()` method in XR, and guaranteed to be unique in a session both within and between evaluators. The specific interface package and the application using it do not need to generate keys for proxy objects, and shouldn't.

The `"AssignedProxy"` class also has slots for the evaluator that created the object, the server class, the module in which the class is defined and the size of the object. The auxiliary information can be useful when deciding whether a computation had the expected result, and potentially in choosing a method to convert the data to R (e.g., to use a special method for converting large objects). The server class and module allow the interface to detect that a proxy class exists for the object. If so, the returned result is promoted to an object from that proxy class, with the `"AssignedProxy"` object as one of the fields.

`"AssignedProxy"` objects are returned from the server language using the general representation of an R object defined in XR (see page 288).

The strategy for deciding when to return a proxy is that scalars (including single numbers, character strings and logical values) are returned by conversion but other types are returned as proxy objects. The evaluator method `$Get()` forces conversion by calling `$ServerEval()` with a non-empty `key` and `keepValue = FALSE`. An interface package could choose a different strategy simply by a different

implementation of the `$ServerEval()` method.

The key returned as the value of a server language expression is not required to be the same as the key passed in from R. Most of the server languages deal with references to objects, meaning that a relatively inexpensive check can be made to see whether the current object is the same as one recently assigned. In this case, it's a good idea to return the previous key. This reduces clutter on the server side and allows R computations to check equality of objects more efficiently.

Proxy objects allow the application to avoid back-and-forth conversion of data. This is desirable for efficiency, but in addition, the object may change subtly on the round trip. Languages may have many similarities—the XR strategy exploits this—but they inevitably have differences, and in particular in terms of the basic structure for data. So, for example, the two R objects:

```
c(1.1, 2.2, 3.3)
list(1.1, 2.2, 3.3)
```

are different, although similar. But the standard Python object corresponding to them is the same:

```
[1.1, 2.2, 3.3]
```

Converting either of the R objects to Python and getting it back will produce the identical result (the list). To maintain the distinction on the server side requires using some special class of objects, as we'll consider in Section 13.8.

Within a session, proxy objects will persist at least until explicitly removed by the `$Remove()` method and will be unchanged unless by some computation in the evaluator. Across sessions, proxy objects are not preserved, as we consider next.

Saving proxy objects; serialization

The objects computed and stored in the server language are maintained only while the corresponding evaluator exists and therefore only for the current R session. To save a copy of the objects on a file, evaluators have a `$Serialize()` method. Essentially all languages of interest for an interface have a mechanism for serialization; that is, for encoding an object as a byte stream that can later be decoded to recover the object, usually with no loss of information. A corresponding `$Unserialize()` method reads the file and converts the result back to an object.

The XR package has a default implementation in R that converts the object to an R object which is then coded using the R `serialize()` function. This approach is very much a choice of "last resort". Interface packages for a particular language will instead use the serialization features of that language to encode the server

language object without conversion, with a corresponding implementation for unserializing. The interface package will implement methods `$ServerSerialize()` and `$ServerUnserialize()` that use appropriate code in the particular server language for serialize/unserialize computations. The server language implementation is likely to be better on all criteria: less ambiguity or inaccuracy in the encoding and less computation required.

Serialization in many languages is modelled as a multi-object storage. An open output connection is expected as an argument to the serialize function. Repeated calls append more objects; similarly, unserialize methods return each of the objects written to the same file.

For R with its OBJECT principle, it may be more natural to think of a one-to-one relation between file and object. The XR structure supports either view through an `append` argument to the `$Serialize()` method. If `append` is `FALSE` (the default), the file is opened and truncated; otherwise the next object is appended to the file. Application code that wants to serialize several objects to one file should call the first time with `append=FALSE` to guarantee that anything previously on the file is wiped out.

When using an interface for serialize/unserialize, keeping a connection open between interface calls may be difficult or undesirable. The packages in the XR family support multiple serialization by closing the file and opening it again in append mode. For unserializing, a more restrictive technique of keeping track of the file position between calls would be needed. Instead, the `$Unserialize()` method always reads the entire saved file, returning a list-like object in the server language containing all the serialized objects as elements. `$Unserialize()` has an argument `all`, which should get the same value as used for `append` in the `$Serialize()` call(s) that created the file. If one object is found, the `multiple` argument determines whether the value is the list of length one or the single object.

13.7 Proxy Functions and Classes

Just as proxy objects are objects in R that in fact refer to objects in the server language, so proxy functions and classes are used in R but turn into function calls and OOP computations in the server language.

The function objects and the class definitions will preferably be provided in an application package. Users of the application package will call the functions and work with objects from the classes in a standard R way, largely unaffected by the server language implementation.

This section discusses the programming steps to create proxy functions and

classes. For the application author, the programming amounts largely to calls to two functions, one that creates a proxy function object and the other that creates an R class to serve as a proxy for a server language class. The actual functions called will likely be specialized for the particular server language, but they all work the same way and in turn will call versions in the XR package, which we describe in this chapter.

The computations to create proxy functions and classes will nearly always use some metadata information in the server language, similar to the function objects and class definition objects in R but with variations reflecting the particular language (page 282). This information may not be easily available when an application package is installed; if not, two mechanisms in the XR package facilities provide alternatives (page 283).

As a running example, we will use the shakespeare package. All calls for proxy function and class creation for this package are collected in the file `"R/proxy.R"`.

Proxy functions

Proxy functions are objects from a subclass of `"function"`. When called as a function, they use the methods of an interface evaluator to call the corresponding server language function and return its result (usually a proxy object). This could be programmed directly just by the appropriate use of the `$Call()` method, but the proxy function has additional information and ensures, for example, that a module is imported if needed. Proxy functions for particular server languages may include more information, such as documentation for the function. For a strongly typed server language, the identity and type of the actual arguments could be checked. In the case of XRPython and XRJulia, most of the checking is left to the server language.

The general class for proxy functions is `"ProxyFunction"` in the XR package. Individual interface packages usually subclass this to include special features. In the XRPython package, for example:

```
PythonFunction(name, module)
```

creates a proxy function object from class `"PythonFunction"`. (The initial capital "P" is because this is a generator for that class and the interfaces have a convention of capitalizing special class names.)

Calls to the proxy function turn into calls to the Python function of the specified name and module. The Python function `parse()` in the xml.etree.ElementTree module can be used from an R proxy by creating an R function:

```
parseXML <- PythonFunction("parse", "xml.etree.ElementTree")
```

Proxy functions provide a more natural and convenient interface than the `$Call()` method. If the application package has this proxy function defined, then

```
hamlet <- ev$Call("xml.etree.ElementTree.parse", "hamlet.xml")
```

is simplified to

```
hamlet <- parseXML("hamlet.xml")
```

A conceptual simplification for the end user is that the evaluator is hidden, being selected automatically in normal circumstances. In addition, the proxy function takes care of importing the module. The user of the application package does not need to call actual interface routines or to initialize the interface evaluator.

Proxy classes

A *proxy class* defines an R class corresponding to a specified server language class. Objects from the proxy class appear to have fields and methods in R corresponding to those in the server class. Expressions are written in R, usually the same expression that one would write in the server language with the operator `` `$` `` in R replacing the dot operator in the server.

A proxy class in R is created by calling `setProxyClass()` in the XR package, or a server-language-specific function that extends `setProxyClass()`. For example, `"ElementTree"` is a Python class in module `"xml.etree.ElementTree"`. The Python object returned by the call to `parseXML()` in the example above has this class. The package defining and using the proxy function would also likely create a corresponding proxy class by:

```
ETree <- setPythonClass("ElementTree", "xml.etree.ElementTree")
```

(We'll look at this example in more detail in Section 14.5.)

The reference class created is a subclass of `"ProxyClassObject"`. References to its fields and methods are interpreted as proxies for similar field and method access in the server language.

Objects from the proxy class may be created directly in R, but more frequently are the value of function calls or other computations via the interface, as in the call to `parseXML()` in our example.

Methods and fields in the server object can be invoked or accessed from R by standard R OOP expressions, using the names of the method or field in the server class. The `"ElementTree"` Python class has a method `findtext()`, among others. Then the corresponding class in R will also have a method `$findtext()`. The R method call will evaluate a call to the corresponding Python method. For the object `hamlet` in the example:

```
> hamlet$findtext("TITLE")
[1] "The Tragedy of Hamlet, Prince of Denmark"
```

For the user of the application package, this combination of proxy function, class and object achieves the goal of making the computation essentially transparent to its implementation via an interface.

Calls to proxy functions trigger corresponding function calls in the server language. If these return a non-scalar value, the R function will return an assigned proxy object specifying the server class as a field. If the interface evaluator finds a corresponding proxy class, the value returned will be from that class, with the proxy object as a field. The user, or more frequently special computations in the application package, can use the methods and fields of the object as they would be used in the server language.

Metadata from the server language

In the examples above, the server language function or class was just supplied by name, optionally with the name of the module where it was defined. Nothing more was needed because Python, like most server languages, provides information which can be used to define specific function and class structure. Methods in the interface evaluator inspect the corresponding object in the server language and return the information to R. As in R, the reference to the name and the module (package in R) is sufficient to find the metadata needed for the proxy.

There is a catch, however. The computations to define the proxy function or class will be part of an application package in a typical project to extend R. It may not be feasible to do those computations when the package is installed. To get around such problems, the XR interface structure supports two optional techniques: load actions and/or a setup step. We'll describe these, but first let's examine how the server language metadata is used.

The interface method creating a proxy function object will access the server function as an object, if possible, obtaining the argument names and perhaps types, online documentation or other information.

Similarly for proxy classes, languages will differ in the information available and in how a class can be used. The computation to create the proxy class uses this information to define the fields and methods in R.

The metadata for a proxy function is returned by the `$ServerFunctionDef()` method, specialized to the individual server language. The generator function for the corresponding proxy function class will use this information; for example, in the `initialize()` method for class `"PythonFunction"`.

For proxy classes, the metadata for a class definition is obtained from the method `$ServerClassDef()`, which returns a named list with the fields and meth-

ods as elements. The methods component is itself a named list, whose names are the method names, with the elements being the functions to use as proxy methods in R. The default `$ServerClassDef()` in the `"Interface"` class returns `NULL`, indicating that no metadata information is available.

Languages differ in their implementation of classes; variations will be dealt with in `setProxyClass()` and by the definition of `$ServerClassDef()`. For example, Python has no formal fields. The interface attempts to infer the fields using an object from the class, generating the default object from the class unless the `example=` argument supplies another object.

Other languages will behave differently. Julia has composite types, with fields defined by the metadata. Methods are functional, not encapsulated, so the metadata returned by `$ServerClassDef()` does not generate proxy method calls. Java supplies very detailed information on fields and methods, including types for the arguments and for the return value of all methods. A proxy class definition would have unambiguous fields and methods.

Load actions and setup steps

Load actions are, usually, functions that are called during package loading, with the namespace of the package as the only argument (Section 7.3, page 119).

A setup step is an R script which itself writes software into the source package. The function `packageSetup()` from the XR package is designed to run such setup scripts. `packageSetup()` ensures that the computations have the package namespace available for use and that the environment for the computations is associated with the package (for example, that classes and other OOP objects will have the name of this package in their `"package"` slot).

Load actions and setup steps are relevant for application packages using interfaces if one cannot assume that the server language evaluator and all relevant modules are available when the application package is installed.

Repositories may require installation with only R, the packages supported by the repository and standard compilers available. There will be no problem if the interface is embedded and the modules used are either available through the server language directly *or* supplied in another R package already installed.

If a module or other server language software is actually part of an application package, it is located using `system.file()` to find the installed package, if the package has been installed and the software was supplied in the `"inst"` directory of that package's source directory. This includes a reference to locations in the current package, even if it has not yet been installed, because of the order of actions in the installation process, as outlined in Section 7.2, page 114.

There remains the possibility that either the language is unavailable (and the

interface is connected) or that a particular module cannot be assumed to be available at installation time. Moving the computations to a load action will allow the installed version of the package to be created and distributed without additional requirements on the central repository (but *only* if the repository omits the default test load in the `INSTALL` command).

The actions to define proxy functions and classes can be delayed until load time by defining a suitable load action. For example, in Section 15.5 we define a proxy for the Julia type `"SVD"`. To do this at load time we define a function

```
.loadSVD <- function(ns) {
    genr <- setJuliaClass("SVD", where = ns)
    assign("SVD", genr, envir = ns)
}
```

The namespace argument is used in the call to `setJuliaClass()` to store the class definition there and in an explicit assignment of the generator object.

With the load action defined, the package's source code needs to set it:

```
setLoadActions(proxySVD = .loadSVD)
```

This assigns metadata in the installed package that is detected during the package loading. The argument name is just to identify the action in any warning or error message.

If a proxy function or class is created at load time, then any R object that depends on it will also have to be delayed. In particular, an R class that has a proxy class as a field or a superclass will need to be created by a load action. The application package will need to do this explicitly.

Suppose an R class `"SvdJTimed"` is a subclass of the proxy class, with an additional numeric field `"time"`. Suppose it would usually have the definition:

```
SvdJTimed <- setRefClass("SvdJTimed", contains = "SVD",
    fields =c(time = "numeric"))
```

If the definition of `"SVD"` is delayed until load time, the subclass will need to be defined then also, after the proxy class. This is most easily done by adding the class definition to the load action that was defined for the proxy class.

```
    genr <- setRefClass("SvdJTimed", contains = "SVD",
             fields =c(time = "numeric"), where = ns)
    assign("SvdJTimed", genr, envir = ns)
```

These two lines would be added to the body of `.loadSVD()`.

An alternative to the `onload` option is to use a setup step to precompute the information. The calls to create the proxy function or class are the same as in a direct call, with an additional `save=` argument.

In the setup step, the `save=` argument is a file name or an open connection. The effect is to write, on that file or connection, an R language call defining the corresponding function or class. The call has all the metadata incorporated explicitly, so evaluating it does not need any server language computations, either at installation or load time.

Here is an example of a setup step, using the shakespeare package as the application and the XRPython interface. The shakespeare package defines proxy functions and classes, some of which refer to the package's own Python code.

The package defines three proxy classes—`"Act"`, `"Scene"` and `"Speech"`—from its own module, thePlay. Their corresponding direct proxy calls would be:

```
Act <- setPythonClass("Act", "thePlay")
Scene <- setPythonClass("Scene", "thePlay")
Speech <- setPythonClass("Speech", "thePlay")
```

Suppose we decide to generate all class definitions in a setup step, with the complete proxy class definitions written onto a file in the package's source directory, say `"proxyClasses.R"` in the `"R"` subdirectory.

It is recommended to put the R script for a setup step in the `"tools"` directory of the source package, as evidence of how the proxy definitions were computed. The setup script in this example, in `"tools/setup.R"`, could be:

```
library(XRPython)
con <- file("R/proxyClasses.R", "w")
setPythonClass("ElementTree", "xml.etree.ElementTree",
               save = con)
setPythonClass("Act", "thePlay", save = con)
setPythonClass("Scene", "thePlay", save = con)
setPythonClass("Speech", "thePlay", save = con)
close(con)
```

The script opens a connection and passes it as the `save=` argument to each of the `setPythonClass()` calls, to write all the output on the same target file.

To carry out the setup step, set the working directory and call `packageSetup()`:

```
> setwd("~/localGit/shakespeare")
> XR::packageSetup()
```

We used the default location, `"tools/setup.R"` for the script, so no arguments were needed to `packageSetup()`. The script creates an environment simulating

the installation of the package to ensure that `packageName()` returns the correct name. Note that the package must be installed and loadable, *before* running the script. In this example, the module containing the class definitions is imported from the package's directory. The proxy functions and classes don't need to be defined for the initial installation, but if they are to be explicitly exported, the `export()` directives should be inserted after running the stup step the first time.

To use a different name for the proxy object created instead of the server language function or class name, supply an `objName=` argument in the call. The shakespeare package defines a proxy function for the `parse()` function in the Python module dealing with XML data, as shown on page 280. The proxy is renamed as `"parseXML"` to avoid confusion with R's `parse()` function. In a setup script:

```
PythonFunction("parse", "xml.etree.ElementTree",
                   save = TRUE, objName = "parseXML")
```

A similar `objName=` argument to `setPythonClass()` would rename the class generator object. This example does not specify a file name or connection in the `save=` argument. A file with a name constructed from the function name will be written in the package's `"R"` directory.

The setup step would only need to be run again if the server class or function definition changed in a way that affected the script generated.

13.8 Data Conversion

The goal of generality in the XR interface strategy seeks to include any computation that can be expressed in the server language. Proxy objects, functions and classes provide the central mechanism for this. The goal also includes the ability to use arbitrary objects in either language, meaning that there must be some way to describe these objects and to convert them to equivalent objects in the other language, as nearly as possible.

This section describes the XR structure for conversion and the procedures for sending data to the server language and getting R objects back.

The actual communication between the languages is through character strings: the `$ServerEval()` method sends a character string to the server language, which parses and evaluates that string and returns the result, also as a character string. The R side of the interface interprets that string as an object.

To send an object to the server, the object is converted to a string by a call to

```
ev$AsServerObject(object)
```

The string returned is an expression in the server language that evaluates to the equivalent of `object`. That string will be substituted into a `"%s"` field, by `$Eval()` or some other method, to produce the eventual argument to `$ServerEval()`.

The `$AsServerObject()` method just calls a generic function:

```
asServerObject(object, prototype)
```

where `prototype`, by default and usually, is an object identifying the interface language. Methods for `asServerObject()` customize the conversion, both in a language-independent way and specialized when the language offers better analogues to R classes (page 292).

To convert a server language result to an R object, the string returned to `$ServerEval()` from the server language is first converted to an object by the function `valueFromServer()`. This object is produced by applying a standard structure for conversion (we'll describe that next). The result is then the argument to a second generic function, `asRObject()` (page 299).

There are three components to the conversion structure: an elementary level that defines string equivalents for a subset of possible objects and two representations defined in terms of the elementary level to specify arbitrary R and server language objects.

The XR package provides an implementation of the conversion structure. Specific interfaces can modify the conversion to a greater or lesser extent. The XRPython and XRJulia packages are examples: the Python interface uses the basic conversion essentially as is; the Julia interface replaces some conversions for classes where Julia has a more direct correspondence to R.

Elementary object conversion

The XR package implements `asServerObject()` and `valueFromServer()` through the JSON object notation. Objects representable in this notation form the elementary level for conversion. Interface packages that use the JSON implementation only need to supply server-language software to parse the JSON string in order to get objects from R and software to produce a JSON string for the same range of objects to represent the server object returned to R.

JSON (*J*ava*S*cript *O*bject *N*otation) is a widely used standard for simple data exchange. For our purposes, it defines a representation of an object as a text string for three kinds of objects:

i. scalars, in the form of numbers (in a version of standard scientific notation), character strings, and the reserved names `true`, `false` and `null`;

ii. unnamed lists of representable objects, enclosed in square brackets and separated by commas;

iii. named lists of representable objects, with each element represented as a character string name followed by `":"` followed by the representation of that element.

Several R packages implement JSON conversions, producing a string from an eligible R object and parsing such a string to return the corresponding object. The current XR implementation uses the jsonlite package, [29]. All the server languages discussed have their own corresponding modules or packages to perform similar conversions.

The second and third forms are referred to in JSON as arrays and objects, but that terminology is too confusing here. Where some general term is needed, the second will be called a JSON list and the third a *dictionary* (the term in Python, Julia and other languages for the analogous structure).

The use of JSON has advantages of simplicity and availability. The standard can be described in a page (`www.json.org`). The notation is widely used; nearly all the languages needed for interfaces have an implementation, and writing one would not be a major undertaking. Several existing basic interface packages use it, including interfaces to Python [6] and JavaScript [28].

The JSON notation is limited in what it can represent. The objects representable in this form do not include all the "basic" R objects, in the sense of the objects that belong to one of the built-in data types (Section 6.1, page 97). Of the vector types, the notation will represent `"logical"`, `"character"` and both `"integer"` and `"double"` forms of numeric. But neither `"complex"` nor `"raw"` can be accommodated. These are handled by the representation for arbitrary R objects.

Also, an R `"list"` object with scalar elements all of the same basic type has the same JSON representation as a vector of that type. The server language may or may not have such a distinction either: Python does not but Julia does.

The natural strategy for an interface package will depend on the server language. Julia has closer analogues to R vectors than does JSON. The XRJulia interface defines conversion methods for these (both in R and in Julia).

Python has an object hierarchy similar to JSON. The XRPython interface uses the default XR structure.

Representing arbitrary R objects

Dictionaries are the mechanism for representing arbitrary R objects. The dictionary will be interpreted as a general R object if it has an element named `".RClass"`. The value of that element (along with an optional `".package"` element) is taken to specify the class.

There are also reserved names for the type and the inherited classes. Other than these reserved names, the remaining elements of the dictionary will be interpreted as the slots in an S4 class definition, including all inherited slots. A class that extends "vector" or one of the vector types also has a pseudo-slot ".Data" for the data part of the object.

For example, although the "ts" class for time series data is an S3 class, it is consistent with an S4 vector class with a numeric slot "tsp" for the time series parameters. Here is a JSON representation of the uspop data in the datasets package:

```
{ ".RClass" : "ts", ".package" : "",
   ".type" : "double", ".extends" : "ts",
   ".Data" : [3.93,5.31,7.24,9.64,12.9,17.1,23.2,31.4,39.8,
              50.2,62.9,76.0,92.0,105.7,122.8,131.7,151.3,
              179.3,203.2],
   "tsp" : [1790.0,1970.0,0.1] }
```

The representation generalizes to any vector with attributes.

Arrays and their special case, matrices, are included in the XR implementation similarly, although they are not in fact S3 classes. Some special methods in XR give them the appearance of a formal class with slots "dim" and "dimnames", plus a data part. If the interface does not have a special asServerObject() method, a matrix will be converted using the general representation. In the example below, ev is an evaluator for the Python interface:

```
> mProxy <- ev$Send(matrix(1:12,3,4))
> ## Converted in Python to a dictionary:
> ev$Command("print %s.keys()", mProxy)
[u'dim', u'.type', u'.package', u'.RClass', u'.Data', u'.extends']
> ## Getting it back decodes the dictionary:
> ev$Get(mProxy)
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

We printed out the keys (element names) in Python. They include ".Data" for the data part, "dim" and "dimnames" for the slots, plus the specially named elements. Server language computations might interpret this object and produce something more idiomatic for the language, but even working directly from the dictionary, the application could compute some revised values and returned the result to R by this mechanism.

The conversion of basic vectors and other built-in types uses the function `typeToJSON()`, which generates a JSON representation for the data in an object, using only the type and the data. The representation by `typeToJSON()` gives essentially the full information, but not directly for types other than those representable as numbers or character strings. Otherwise, a combination of formatting the individual elements and, as a last resort, serializing the object provides at least a mechanism for transmitting the data and later retrieving it. See page 294 for details.

The slots in an object will often be themselves objects from some class other than the basic data types. The explicit representation will be applied to these as well. For example, in Section 10.2, page 149, we considered the definition of a class `"track"`:

```
track <- setClass("track",
    slots = list(lat = "degree", long = "degree",
                 time = "DateTime"))
```

The slots of this object would be represented in JSON by explicit forms for classes `"degree"` and `"DateTime"`.

R accepts objects from a subclass of the declared class for a slot. For simple inheritance, the server software will still be able to find the required slots. For a virtual class like `"DateTime"`, the R side may need to coerce the slot into a particular representation required by the server computations. In the example above, if the server required the `"POSIXct"` representation of date/times, the R side would require:

```
object@time <- as(object@time, "POSIXct")
```

This could be in application-specific code or in a method for `asServerObject()` for class `"DateTime"`.

Languages can return R vectors of a specific type in the JSON notation by using the `".RClass"` representation for an object from the R class `"vector_R"`. This class represents a vector in an explicit form, including its type as an explicit slot, with a `"data"` slot for which a JSON list is adequate and a slot `"missing"` to explicitly denote elements that should be `NA` in R. For details and an example, see page 300.

Arbitrary server objects

XR also has a mechanism for representing arbitrary objects from the server language. Although the mechanism is general, it is used most often to represent objects from a server language class defined by a set of named fields/slots.

The server language code responsible for returning a value to R will use the ".RClass" mechanism to return an R object of class "from_Server" that represents the particular server language object. This class has slots for the server class, the module and the language, plus slots to accomodate essentially arbitrary structure. Specifically, the "data" slot holds any R object while the "fields" slot is expected to be a list with names corresponding to the named fields of the server language object. The contents of the relevant slots will be set by the server language computation that responds to the request for a converted version of the object.

In the case that the server class is defined by its fields, the converted object can be used in much the same way as the proxy class objects in the previous section. The `$` operator will extract or set fields and the valid fields will be those in the server language class. The difference is that the computations are now all being done in R and the contents of the fields will have been converted, by whatever method applies to *their* class.

Particular server language interfaces may subclass "from_Server" if that simplifies handling some special server classes. The XRJulia package defines a class "from_Julia". In this case, actually, most of the specialization is done on the Julia side, taking advantage of Julia's functional OOP. For some examples of general conversion, see Section 15.6, page 335.

The application package can request an arbitrary result to be returned by supplying .get = TRUE to an evaluator method or a proxy function. The server language side of the interface will convert basic objects, usually through the JSON mechanism, and will recognize other classes of objects for which specific R analogues exist (for example, arrays in Julia). Then, generally, an object from a server class defined by its fields will be converted by the "from_Server" mechanism provided that the objects in the fields are themselves convertible. This is a general, recursive procedure for converting arbitrary objects.

The limitation is that some "basic" objects may simply have no corresponding R object. Either the specific interface or the application will then need to come up with some combination of:

- server-side computations that recode the information in a form that does correspond to an R object; and
- R computations that interpret this converted object in terms of the information in the server object, as that applies to the particular application.

The XR structure cannot solve the specific problems, but it allows arbitrary conversions in the sense that if there *is* a solution, then the conversion should fit into the "from_Server" class. The mechanism is the "data" slot in that class, which is essentially for "anything else".

As an example, suppose that some class or data type in the server has no R analogue, and the decision is to serialize the server object. If the serialized object can be kept as a `"raw"` vector in R, then the object can be converted to a `"from_server"` object with that vector in the `"data"` slot, no fields and the appropriate `"serverClass"` and `"module"` values.

We may not seem to be much further ahead but there are two key advantages. Most importantly, this allows conversion of objects that have fields of a non-convertible type, making the rest of the object available. Also, the R object can be passed back to the server side of the interface, presumably without loss of information.

Sending an R object

The character string to be passed to the server is defined to be the value returned by the generic function:

```
asServerObject(object, prototype)
```

The function will be called from an evaluator whenever an R object needs to be passed to the server language, typically as an argument to a function or method call. You can see the expression explicitly by calling the corresponding evaluator method:

```
ev$AsServerObject(object)
```

In fact, all the conversions to a server language expression in the interface methods are generated by a call to this method.

In calls to `asServerObject()`, the `object` argument is the R object to convert. The `prototype` argument represents the target for the conversion. In the evaluation methods, this object is the `$prototypeObject` field of the evaluator and has a class specifying the interface; for example, `"PythonObject"` or `"JuliaObject"`.

The JSON-compatible text string for conversion to the server is returned by a call to another function, `objectAsJSON()`. The default `asServerObject()` method shows how this works:

```
function(object, prototype) {
    jsonString <- objectAsJSON(object, prototype)
    if(is(jsonString, "JSONScalar"))
        jsonString
    else
        gettextf("objectFromJSON(%s)",
                 typeToJSON(jsonString, prototype))
}
```

The translated string for a scalar has a class that inherits from `"JSONScalar"`. The method assumes that the scalar format will be legal as a constant expression in the server language. For all other objects, the method constructs a call to the server language function `objectFromJSON()`, with the JSON string as the argument. Each server language interface will have a definition for this function, usually a simple call to the language's module for dealing with JSON.

The non-default methods for `asServerObject()` in the XR package all return names for objects in the server language. Methods for proxy and proxy class objects return the character string containing the unique server language key generated by the interface. The XR strategy assumes that proxy objects have been assigned in some environment in the server language using the key as the name. The character string key is then a reference to the corresponding server object.

The method for R objects of class `"name"` returns that name as an unquoted string. If an application makes an explicit assignment:

```
ev$Command("piBy2 = %s", pi/2)
```

then `as("piBy2", "name")` as an argument to `asServerObject()` will refer to the assigned object. Generally, however, it's better to use the unique keys generated by the interface to assign objects automatically.

The value returned by `objectAsJSON()` in the method shown above is the JSON string representing the R object. The string is substituted into a server language call to `objectFromJSON()`, unless the R object is to be treated as a scalar. So, for example, the R object `1:4` would be sent to the server as the string

```
objectFromJSON("[1,2,3,4]")
```

The function `objectAsJSON()` is another generic function. The JSON string returned by `objectAsJSON()` must represent a scalar, a list or a dictionary. Methods for R environments and named lists produce a dictionary directly.

R itself has no scalars, but `objectAsJSON()` interprets vectors of length `1` as scalars in the server language for the types that JSON can represent. This will likely be the right strategy most of the time, but to suppress scalars (e.g., because the server language function requires an array for this argument), pass the object to the interface through the call:

```
noScalar(object)
```

The `noScalar()` function uses an internal convention to force the string produced to be a JSON list.

Other objects will use the default `objectAsJson()` method, unless a particular interface package or an application has added more methods. The default method distinguishes two cases: basic data types in R and all objects with more structure,

including formal classes, informal S3 classes and vector structures such as matrix and array.

Here are some examples to illustrate:

```
> prototype <- ev$prototypeObject
> objectAsJSON(1:3, prototype)
[1] "[1,2,3]"
> objectAsJSON(list(1,2,3), prototype)
[1] "[ 1.0, 2.0, 3.0 ]"
> objectAsJSON(list(a=1,b=2,c=3), prototype)
[1] "{ \"a\" : 1.0, \"b\" : 2.0, \"c\" : 3.0 }"
> objectAsJSON(1, prototype)
An object of class "JSONScalar"
[1] "1.0"
> objectAsJSON(noScalar(1), prototype)
[1] "[ 1.0 ]"
```

Basic data types in JSON

Basic data types in R include vectors of various flavors, special types for representing the language and functions, class `"NULL"`, and some specialized types such as external references, byte-code and others.

The conversion to JSON creates a list for those vector types that map individual values into JSON scalars: `"numeric"`, `"integer"`, `"logical"` and `"character"`, with some special consideration for missing values. Essentially everything else is sent in the form of an explicitly identified R class. The representation of the class tries to retain all the information in the original data, encoded in some suitable form. Whether and how this information can be used is then up to the server language software.

Numeric data in JSON includes floating-point and integer-like notation. The format does not explicitly differentiate integers from numeric (float). The conversion in XR extends JSON notation so that, for example, the integer R value `1L` is translated as `"1"` but the numeric `1` is `"1.0"`, where JSON makes no distinction. All values in `"numeric"` vectors for conversion are adjusted by appending `".0"` to integral values:

```
> objectAsJSON(1:4) # integer
[1] "[1,2,3,4]"
> objectAsJSON(1:4+0.) # numeric
[1] "[1.0,2.0,3.0,4.0]"
```

For floating point numbers, JSON does not include features of the floating-point standard for not-a-number (`NaN`) and for infinite values (`Inf` and `-Inf` in R). What are referred to as the "JavaScript extensions" to the notation do provide these. The XR conversion includes these in its output; the JSON modules in different server languages may or may not cooperate (yes in Python; no in Julia).

A more general conversion problem is that R has the concept of a missing value, `NA`, for several basic types. In numeric data, this can usually be substituted by the standard's `NaN`, but most server languages will have no equivalent for other basic data types. In this case, XR follows jsonlite in substituting a character string `"NA"`. That is likely to cause exceptions on the server side, but no general, acceptable solution exists.

Other types have no direct analogue in JSON, including the types for function definition, language objects and the specialized non-vector types, such as external pointer. Type `NULL` is sent as the approximate JSON equivalent `"null"`; class `"name"` (type `"symbol"`) is transmitted as its string value.

Objects from the remaining types are either deparsed or serialized to obtain what should be an equivalent character form. The resulting data is then enclosed in the explicit R representation used for S4 classes; that is, dictionaries with special element `".RClass"` (page 288). The resulting string is passed as an argument to `objectFromJSON` in the server language.

Objects without JSON equivalent also include vectors of types `"raw"` and `"complex"`. These are formatted into character vectors and once again sent in the explicit representation as dictionaries with `".RClass"`.

As an example, and to illustrate one useful way of studying the process, let's create a little vector of type `"complex"`.

```
> z <- complex(real = c(1.5, 2.5), imag = c(-1.,1.))
```

To see the conversion process, we can call `objectAsJSON()` and then parse the result, as the server language would, but using the `fromJSON()` function in package jsonlite:

```
> zJSON <- objectAsJSON(z)
> jsonlite::fromJSON(zJSON)
$.RClass
[1] "complex"

$.type
[1] "complex"

$.package
```

```
[1] "methods"

$.extends
[1] "complex" "vector"

$.Data
[1] "1.5-1i" "2.5+1i"
```

While "complex" vectors are not formal classes, this representation is what they would look like if they were: ".Data" is the pseudo-slot that can always be used to refer to the data part of a vector structure.

Having an explicit dictionary representation of an object from an R class allows the server side of the interface code to include methods that interpret the R object in whatever form is suitable.

Methods for sending objects

The conversions are customized by methods, either for asServerObject() or for objectAsJSON() if the method only needs to alter the standard representation as a JSON string. Methods in the XR package for these functions are language-independent, with only the object argument in the signature. Methods in interface packages are likely to be specialized to that language. For example, conversions are specialized for sending "array" objects to Julia by:

```
setMethod("asServerObject", c("array", "JuliaObject"),
          function(object, prototype) {
              data <- asServerObject(as.vector(object), prototype)
              dims <- paste(dim(object), collapse = ",")
              value <- gettextf("reshape(%s, %s)", data, dims)
              value
          })
```

This uses the Julia function reshape() to convert a one-way array to the appropriate multi-way form.

The method recalls asServerObject() to convert the "vector" of data and inserts the text for this conversion into the expression being constructed. The conversion of a vector to Julia produces a one-way array, suitable as the argument to reshape().

Another detail in this example is worth noting. Julia requires each element of dim() to be a separate argument to reshape(). The call to paste() in R constructs this variable length call in Julia. The "dim" slot does not turn into

a field in Julia but into part of the expression. Constructing the server language expression as a string allows this sort of flexibility.

The general conversion technique based on JSON is convenient and facilitates our goal of supporting arbitrary computations. But it may not be the best choice when the data involved are large or have a structure quite different from typical R objects. One may then prefer to use some special-purpose conversion mechanism available both in R and in the server language.

XML data, as used in the shakespeare examples, illustrates the need for special treatment. XML is itself a standard for data representation. All the usual server languages, and R itself, have modules to deal with such data and one or more internal representations for it. The obvious conversion mechanism is to write and read XML itself as the intermediate form. Encoding an XML object via the general form would be difficult to define, inefficient and potentially inaccurate for complex examples.

In the example on page 280, a proxy for the `parse()` function in the Python module converted XML on a file into an `"ElementTree"` object in Python. The R package XML has several classes for representing such data. If an application is producing XML objects in R, it might like to send the objects to Python. This would be implemented by a method for `asServerObject()`. The `object` to be converted would be from a suitable class that can be saved to a file, such as `"XMLInternalDocument"` in the XML package. The prototype class in the signature can be left as a general `"PythonObject"` if we want to use this conversion by default when sending XML to Python.

A simple version of the method creates a temporary file, saves the R object to it, and constructs a Python expression to parse the file:

```
setMethod("asServerObject",
    c("XMLInternalDocument", "PythonObject"),
          function(object, prototype) {
              file <- tempfile()
              XML::saveXML(object, file)
              gettextf("xml.etree.ElmentTree.parse(%s)",
                       asServerObject(file, prototype))
          })
```

`asServerObject()` is recalled to convert the string in `file` to a string in the server language. In practice, a somewhat better method would be needed.

The R object was written to a temporary file. This file has to stay around until the corresponding Python method reads it, but then it should be removed. Rather than the `parse` function in the `"ElementTree"` module, we should write a specialized Python function, say `XMLFromR`, that ends by removing the file.

There is an important limitation to `asServerObject()` methods that replace the JSON-based set of methods for conversion. The approach is fine to convert an object from a given class. But if the computation is converting an object that *contains* such an object, it is the method for the containing object that counts. If that method constructs a JSON string, then our method for `asServerObject()` will not be called; instead, the conversion is likely to call `objectAsJSON()` with the XML data as argument. That will happen if our XML object is an element in an ordinary list or is a slot in an object that will be converted using the general representation discussed above.

What to do? A reasonable approach might be something like this:

- Specialized methods replacing the standard approach are mainly useful for seriously large and/or complex objects. These should usually be converted once and then utilized in the server language. The strategy would then be to construct containers and higher-level objects in that language.

- Where it does make sense to convert higher-level objects all at once, give these their own class, with an appropriate `asServerObject()` method for conversion. A list of XML objects could have a class `"listOfXML"` with an `asServerObject()` method.

- If the decision is that the JSON-based approach is unacceptable, rather than just slightly sub-optimal, include a method for `objectAsJSON()` that signals an error.

The method for a higher-level object containing one or more of our specially converted classes will often work best through a custom function in the server language. For example, suppose the server language had a function `listOf()` taking an arbitrary number of arguments and making a list-like object with each of the arguments as an element. Then the essential computation in a `"listOfXML"` method would be

```
calls <- sapply(object, function(x) asServerObject(x, prototype))
paste0("listOf(", paste(calls, collapse = ", "), ")")
```

The first line creates a character vector with the individual expressions to convert the elements, such as calls to our `XMLFromR()` function, the second line constructs a single, potentially large call to `listOf()`.

See the treatment of multi-way arrays in Julia, in Section 15.6, for another example.

Getting a server object

The value returned from the server is always a string. This is first parsed using JSON notation.

Since JSON has no typed arrays, an R vector of a basic type sent and then returned using JSON notation will become a list. The user (or the specializing server language interface package) can avoid this by setting the field `"simplify"` in the evaluator to `TRUE`. Any list object in JSON whose elements are all basic scalars will then be turned into an R vector of the necessary type. The XR package defaults to `simplify = FALSE` so as not to interfere with specializing strategies.

The R object obtained from parsing the JSON string is then passed as the `object` argument to the generic function:

```
asRObject(object, evaluator)
```

The value of the call will be the actual object returned from the server. The default method just returns `object`, but custom methods can do anything that suits the application. The `evaluator` argument allows the method signature to specify computations for a specific interface class. Unlike an `initialize()` method, there is no requirement that `asRObject()` returns any particular class of object, or that it always returns the same class.

The direct conversion procedure can produce an object from an arbitrary R class, using the explicit dictionary representation illustrated starting on page 288. To do so, the function or method in the server language needs to construct a dictionary object, or anything that will be returned to R as such. The object must have a `".RClass"` element with the appropriate class name and other elements with the names of the slots in the R class definition.

After the conversion produces `object` from a class through the special dictionary representation, XR interface computations will call `asRObject()` for this object.

For the computation to produce a valid object in the target R class, each of the slots must be converted to an object from the class specified for that slot in the R class definition. If the target class for some slots is inconvenient to generate from the server language, it may be best to define an intermediate R class with slots that *are* convenient and that contain the essential information to define the target class or classes via computations in R. The mechanism to return typed R vectors is an example.

Class `"vector_R"`

There is considerable variation among server languages in how they treat objects analogous to `"vector"` classes in R. Some languages have as basic data types only

something similar to "list" in R: array-like objects with arbitrary data in each element. This is also the case for the basic JSON notation underlying the direct conversion procedure in XR.

Conversion of a general list to a chosen vector type is provided in the XR package by the class "vector_R" with slots "type" and "data":

```
setClass("vector_R",
         slots = c(data = "vector", type = "character",
                   missing = "vector"))
```

The important distinction is that the object returned from the server side can have a "data" slot that is a list.

To return a vector of a chosen type, the server side must provide a suitable mechanism to generate an object from the "vector_R" class. XRPython and XRJulia have a corresponding server-language function vector_R(). Like JSON, Python does not have typed vectors, so an application package will need to be explicit when wanting to have the R object be a vector of a specific type, by calling vector_R(). But all the arguments can be standard Python objects—lists and character strings—so no exotic computations are needed.

Julia does have typed single- and multi-dimensional arrays. Conversion to R via the vector_R() class is automatic, with no action on the part of the application needed. To implement the return of suitable "vector_R" objects, the XRJulia interface uses Julia's own generic functions and functional OOP. In effect, vector_R() implements the method for one-way arrays of the generic function toR() in XRJulia, which transforms an arbitrary Julia object into the form needed for transmission to R. See Chapter 15 for the details.

In addition to providing type information, the class "vector_R" provides explicit information about missing values. Neither JSON nor most server languages have a uniform mechanism for indicating missing values from vectors of different types. Standard floating point representation has a somewhat analogous concept of NaN for not-a-number, but integer, logical or character string data has no similar mechanism.

Objects from "vector_R" have a slot "missing", which is interpreted as a vector of indices for all elements of the vector that should be treated as NA. As with the type information, missing value information can be generated automatically in R and passed to the server language or can be explicitly computed and returned to R by constructing the R representation, including the "missing" slot.

Objects for exceptions

Another important application for returning an explicit form of an R class comes when handling errors and other exceptions. The XR design requires that an in-

terface package for a particular language return an object from a subclass of class "InterfaceCondition" when an error or other exception arises. The subclasses include "InterfaceError" and "InterfaceWarning", expected to be treated as errors and warnings in R. There can also be any specialized subclasses that make sense for a particular language or application. All the classes will, as always, inherit the slots of their superclass, "InterfaceCondition":

```
setClass("InterfaceCondition",
  slots = c(message = "character", value = "ANY",
            expr = "character", evaluator = "Interface"))
```

The "message" slot is the descriptive message for the error or other exception, as provided by the server language (one hopes). The "expr" slot is the expression provided by the R side for evaluation. These are both character strings.

The "value" slot is the value of the expression that would have been returned in the absence of the exception. For an error, value is ignored. In the case of a warning exception, it will normally be returned after the user is notified of the exception. From a data conversion perspective, the value slot requires nothing special, but is handled exactly as the result of the computation would have been without the exception.

The "evaluator" field has the evaluator object being used when the exception occurred, and is inserted by the R side of the interface. The XR function doCondition() is expected to be called from the $ServerEval() method of a particular interface. This is another generic function. Its default mechanism will signal the corresponding R condition, with information about the interface added to the message in the object. Applications can further specialize condition handling by defining new classes that extend "InterfaceCondition".

Methods for doCondition(object) can specialize the handling. To inherit the actual condition behavior, the method should end with

```
callNextMethod()
```

In particular, this will eventually call stop() or warning() for conditions that extend "InterfaceError" or "InterfaceWarning".

Before this point, asRObject() will be called with the "InterfaceCondition" object as an argument. If the application wanted to do something completely unrelated to R condition handling, it could define a method for asRObject() that returned a different object, perhaps conditional on the circumstances.