Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. *http://www.jstatsoft.org/*

Supplement to: RKWard – A Comprehensive Graphical User Interface and Integrated Development Environment for Statistical Analysis with R

Stefan Rödiger[‡,§]
Lausitz University of Applied Sciences & Charité

Thomas Friedrichsmeier[‡,§]
Ruhr-University Bochum

Prasenjit Kapat
The Ohio State University

Meik Michalke
Heinrich Heine University Düsseldorf

Abstract

RKWard is a GUI to R with the objective to provide a portable and extensible R interface for both basic and advanced statistical and graphical analysis. This supplement discusses in detail technical aspects of **RKWard** including the usage of the **KDE**/**Qt** software libraries which are the base of **RKWard**. Statistical procedures and plots are implemented extendable plugin architecture in **RKWard**. This plugin architechture is based on ECMAScript (JavaScript), R, and XML. The general design is described and its application is exemplified on a t-test.
Disclaimer: This supplement was not part of the peer-review process of the Journal of Statistical Software.

[‡]Equal contribution, [§]Corresponding authors

Keywords: cross-platform, GUI, integrated development environment, plugin, R.

1. Technical design

This supplemt will give a compact overview of the key aspects of **RKWard**'s development process and technical design. We will give slightly more attention to the details of the plugin framework (see origial paper for further information) used in **RKWard**, since this is central to the extensibility of **RKWard**.

1.1. Development process

***RKWard** core and external plugins*

Newly developed plugins are placed in a dedicated plugin map file. Plugins in this map are not visible to the user by default, but need to be enabled manually. Once the author(s) of a plugin announces that they consider it stable, the plugin is subjected to a review for correctness, style, and usability. The review status is tracked in the project wiki. Currently at least one positive review is needed before the plugin is allowed to be made visible by default, by moving it to an appropriate plugin map.

The current development version adds support for downloading additional sets of plugins, which are neither officially included nor supported by the **RKWard** developers, from the internet.

Automated testing

A second requirement for new plugins is that each plugin must be accompanied by at least one automated test. The automated testing framework in **RKWard** consists of a set of R scripts which allow to run a plugin with specific GUI settings, automatically[1]. The resulting R code, R messages, and output are then compared to a defined standard. Automated tests are run routinely after changes in the plugin infrastructure, and before any new release.

The automated testing framework is also useful in testing some aspects of the application which are not implemented as plugins, but this is currently limited to very few basic tests.

1.2. Asynchronous command execution

One central design decision in the implementation of **RKWard** is that the interface to the R engine operates asynchronously. The intention is to keep the application usable to a high degree, even during the computation of time-consuming analysis. For instance, while waiting for the estimation of a complex model to complete, the user should be able to continue to use the GUI to prepare the next analysis. Asynchronous command execution is also a prerequisite for an implementation of the plot-preview feature (see Section **??**). Commands generated from plugins or user actions are placed in queue and are evaluated in a separate thread in the order they were submitted[2]. The asynchronous design implies that **RKWard** avoids relying on the R engine during interactive use. This is one of several reasons for the use of ECMAScript in plugins, instead of scripting using R itself (see Sections 1.4 and

[1] In the current development version, the scripts have been converted into a proper R package.

[2] It is possible, and in some cases necessary, to enforce a different order of command execution in internal code. For instance, **RKWard** makes sure that no user command can potentially interfere while **RKWard** is loading the data of a `data.frame` for editing.

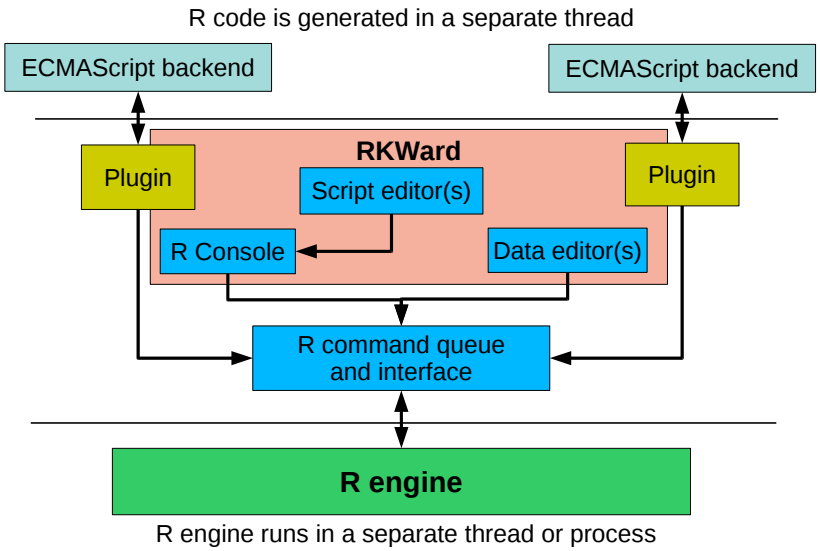

Figure 1: Technical design of **RKWard**. Only a few central components are visualized. All communication with the R engine is passed through a single interface living in the main application thread. The R engine itself runs in a separate thread (or in a separate process). Separate threads are also used to generate R code from plugins.

??). A further implication is that **RKWard** avoids querying information about the existence and properties of objects in R interactively. Rather, **RKWard** keeps a representation of R objects and their basic properties (e. g., class and dimensions), which is used for the workspace browser (Section **??**), object name completion, function argument hinting, and other places. The object representation includes objects in all environments in the search path, and any objects contained within these environments in a hierarchical tree[3]. The representation of R objects is gathered pro-actively. This has a notable impact on performance when loading packages. Specifically, objects which would usually be "lazy loaded" only when needed (see Ripley 2004) are accessed in order to fetch information on their properties. This means the data has to be loaded from disk; however, the memory is freed immediately after fetching information on the object. Additionally, for packages with extremely large number of objects, **RKWard** provides an option to exclude specific packages from scanning the object structures.

A further side-effect of the asynchronous threaded design is that there is inherently a rather clear separation between the GUI code and the code making direct use of the R application programming interface (API) (see also Figure 1). In the current development version, the evaluation of R commands has even been moved into a separate process. Therefore in future releases it could be made possible to run GUI and R engine on different computers.

1.3. Object modification detection

RKWard allows the user to run arbitrary commands in R at any time, even while editing a `data.frame` or while selecting objects for analysis in a GUI dialog. Any user command

[3] Currently, environments of functions or formulas are not taken into account.

can potentially add, modify, or remove objects in R. **RKWard** tries to detect such changes in order to always display accurate information in the workspace browser, object selection lists, and object views. Beyond that, detecting any changes is particularly important with respect to objects which are currently being edited in the data editor (which provides an illusion of in-place editing, see Section **??**). Here, it is necessary to synchronize the data between R and the GUI in both directions.

For simplicity and performance, object modification detection is only implemented for objects inside the "global environment" (including environments inside the global environment), since this is where changes are typically done. Currently, object modification detection is based on active bindings. Essentially, any object which is created in the global environment is first moved to a hidden storage environment, and then replaced with an active binding. The active binding acts as a transparent proxy to the object in the storage environment, which registers any write-access to the object[4].

The use of active bindings has significant performance implications when objects are accessed very frequently. This is particularly notable where an object inside the global environment is used as the index variable in a loop, as illustrated by the following example. When control returns to the top level prompt, after the first assignment, `i` will become subject to object modification detection (i. e., it will be wrapped into an active binding). The subsequent `for` loop will then run slow.

```
R> i <- 1
R> for (i in 1:100000) i + i
```

In contrast, in the following example, `i` is a local object, and will not be replaced by an active binding. Therefore the loop will run approximately as fast as in a plain R session:

```
R> f <- function () {
R+    i <- 1
R+    for (i in 1:100000) i + i
R+ }
R> f ()
```

Future versions of **RKWard** will try to avoid this performance problem. One approach that is currently under consideration is to simply perform a pointer comparison of the `SEXP` records of objects in global environment with their copies in a hidden storage environment. Due to the implicit sharing of `SEXP` records (R Development Core Team 2010b,a), this should provide for a reliable way to detect changes for most types of R objects, with comparatively low memory and performance overhead. Special handling will be needed for environments and active bindings.

1.4. Choice of toolkit and implementation languages

In addition to R, **RKWard** is based on the **KDE** libraries, which are in turn based on **Qt**, and implemented mostly in C++. Compared to many competing libraries, this constitutes

[4] This is similar to the approach taken in the **trackObjs** package (Plate 2009).

a rather heavy dependency. Moreover, the **KDE** libraries are still known to have portability issues especially on Mac OS X, and to some degree also on the Microsoft Windows platform.

The major reason for choosing the **KDE** and **Qt** libraries was their many high level features which have allowed **RKWard** development to make quick progress despite limited resources. Most importantly, the **KDE** libraries provide a full featured text editor (Cullmann n.d.) as a component which can be seamlessly integrated into a host application using the KParts technology (Faure 2000). Additionally, another KPart provides HTML browsing capabilities in a similarly integrated way. The availability of **KWord** (KOffice.Org 2010) as an embeddable KPart might prove useful in future versions of **RKWard**, when better integration with office suites will be sought.

Another technology from the **KDE** libraries that is important to the development of **RKWard** is the "XMLGUI" technology (Faure 2000). This is especially helpful in providing an integrated GUI across the many different kinds of document windows and tool views supported in **RKWard**.

Plugins in **RKWard** rely on XML[5] and ECMAScript[6] (see Section **??**). XML is not only well suited to describe the layout of the GUI of plugins, but simple functional logic can also be represented (see also Visne *et al.* 2009). ECMAScript was chosen for the generation of R commands within plugins, in particular due to its availability as an embedded scripting engine inside the **Qt** libraries. While at first glance R itself would appear as a natural choice of scripting language as well, this would make it impossible to use plugins in an asynchronous way. Further, the main functional requirement in this place is the manipulation and concatenation of text strings. While R provides support for this, concatenating strings with the +-operator, as available in ECMAScript, allows for a much more readable way to perform such basic text manipulation.

1.5. On-screen graphics windows

Contrary to the approach used in **JGR** (Helbig *et al.* 2010), **RKWard** does not technically provide a custom on-screen graphics device. **RKWard** detects when new graphics windows are created via calls to `X11()` or `windows()`. These windows are then "captured" in a platform dependent way (based on the XEmbed (Ettrich and Taylor 2002) protocol for **X11**, or reparenting for the Microsoft Windows platform). An **RKWard** menu bar and a toolbar is then added to these windows to provide added functionality. While this approach requires some platform dependent code, any corrections or improvements made to the underlying R native devices will automatically be available in **RKWard**.

A recent addition to the on-screen device is the "plot history" feature which adds a browsable list of plots to the device window. Since **RKWard** does not use a custom on-screen graphics device, this feature is implemented in a package dependent way. For example, as of this writing, plotting calls that use either the "standard graphics system" or the "**lattice** system" can be added to the plot history; other plots are drawn but not added. The basic procedure is to identify changes to the on-screen canvas and record the existing plot before a new plot wipes it out. A single global history for the recorded plots is maintained which is used by all the on-screen device windows. This is similar to the implementation in Rgui.exe (Microsoft Windows), but unlike the one in Rgui.app (Mac OS X). Each such device window points to a

[5] http://www.w3.org/XML/
[6] http://www.ecmascript.org/

position in the history and behaves independently when recording a new plot or deleting an existing one.

The **lattice** system is implemented by inserting a hook in the `print.lattice()` function. This hook retrieves and stores the `lattice.status` object from the `lattice:::.LatticeEnv` environment, thereby making `update()` calls on trellis objects transparent to the user. Any recorded trellis object is then replayed using `plot.lattice()`, bypassing the recording mechanism. The standard graphics system, on the other hand, is implemented differently because the hook in `plot.new()` is ineffective for this purpose. A customized function is overloaded on `plot.new()` which stores and retrieves the existing plot, essentially, using `recordPlot()` and replays them using `replayPlot()`.

The actual plotting calls are tracked using appropriate `sys.call()` commands in the hooks. These call strings are displayed as a drop-down menu on the toolbar for non-sequential browsing (see Figure **??**) providing a very intuitive browsing interface unlike the native implementations in `windows()` and `quartz()` devices.

1.6. Internationalization

Currently strings in the main application are translated to varying extents in Czech (cs), Catalan (ca), Spanish (es), German (de), Chinese (zh_CN), Turkish (tr), Polish (pl), Italian (it), French (fr), Greek (el), and Danish (da). Translatable strings are to be found under po/*.po in the sources. These files can conveniently be edited with front-ends like Lokalize[7].

Plugins and help pages in **RKWard** are not translatable at the time of this writing. While it will be technically possible to include the respective strings in message catalogs, this is currently not implemented in **RKWard**. Similarly, any output generated by R functions defined for **RKWard** is currently not translatable. Again, however, there is no technical barrier with respect to internationalization of R code, as discussed by Ripley (2005), and it is planned to make **RKWard** fully translatable in future versions.

[7] http://i18n.kde.org/tools/

2. Extending RKWard – an example of creating a plugin

As discussed in Section **??**, plugins in **RKWard** are defined by four separate files (Figure **??**). To give an impression of the technique, this section shows (portions of) the relevant files for a plugin that provides a simple dialog for a t-test. For brevity, the help-file is omitted.

2.1. Defining the menu hierarchy

A so called ".pluginmap" file declares each plugin, and, if appropriate, defines where it should be placed in the menu hierarchy. Usually each .pluginmap file declares many plugins. In this example we only show one, namely, a two variable t-test (see Figure 2). The pluginmap (`<!DOCTYPE rkpluginmap>`) gives a unique identifier ("id"), the location of the GUI description ("file"), and the window title ("label"). The menu layout is defined in a hierarchical structure by nesting `<menu>` elements to form toplevel menus and submenus. Menus with the same "id" are merged across .pluginmap files. Moreover, the position within the menu can be explicitly defined (attribute "index"). This might be required if the menu entries are to be ordered non-alphabetically.

```
<!DOCTYPE rkpluginmap>
<document base_prefix="" namespace="rkward">
  <components>
    <component type="standard" id="t_test_two_vars"
          file="demo_t_test_two_vars.xml" label="Two Variable t-test" />
  </components>
  <hierarchy>
    <menu id="analysis" label="Analysis" index="4">
      <menu id="means" label="Means" index="4">
        <menu id="ttests" label="t-Tests">
          <entry component="t_test_two_vars" />
        </menu>
      </menu>
    </menu>
  </hierarchy>
</document>
```

2.2. Defining the dialog GUI

The main XML file of each plugin defines the layout and behavior of the GUI, and references the ECMAScript file that is used for generating R code from GUI settings and the help file (not included in this paper).

GUI logic can be defined directly in the XML file (the `<logic>` element). In this example, the "Assume equal variances" checkbox is only enabled for paired sample tests. Optionally, GUI behavior can also be scripted in ECMAScript.

The XML file defines the t-test plugin (`<!DOCTYPE rkplugin>`) to be organized in two tabs (Figure **??**A). On the first tab, two variables can be selected (`<varslot .../>`). These are set to be "required", i. e., the "Submit" button will remain disabled until the user has made a valid selection for both. The second tab includes some additional settings like the confidence level (default 0.95).

```
<!DOCTYPE rkplugin>
<document>
```

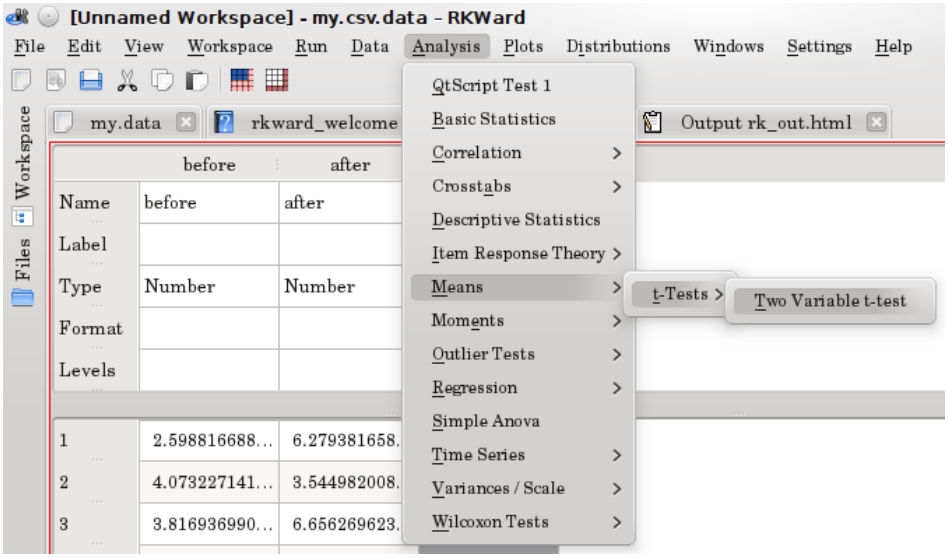

Figure 2: Generated menu structure as defined by the plugin map.

```
<code file="demo_t_test_two_vars.js"/>
<help file="demo_t_test_two_vars.rkh"/>

<logic>
  <connect client="varequal.enabled" governor="paired.not"/>
</logic>

<dialog label="Two Variable t-Test">
  <tabbook>
    <tab label="Basic settings" id="tab_variables">
      <row id="basic_settings_row">
        <varselector id="vars"/>
        <column>
          <varslot type="numeric" id="x" source="vars" required="true"
            label="compare"/>
          <varslot type="numeric" id="y" source="vars" required="true"
            label="against"/>
          <radio id="hypothesis" label="using test hypothesis">
            <option value="two.sided" label="Two-sided"/>
            <option value="greater" label="First is greater"/>
            <option value="less" label="Second is greater"/>
          </radio>
          <checkbox id="paired" label="Paired sample" value="1" value_unchecked="0" />
        </column>
      </row>
    </tab>
    <tab label="Options" id="tab_options">
      <checkbox id="varequal" label="assume equal variances" value="1"
        value_unchecked="0"/>
      <frame label="Confidence Interval" id="confint_frame">
        <spinbox type="real" id="conflevel" label="confidence level" min="0" max="1"
          initial="0.95"/>
        <checkbox id="confint" label="print confidence interval" value="1"
          checked="true"/>
```

```
        </frame>
        <stretch/>
      </tab>
    </tabbook>
  </dialog>
</document>
```

2.3. Generating R code from GUI settings

A simple ECMAScript script is used to generate R code from GUI settings (using `echo()` commands)[8]. Generated code for each plugin is divided into three sections: "Preprocess", "Calculate", and "Printout", although each may be empty.

```
var x;
var y;
var varequal;
var paired;

function preprocess () {
  x = getValue ("x");
  y = getValue ("y");

  echo ('names <- rk.get.description (' + x + ", " + y + ')\n');
}

function calculate () {
  varequal = getValue ("varequal");
  paired = getValue ("paired");

  var conflevel = getValue ("conflevel");
  var hypothesis = getValue ("hypothesis");

  var options = ", alternative=\"" + hypothesis + "\"";
  if (paired) options += ", paired=TRUE";
  if ((!paired) && varequal) options += ", var.equal=TRUE";
  if (conflevel != "0.95") options += ", conf.level=" + conflevel;

  echo ('result <- t.test (' + x + ", " + y + options + ')\n');
}

function printout () {
  echo ('rk.header (result\$method, \n');
  echo ('  parameters=list ("Comparing", paste (names[1], "against", names[2]),\n');
  echo ('  "H1", rk.describe.alternative (result)');
  if (!paired) {
    echo (',\n');
    echo ('  "Equal variances", "');
    if (!varequal) echo ("not");
    echo (' assumed"');
  }
  echo ('))\n');
  echo ('\n');
  echo ('rk.results (list (\n');
  echo ('  \'Variable Name\'=names,\n');
```

[8] See Figure ??A) for code generated in this example.

```
  echo ('  \'estimated mean\'=result\$estimate,\n');
  echo ('  \'degrees of freedom\'=result\$parameter,\n');
  echo ('  t=result\$statistic,\n');
  echo ('  p=result\$p.value');
  if (getValue ("confint")) {
    echo (',\n');
    echo ('  \'confidence interval percent\'=(100 * attr(result\$conf.int, "conf.level")),\n');
    echo ('  \'confidence interval of difference\'=result\$conf.int ');
  }
  echo ('))\n');
}
```

References

Cullmann C (n.d.). ***KatePart***. URL http://kate-editor.org/about-katepart/.

Ettrich M, Taylor O (2002). *XEmbed Protocol Specification Version 0.5*. URL http://standards.freedesktop.org/xembed-spec/xembed-spec-latest.html.

Faure D (2000). "Creating and Using Components (KParts)." In D Sweet (ed.), ***KDE** 2.0 Development*. Sams, Indianapolis, IN, USA.

Helbig M, Urbanek S, Fellows I (2010). ***JGR**: Java Gui for R*. R package version 1.7-3, URL http://CRAN.R-project.org/package=JGR.

KOfficeOrg (2010). ***KWord***. URL http://www.koffice.org/kword/.

Plate T (2009). ***trackObjs**: Track Objects*. R package version 0.8-6, URL http://CRAN.R-project.org/package=trackObjs.

R Development Core Team (2010a). *R Internals*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-14-3.

R Development Core Team (2010b). *Writing R Extensions*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-11-9.

Ripley BD (2004). "Lazy Loading and Packages in R 2.0.0." *R News*, **4**(2), 2–4. URL http://CRAN.R-project.org/doc/Rnews/.

Ripley BD (2005). "Internationalization Features of R 2.1.0." *R News*, (1), 2–7. ISSN 1609-3631, URL http://CRAN.R-project.org/doc/Rnews/.

Visne I, Dilaveroglu E, Vierlinger K, Lauss M, Yildiz A, Weinhaeusel A, Noehammer C, Leisch F, Kriegner A (2009). "**RGG**: A general GUI Framework for R scripts." *BMC Bioinformatics*, **10**(1), 74. ISSN 1471-2105. doi:10.1186/1471-2105-10-74. URL http://www.biomedcentral.com/1471-2105/10/74.

Affiliation:

Stefan Rödiger
Lausitz University of Applied Sciences
Department of Bio-, Chemistry and Process Engineering
AND
Center for Cardiovascular Research (CCR)
Charité, Germany
E-mail: stefan_roediger@gmx.de
E-mail: rkward-devel@lists.sourceforge.net

Journal of Statistical Software http://www.jstatsoft.org/
published by the American Statistical Association http://www.amstat.org/
Volume VV, Issue II *Submitted:* yyyy-mm-dd
MMMMMM YYYY *Accepted:* yyyy-mm-dd