Plummer (2004) just released 0.50 of JAGS. Quote from Martyn Plummer: "JAGS is Just Another Gibbs Sampler - an alternative engine for the BUGS language that aims for the same functionality as classic BUGS. JAGS is written in C++ and licensed under the GNU GPL. It was developed on Linux and also runs on Windows." The functions in package `rbugs` can also be used to prepare files for JAGS. I am looking forward to seeing the growth of JAGS.

I also tried using R for Windows through Wine. It worked last winter with Wine 20031016, but is not working with Wine 20040408 now. Unfortunately, since my Wine 20040408 was compiled after my system has been recently upgraded to Red Hat Workstation 3.0, I cannot tell which change has caused it.

Bibliography

Gelman, A. (2004), "bugs.R: functions for running WinBugs from R," http://www.stat.columbia.edu/~gelman/bugsR/. 19

Plummer, M. (2003), "Using WinBUGS under Wine," http://calvin.iarc.fr/bugs/wine/. 19, 20

Plummer, M. (2004), "JAGS version 0.50 manual," http://www-fis.iarc.fr/~martyn/software/jags/. 20

Plummer, M., Best, N., Cowles, K., and Vines, K. (1996), "coda: Output analysis and diagnostics for MCMC," http://www-fis.iarc.fr/coda/. 19

Rice, K. (2002), "EmBedBUGS: An R package and S library," http://www.mrc-bsu.cam.ac.uk/personal/ken/embed.html. 19

Smith, B. (2004), "boa: Bayesian Output Analysis Program for MCMC," http://www.public-health.uiowa.edu/boa. 19

Spiegelhalter, D. J., Thomas, A., Best, N. G., and Gilks, W. (1996), *BUGS: Bayesian inference Using Gibbs Sampling, Version 0.5, (version ii)* http://www.mrc-bsu.cam.ac.uk/bugs. 19

Sturtz, S. and Ligges, U. (2004), "R2WinBUGS: Running WinBUGS from R," http://cran.r-project.org/src/contrib/Descriptions/R2WinBUGS.html. 19

Wine (2004), "Wine," http://www.winehq.org. 19

Jun Yan
University of Iowa, U.S.A.
jyan@stat.uiowa.edu

R Package Maintenance

Paul Gilbert

Introduction

Quality control (QC) for R packages was the feature that finally convinced me to maintain R packages and also run them in S, rather than the reverse. A good QC system is essential in order to contain the time demands of maintaining many packages with interdependencies. It is necessary to have quick, easy, reliable ways to catch problems. This article explains how to use the R package QC features (in the "tools" package by Kurt Hornik and Friedrich Leisch) for ongoing maintenance and development, not just as a final check before submitting a package to CRAN. This should be of interest to individuals or organizations that maintain a fairly large code base, for their own use or the use of others.

The main QC features for an R package check that:

- code in package directory R/ is syntactically correct
- code in package directory tests/ runs and does not crash or stop()
- documentation is complete and accurate in several respects
- examples in the documentation actually run
- code in package directory demo/ runs
- vignettes in package directory inst/doc/ run

These provide several important features for package maintenance. Developers like to improve code, but documentation updates are often neglected. A simple method to identify necessary documentation changes means documentation maintenance is (almost) painless. The QC tools can be used to help flag when documentation changes are necessary. They also ensure that packaged code can be quickly tested to ensure it works with a new version of R (or a new compiler, or a new operating system, or a new computer). The system explained below also helps check dependencies among functions in different packages, easing development by quickly identifying changes that break code in other packages.

The system described here uses the QC features in R in conjunction with the *make* utility. It checks code and documentation of multiple packages, automatically when changes to source files imply that these checks need to be done. The key is a good 'Makefile' with interdependencies properly identified. It should be possible to run this system with a relatively small investment in "local setup" for a different set of packages, perhaps only a couple of hours. This presumes a certain familiarity with *make*. For a complicated set of package, a somewhat larger amount of time may be necessary in order to understand interdependencies among packages. If your packages are not well organized then a much larger time investment will be necessary, but well worthwhile.

Make

This is not a tutorial about *make*, but a rudimentary explanation is given in order to make the remainder of the artical accessible to a wider audience. Briefly, the *make* utility uses targets (rules) which may have prerequisites (other targets or files). These are indicated in a file typically called 'Makefile'. This works most easily when a target is the name of a file generated from another file, for example, a compiled target file called foo generated from a C code prerequisite file called foo.c. Make determines that a target is out of date and must be re-generated if the file timestamp for the target is older than the timestamp of any prerequisite. This is recursive, so a target must be re-generated (or "re-made") if it depends on a target, that depends on a target, ..., that depends on a file that is newer. Properly mapping out the dependencies in a 'Makefile' eventually saves an enormous amount of time, because a change in a source file only necessitates re-generating dependent targets. To understand correctly how this is used in the context of R package maintenance, it is important to recognize that "re-made" does not mean simply that code (or documentation) is checked to be syntactically correct, it also means a number of tests are completed to insure it works correctly.

Make and R Package QC

In order to implement the system for R package maintenance, one critical simplifying assumption is that code testing does not depend on documentation testing. This may seem obvious, but it has the implication that examples in the documentation are not the most important way to catch mistakes in the code. That is, there should be files in the tests/ directory of a package that will generate errors if mistakes are introduced into the code. These would typically run functions, check results against known values, and *stop()* if an error is indicated.

With this simplifying assumption it is possible to distinguish two main targets for each package: "code" and "doc." These are each aliases for several "sub-targets." The code target tests the code in a package. It may be a prerequisite for code in other packages, but the doc target in a package is never a prerequisite in other packages. This means that a change to .Rd files in the man directory, or to .R files in the demo directory, or to vignette files, will signal re-making only for the package itself, and not for other packages. Changes to code files in the R directory or files in the tests directory will signal re-making for the package, and this may imply re-making of other packages that depend on it.

As an example, I have package dse2, which depends on dse1, which depends on packages tframe and setRNG. Changes in files in tframe/R should provoke a remake of dse1 and dse2, but changes in tframe/man or tframe/inst/doc should not provoke a remake of dse1 and dse2.

The 'Makefile' line for some targets uses "R CMD check", but in most cases the targets directly use functions in library("tools"). Shell variables, doc targets, and many code targets, are common to all packages and can be specified in common files, 'Makevars' and 'Makerules', which are included into the 'Makefile' for each package. (For technical reasons it is best to have these in two files rather than one.) The key code sub-target (Rcode) has different prerequisites for each package and must thus be specified in the specific 'Makefile' for each package.

As an example, Figure 1 shows the critical part [1] of the 'Makefile' for my dse1 package, which has the packages tframe and setRNG as a prerequisites:

After first including the common variables from ../Makevars, this specifies the default target prerequisites. (Left of the colon is a target name, right of the colon is the list of prerequisites, backslash indicates line continuation.)

Packages are each in a subdirectory below a common directory, so *../tframe* refers to the relative path from the package dse1 directory to the directory for the package tframe. Some targets, like Rcode, are not naturally files, so to take advantage of the timestamp mechanism used by *make* it is necessary to create an artificial file (placed in a subdirectory referred to by the variable *FLAGS*). The critical part of the macro[2] *RchkCodeMacro* is specified in 'Makevars' in Figure 2

This checks the code using any necessary packages from the location indicated by *CHKLIBS*, which is where packages that have already been checked are installed.

As another example, some of the documentation targets are specified in 'Makerules' (in Figure 3) by

[1] The complete generic makefiles should be available in the contributed section of CRAN.

[2] The *define* feature and some other aspects of these files may be specific to GNU *make*.

```
include ../Makevars

default:    undoc checkDocFiles codoc examples latex demos \
            checkDocStyle  checkFF checkMethods checkReplaceFuns \
            Rcode checkVignettes pdfVignettes tar

Rcode: R/*.R tests/*.R LICENSE DESCRIPTION INDEX \
            ../tframe/$(FLAGS)/Rcode  ../setRNG/$(FLAGS)/Rcode
     ${RchkCodeMacro}

include ../Makerules
```

Figure 1: Makefile for `dse1`

```
define RchkCodeMacro
     ...
     R_LIBS=$(CHKLIBS)  $(RENV) R CMD check \
          --outdir=$(CURDIR)/$(TMP) --library=$(CURDIR)/$(TMP) \
          --no-vignettes --no-codoc --no-examples \
          --no-latex  $(CURDIR)
     ...
     @touch $(FLAGS)/$@
endef
```

Figure 2: *RchkCodeMacro* is specified in 'Makevars'

```
undoc checkDocFiles checkDocStyle:      man  R/*.R
     @$(MKDIR) $(TMP)
     @echo "library(tools); $@(dir='$(CURDIR)')" | R --vanilla -q >$(TMP)/$@
     @test -z "`grep 'Error' $(TMP)/$@`" || (cat $(TMP)/$@ ; exit 1 )
#       check errors from undoc and checkDocFiles
     @test -z "`grep 'Undocumented' $(TMP)/$@`" || (cat $(TMP)/$@ ; exit 1 )
     @$(MKDIR) $(FLAGS)
     @mv  $(TMP)/$@ $(FLAGS)/$@
```

Figure 3: Documentation targets in 'Makerules'

This specifies the targets undoc, checkDocFiles, and checkDocStyle, which all depend on any files in the man directory, as well as any code files R/*.R. The output from the R sessions that runs *undoc()* and *checkDocFiles()* print errors and warnings, but these do not automatically produce a shell error signal as a flag that *make* recognizes. It is possible to do this using *R* code that determines if the result should indicate an error, and sets *q(status=1)* but that is not done in this example. Instead, a test on a grep of the output is used to determine the shell error status. (This may change in the future.) If the signal does not indicate a failure (exit 1) then the output is moved to the *FLAGS* directory to indicate that the target has completed successfully.

Summary

There are trade-offs in the way R code is organized into packages. If all code is in one package then there are no package inter-dependencies, but everything must be tested after any change. Faster computers make it possible to consider this, and the *make*/QC system described here would be extra overhead and of limited value in that situation. However, more documentation and examples, along with more extensive test suites, take longer to run, and so encourage a finer breakdown into packages. In addition to this, there are two complementary reasons for organizing functions into packages. One is to limit dependencies, as much as reasonably possible, between groups of functions that are not closely related and may not often be used together. The second is to group together "kernel" functions which are tools used by several other packages. The dependencies among packages must be carefully mapped out, which forces one to think carefully about what is kernel code and what is not. These reasons for organizing code into packages may be even more important in a situation where multiple programmers or users are maintaining packages.

It is important to see that the savings in this *make*/QC system come from a few different aspects. The first is that packages of kernel code used by other packages tend to be more stable and less frequently changed than the packages that use them. If kernel packages are not changed, they do not need to be re-made. The second aspect is that dependencies among packages are in the code, not in the documentation. Thus documentaion changes imply only that the documentation for that particular package needs to be checked. The aspect that results in the most important savings, however, is that the need for many documentation changes are flagged immediately, while you still remember what that marvelous change in the code really did.

Acknowledgments

I am grateful to Kurt Hornik for many helpful explanations and comments.

Paul Gilbert,
Department of Monetary and Financial Analysis,
Bank of Canada,
234 Wellington St.,
Ottawa, Canada, K1A 0G9
pgilbert@bank-banque-canada.ca

Changes in R

by the R Core Team

User-visible changes in 2.0.0

- The stub packages from 1.9.x have been removed: the `library()` function selects the new home for their code.

- 'Lazy loading' of R code has been implemented, and is used for the standard and recommended packages by default. Rather than keep R objects in memory, they are kept in a database on disc and only loaded on first use. This accelerates startup (down to 40% of the time for 1.9.x) and reduces memory usage – the latter is probably unimportant of itself, but reduces commensurately the time spent in garbage collection.

  Packages are by default installed using lazy loading if they have more than 25Kb of R code and did not use a saved image. This can be overridden by `INSTALL --[no-]lazy` or via a field in the `DESCRIPTION` file. Note that as with `--save`, any other packages which are required must be already installed.

  As the lazy-loading databases will be consulted often, R will be slower if run from a slow network-mounted disc.

- All the datasets formerly in packages 'base' and 'stats' have been moved to a new package 'datasets'. `data()` does the appropriate substitution, with a warning. However, calls to `data()` are not normally needed as the data objects are visible in the 'datasets' package.

  Packages can be installed to make their data ob-