Package 'scriptests'

February 20, 2015

Type Package

Title Transcript-based unit tests that are easy to create and maintain

Version 1.0-9

Date 2012-04-11

Author Tony Plate

Maintainer Tony Plate <tplate@acm.org>

Description Support for using .Rt (transcript) tests in the tests directory of a package. Provides more convenience and features than the standard .R/.Rout.save tests. Tests can be run under R CMD check and also interactively. Provides source.pkg() for quickly loading code, DLLs, and data from a package for use in an edit/compile/test development cycle.

License GPL

Repository CRAN

Date/Publication 2012-04-13 05:22:03

NeedsCompilation no

R topics documented:

`scriptests-package` *Support for running transcript-style tests*

Description

Support for running transcript-style tests

Details

Transcript-style tests are text files containing R commands and output, as though copied verbatim from an interactive R session. The output in the transcript file must match the actual output from running the command for the test to pass (with some exceptions - see "Control over matching" below). This testing framework is identical in concept to the standard `.R/.Rout.save` tests that are run by the `R CMD check`, but with some enhancements that are intended to make test development and maintanance faster, more convenient, and easier to automate:

- Only the output file is needed - the inputs are parsed from the output file (i.e., `.Rt` file, which is analogous to an `.Rout.save` file)
- Test output matching is more lenient on white space differences, and more flexible in that some test output can be transformed by regular expressions prior to matching, or ignored entirely
- directives can specify whether a test-case output mismatch should be noted as an informational message, a warning, or an error (one or more errors results in `R CMD check` stopping with an indication of error after running all tests). Unlike the standard tests in `R CMD check`, output mismatch detected by `scriptests` results in `R CMD check` stopping with an error.
- A concise summary of warnings and errors is given at the end
- Testing can continue after errors and can report multiple errors at the end, rather than stopping at the first error.

To make it so that "R CMD check" will run transcript-style tests, do the following:

1. Make sure the `scriptests` package is installed on your system
2. Put a file named "runtests.R" in the "tests" directory in your own package with the following contents:

   ```
   library(scriptests)
   runScripTests()
   ```

3. add transcript files ending in .Rt in the "tests" directory in your own package, e.g.:

   ```
   > 1 + 2
   [1] 3
   >
   ```

4. add the line `Suggests: scriptests` to DESCRIPTION file. If there is an existing "Suggests:" line, just add `scriptests` to it. (It's better to use the `Suggests:` than the `Depends:` fields, because packages listed in the depends field are loaded when the package is loaded for normal use, and the `scriptests` package is usually not needed for normal use of a package – the `scriptests` package will only be needed for testing.

At the end of testing, the file `test-summary.txt` will be left in the `tests` directory. To be entirely sure that the tests were run, also check for the existence of `test-summary.txt`.

If any tests fail, the file `test-summary.fail` (a copy of `test-summary.txt`) will also be left in the `tests` directory – the existence of this file can be used in a programmatic check for whether all tests passed.

Tests can be run interactively using the function `runtests()`. The function `source.pkg()` can be useful to quickly re-read the function definitions in a package during code development. See the section "Running tests" in the documentation for `runtests()` for further details.

Notes:

- All commands in the transcript file must be prefixed with command or continuation prompts, exactly as they appear in a transcript.
- scriptests uses simple heuristics to identify commands, comments and output. If the transcript cannot be separated into comments, commands and output by these heuristics (e.g., if a command prints out a line starting with the command prompt "> "), things will not work properly.
- When running tests in a package, scriptests uses a heuristic to guess the package name and automatically include an appropriate `library(package-being-tested)` command before the tests. If this heuristic fails, the functions from the package being tested may not be accessible. If this problem occurs, it can be worked around by explicitly including a `library(package-being-tested)` command at the beginning of each `.Rt` file.
- To have tests continue to run after encountering an error, put the command `options(error=function() NULL)` at the beginning of the transcript file. This will cause the non-interactive R session that runs the commands in the scripts to continue after an error, instead of stopping, which is the default behavior for non-interactive R.

Control over matching

Actual output is matched to desired output extracted from the transcript file in a line-by-line fashion. If text is wrapped differently over multiple lines, the tests will fail (unless `ignore-linebreaks` is used). Different output width can easily happen if `options("width")` was different in the session that generated the desired output. Before trying to match, scriptests converts all white-space to single white-space, unless a control line specifies otherwise.

The following control lines can be present in the transcript after a command and before its output:

#@ignore-output Ignore the output of this particular command – a test with this control line will always pass (unless it causes an R error, and `options(error=function() NULL)` was not set.)

#@gsub(pattern, replacement, WHAT) where `WHAT` is `target`, `actual` or `both` (without quotes). Make a global substitution of `replacement` text for `pattern` text (a regular expression) in the desired (target) output or the actual output. E.g.,

```
> cat("The date is <", date(), ">\n", sep="")
#@gsub("<[^>]*>", "<a date>", both)
The date is <Sat Jul 10 16:20:01 2010>
>
```

#@warn-only: OPTIONAL-TEXT A mismatch is treated as an "warning", not an error

#@info-only: OPTIONAL-TEXT A mismatch is treated as an "info" event, not an error

#@diff-msg: OPTIONAL-TEXT Output OPTIONAL-TEXT if the desired and actual output do not match

#@keep-whitespace Leave the whitespace as-is in the desired and actual output

#@ignore-linebreaks Target and actual will match even if wrapped differently over multiple lines

The `tests` directory can also contain a `CONFIG` file, which can specify the functions to call for testing. The defaults are equivalent to the following lines in the `CONFIG` file:

```
Depends: scriptests
Debug: FALSE
Rsuffix: R
StopOnError: FALSE
Initialize: scriptests:::initializeTests()
Diff: scriptests:::ScripDiff()
Finalize: scriptests:::summarizeTests()
```

The `CONFIG` file is optional and is not needed in ordinary usage.

Note

The standard Emacs ESS functions for writing out `".Rt"` files will strip trailing white space, which can result in many unimportant mismatches when using `ediff` to compare `".Rt"` and `".Rout"` files (e.g., because an R transcript will have `"> "` for empty command lines). Also, `".Rt"` files are read-only by default, and the return key is bound to a command to send the current line to an R interpreter. It is more convenient if all these special behaviors are turned off. Put the following in your `.emacs` file to tell ESS not mess with `".Rt"` files prior to saving them:

```
(add-hook 'ess-transcript-mode-hook
          ;; According to the ess docs, ess-nuke-trailing-whitespace-p
          ;; is supposed to be nil by default (see the defvar in ess-utils.el).
          ;; But it gets set to t somewhere else, so disable it here for
          ;; .Rt files, and also make RET behave the regular way.
          (lambda ()
          (if (string-match ".[Rr]t$" (buffer-name))
             (progn
                (make-variable-buffer-local 'ess-nuke-trailing-whitespace-p)
                (define-key ess-transcript-mode-map (kbd "RET") 'newline)
                (toggle-read-only 0)
                (setq ess-nuke-trailing-whitespace-p nil)))) t)
```

Author(s)

Tony Plate <tplate@acm.org>

compareTranscriptAndOutput *Compare desired and actual output from running transcript-style tests.*

Description

Compare desired and actual output from running transcript-style tests. This function is called by the scripts supplied with the EasyTests package.

Usage

```
compareTranscriptAndOutput(name, tests, results, verbose = TRUE)
```

Arguments

name	Name of the file of tests.
tests	A list of test blocks, as generated by parseTranscriptFile
results	A list of test blocks, as generated by parseTranscriptFile
verbose	A logical flag indicating whether to write progress reports to stdout.

Details

Compares tests line-by-line, with some control over how matching is done and what mismatches counts as errors, warnings, or informational events.

Value

Returns an object of class RtTestSetResults, for which there are summary and print methods.

Author(s)

Tony Plate

parseTranscriptFile *Parse a R transcript file into blocks of commands and output*

Description

Parse a R transcript file into blocks of commands and output.

Usage

```
parseTranscriptFile(file, ignoreUpToRegExpr = NULL, ignoreAfterRegExpr=NULL, subst=NULL)
```

Arguments

`file`	The name of the file containing transcripts.
`ignoreUpToRegExpr`	If non-NULL, discard lines in the file up to and including the line that matches this regular expression.
`ignoreAfterRegExpr`	If non-NULL, discard lines in the file including and beyond the line that matches this regular expression.
`subst`	Provides control over whether the string "pacakge:::" is removed from test code. The default value should work.

Details

Tries to split the lines in `file` up into blocks with the following structure:

- commments: lines preceeding the test that begin with `> #` or that are empty
- command: line beginning with > and subsequent lines beginning with the continuation char +
- control: lines following the command that begin with `#@`
- output: other lines following the command and before the next comment or command, which are presumed to be output from the command

Value

A list of commands and their associated (apparent) output. Each element of the list is a list with the following possible components:

`comment`	A character vector containing the lines that were interpreted as comments
`input`	A character vector containing the lines that were interpreted as commands
`expr`	The parsed input (an expression). Will be a character vector containing an error message if the input could not be parsed as an R expression.
`control`	A character vector containing the lines that were interpreted as control lines
`output`	A character vector containing the lines that were interpreted as output lines

Author(s)

Tony Plate

plus *addition*

Description

Add two numbers. This function is created for testing purposes.

Usage

```
plus(x, y)
```

Arguments

x	a number
y	a number

Value

The result of adding x and y

runScripTests *Run the tests in the tests directory of a package*

Description

This function is not intended for interactive use, see runtests() for that. This function is intended to be called from the file runtests.R in the tests subdirectory of a package. The runtests.R file should contain these two lines:

```
library(scriptests)
runScripTests()
```

Usage

```
runScripTests(..., initializeFun = Quote(initializeTests()),
              finalizeFun = Quote(summarizeTests()),
              diffFun = Quote(ScripDiff()),
              subst = NULL, pattern = NULL, quit = TRUE)
```

Arguments

...	
initializeFun	Function for initializing tests.
finalizeFun	Function for finalizing tests – should compute a summary of test results.
diffFun	Function for checking differences between target output and actual test output.
subst	
pattern	Only run tests whose filename matches this regular expression pattern.
quit	Call q("no") at the end? Ignored in interactive sessions.

Details

When runScripTests() runs, it will create a .R file with the commands extracted from each .Rt file. It will then run each .R file in a separate R session, save the output in a .Rout file, and compare the output with the .Rt file. runScripTests will leave a summary in the file test-summary.txt. If there are errors, the summary will be duplicated in the file test-summary.fail (the presence of a file ending in .fail signals an error to "R CMD check".)

Value

invisible(NULL)

Author(s)

Tony Plate <tplate@acm.org>

---

runtests *Interactively run some Rt test files in a package*

Description

Run some Rt test files in a package from within an interactive R session. There are two major modes in which runtests() can be used:

- full=FALSE: Running tests within the current R session, which can change existing R variables, and create new ones.
- full=TRUE: Running tests by creating a new R session for each test file. This is safer, but slower, and also requires an installed version of the package in a local directory, such as one created by running R CMD check at the command line.

Usage

```
runtests(pkg.dir = getOption("scriptests.pkg.dir", "pkg"),
         pattern = ".*",
         file = NULL,
         full = FALSE,
         dir = TRUE,
         clobber = FALSE,
         output.suffix = NULL,
         console = FALSE,
         ...,
         verbose = TRUE,
         envir = globalenv(),
         subst = NULL,
         path = getOption("scriptests.pkg.path", default=getwd()))

dumprout(res = .Last.value,
         output.suffix = ".Rout.tmp",
         verbose = TRUE,
         console = FALSE,
         files = !console,
         clobber = identical(output.suffix, ".Rout.tmp"),
         level=c("error", "all", "info", "warning"))
```

Arguments

pkg.dir	The directory in which the package code and tests reside.
path	The path to the package in which the tests reside. If path contains $PKG, the value of pkg.dir will be substituted for $PKG, otherwise the value of pkg.dir will be appended to pkg.dir to give the location of the pacakge.
pattern	A regular expression pattern of test files to be run. Only one of pattern and file should be supplied.
file	The name of the file(s) containing tests to be run.
full	If TRUE run in full testing environment: create directory for tests; copy tests to that directory; chdir to directory; run each test file in a newly created R session.
dir	Directory where tests will be run. The default is usually satisfactory. This can be important if the files access and/or create any files or directories. The default value depends on full: if full=FALSE the default value is paste(pkg.dir, ".tests", sep=""; if full=TRUE the default value is paste(pkg.dir, ".Rcheck/tests", sep="". Supply dir=FALSE to run tests in the current directory.
console	If TRUE, output from the test will be written as output from this function. The default is FALSE for runtests() and TRUE for dumprout().
files	Opposite of console (specify just console=TRUE or file=TRUE)
clobber	Should existing output files or directories be clobbered? For safety, the default is FALSE. For runtests(), if the dir already exists, clobber=TRUE will result in replacing that directory, clobber=FALSE will result in stopping with an error if the directory already exists.

output.suffix	File suffix for actual output transcript files (default NULL for runtests, meaning no files written), if output.suffix==TRUE, uses suffix .Rout.tmp
level	Skip files unless they have notifications at the specified level or above.
...	Arguments to pass on to runScripTests(). Only relevant when full=TRUE. (Ignored with a warning message otherwise.)
verbose	Should progress indications be printed?
envir	The environment in which to run the tests.
subst	Provides control over whether the string "package:::" is removed from test code. Supply subst=FALSE to prevent any substitution. The default value will remove "package:::" when appropriate.
res	A value returned from runtests().

Details

runtests() runs some or all of the tests (in .Rt files) found in the tests directory of a package. The arguments pkg.dir and path are used to specify the location of the package (see the section **Running tests** below for further details). runtests() is designed to be used in conjunction with source.pkg(), which reads all the R code in a package into a special environment on the search path. With the default argument full=FALSE, runtests() will run the tests in the same R session, though it will create a special directory which will be used as the working directory while running the tests. In this mode of operation, output from tests is not is not written to files (unless output.suffix is supplied with a value), though tests may create or modify files themselves.

Supplying runtests(..., full=TRUE) will run each test file in a new R session - the way tests are run under R CMD check. When run like this, the package must be installed in the location <pkg.dir>.Rcheck or or <pkg.name>.Rcheck, relative to the working directory of the current R session. This can be accomplished by running the shell command

```
R CMD check --no-codoc --no-examples --no-tests --no-vignettes --no-latex <pkg.dir>
```

in the same directory as the R session is running in.

For working with a package in a different location relative to the working directory of the R session, supply the path to the package directory as path (can be either a relative or absolute path.)

After either of the package or path has been specified once, it is remembered and will be used as the default value next time either of runtests() or source.pkg() is called.

dumprout() writes actual R output to the console or to files. It creates one file for each test run. Specifying level= skips files that have no notification at the given level or above. With a missing first argument, dumprout() uses the value of the previously run command, which allows it to be meaningfully used directly after a runtests() command.

Value

runtests() returns an invisible list of RtTestSetResults objects (each element of the list is the result of running and checking the test in one file.) This result can be given to dumprout() to write actual R output to temporary files for test debugging and development purposes. dumprout() returns its first argument (whose default value is .Last.value), allowing it to be run several times in a row without having to supply any arguments

Running tests

runtests() is intended to be used in an R session while developing and/or maintaining code and/or tests. It is designed to work together with source.pkg(), which rapidly reads the R code in a package.. The arguments given to runtests() will depend on how package directories are organized, and the working directory for the R session used for code and test development.

Scenario 1: working directory of R session contains the package source code

The package source code mynewpkg is in

```
  /home/tap/R/packages/mynewpkg/DESCRIPTION
  /home/tap/R/packages/mynewpkg/R
  /home/tap/R/packages/mynewpkg/tests
... etc ...
```

and the working directory for the R session is

```
  /home/tap/R/packages
```

To read package R code and run some tests, do:

```
  > source.pkg(pkg.dir="mynewpkg")
  > res <- runtests(pkg.dir="mynewpkg", pattern="testfile1", clobber=T)
```

This will create a directory /home/tap/R/packages/mynewpkg.tests, copy all tests to it, and run the tests whose name matches testfile1 (to run all the tests, omit the argument pattern=).

Test files (ending in .Rt) will be looked for in the directory mynewpkg/tests/, and various output files could be created in /home/tap/R/packages (such as those created by the tests, and also some created by runtests). If there are errors or warnings in matching actual and desired output, a brief description and summary will be printed. A transcript of the actual output from running the tests can be printed by

```
  > dumprout(res)
```

The default is to print transcripts only of tests that had warnings or errors. To write the transcripts to files, do:

```
  > dumprout(res, file=T)
```

This will create files like testfile1.Rout.tmp in the working directory of the R session. These functions remember the most recent pkg.dir argument, and dumprout() will use a .Last.value left by runtests(), so it is possible to omit various arguments in the above commands, e.g.:

```
  > source.pkg(pkg.dir="mynewpkg")
  > runtests(pattern="testfile1", clobber=T)
  > dumprout(file=T)
```

Note that the name of the directory where the package lives (here mynewpkg) may be the same as or different from the name of the package.

Scenario 2: working directory of R session is in a different filesystem branch to package source code

The package code mynewpkg is in

```
  /home/tap/R/packages/mynewpkg/DESCRIPTION
  /home/tap/R/packages/mynewpkg/R
  /home/tap/R/packages/mynewpkg/tests
... etc ...
```

and the working directory for the R session is

```
/home/tap/R/sandbox
```

To read package R code and run some tests, do:

```
> source.pkg(pkg.dir="mynewpkg", path="/home/tap/R/packages")
> res <- runtests(pkg.dir="mynewpkg", pattern="testfile1", clobber=T)
```

and proceed as above. The directory `/home/tap/R/sandbox/mynewpkg.tests` will be created and used as the working directory for running tests.

Scenario 3: deeper directory structure for package source code

In an R-forge-like directory structure, the package files and directories may be in

```
/home/tap/R/rforge/projectname/pkg/DESCRIPTION
/home/tap/R/rforge/projectname/pkg/R/
/home/tap/R/rforge/projectname/pkg/tests/
... etc ...
```

and the working directory could be

```
/home/tap/R/rforge/projectname
```

To conveniently source R code and run tests in this directory organization, do this

```
> source.pkg(pkg.dir="projectname", path="/home/tap/R/rforge/$DIR/pkg")
> res <- runtests(pkg.dir="projectname", pattern="testfile1", clobber=T)
```

Tests will be run in the directory

```
/home/tap/R/rforge/projectname/projectname.tests
```

Author(s)

Tony Plate <tplate@acm.org>

See Also

`source.pkg()` shares the options `scriptests.pkg.dir` and `scriptests.pkg.path` that provide defaults for the `pkg.dir` and `path` arguments.

For running tests in under `R CMD check`, use the function `runScripTests()` in the file `tests/runtests.R` in the package.

For an overview and for more details of how tests are run, inlcuding how to make test output matching more flexible, see scriptests.

Examples

```
## Not run:
> # To run like this example, set the current working directory
> # to where the package code lives.
> # source.pkg() reads in the functions -- could just as well
> # load the library, but source.pkg() can be more convenient
> # when developing a package.
> source.pkg("scriptests")
Reading 5 .R files into env at pos 2: 'pkgcode:scriptests'
Sourcing scriptests/R/createRfromRt.R
Sourcing scriptests/R/interactive.R
Sourcing scriptests/R/oldcode.R
Sourcing scriptests/R/plus.R
Sourcing scriptests/R/rttests.R
list()
> runtests("simple1")
Running tests in scriptests/tests/simple1.Rt (read 4 chunks)
....
Ran 4 tests with 0 errors and 0 warnings from scriptests/tests/simple1.Rt
> runtests("simple2")
Running tests in scriptests/tests/simple2.Rt (read 5 chunks)
.....
Ran 5 tests with 0 errors and 0 warnings from scriptests/tests/simple2.Rt
> runtests("simple")
Running tests in scriptests/tests/simple1.Rt (read 4 chunks)
....
Ran 4 tests with 0 errors and 0 warnings from scriptests/tests/simple1.Rt
Running tests in scriptests/tests/simple2.Rt (read 5 chunks)
.....
Ran 5 tests with 0 errors and 0 warnings from scriptests/tests/simple2.Rt
>

## End(Not run)
```

ScripDiff *Support functions for package scriptests*

Description

These are support functions for package scriptests. They are called to prepare a test directory for running tests, compare the output of each test with target out, and summarize test results.

Usage

```
ScripDiff(commandfile, outfile = NULL, savefile = NULL, debug = FALSE, R.suf = "R")
initializeTests(debug = FALSE, create.Rout.save = FALSE, addSelfCheck = FALSE, pattern = NULL, subst =
summarizeTests(debug = FALSE)
```

Arguments

commandfile	The name of the file containing R commands.
outfile	The name of the file containing R output.
savefile	The name of the file containing pre-existing R output to compare against.
debug	Debugging flag.
create.Rout.save	Should a .Rout.save file be created. Default is FALSE.
addSelfCheck	Controls an unused an obselete feature of adding extra code to the end of the R commands that will check the output created.
pattern	Only process test files whose name matches pattern
subst	Substitutions to make when parsing the transcript file.
R.suf	The suffix (not including a dot) to use for auto-generated files containing R commands.

Details

initializeTests Pre-process tests, writing commands to a file, and creating a file with a saved R object containing information to analyze the results of the test.

ScripDiff Compare test output against desired output and summarize the differences.

summarizeTests Summarize the results of all tests.

Value

Returns the value 0 on successful completion.

Author(s)

Tony Plate <tplate@acm.org>

---

scriptests.design *Design considerations for package scriptests*

Description

Design considerations for package scriptests. This file is a poorly organized collection of notes regarding the design and evolution of the scriptests package.

Details

"R CMD check" is (as of R-2.9.0) an R script `src/scripts/check.in`, which invokes R with the command `tools:::.runPackageTestsR($extra)` to run tests. If this R session returns non-zero, the check process stops with an error, outputting 13 lines from the first .fail file found (if one exists).

`tools:::.runPackageTestsR($extra)` invokes `R CMD BATCH` to run each .R file. This is where the non-overrideable redirection of stderr to .Rout happens.

New design for scriptests to work with R-2.9.0 in which "R CMD check" no longer uses Makefiles in tests directory.

- All transcript files are stored in the `tests` directory with the file extension `.Rt`.
- There should be a file called `runtests.Rin`, which should contain the following two lines:

```
library(scriptests)
runScripTests()
```

- When `runScripTests()` runs, it will create a `.R` file with the commands extracted from each `.Rt` file. It will then run each `.R` file in a separate `R` session, save the output in a `.Rout` file, and compare the output with the `.Rt` file. `runScripTests` will leave a summary in the file `test-summary.txt`. If there are errors, the summary will be duplicated in the file `test-summary.fail` (the presence of a file ending in `.fail` signals an error to `"R CMD check"`.)

Comparisons to previous scriptests designs

The functionality that this design gives up over previous designs is immediate checking of test output. Instead test output is checked after all tests are run. This is because of the difficulty of controlling the execution order of different test files – it's hard to create pairs of test/check files that will run in the correct order (order of execution depends on the order of files in the value of `dir`).

The functionality that this design retains from previous designs (and from the native testing framework in R) is that each file of R commands to be checked in run in a new R session. This means that commands in one test file cannot mess up another test file. It also means that the testing framework doesn't need to mess around with trying to capture output.

There are several possible methods that could be used to check output immediately after running each test:

1. prepend each `.R` file with a function that checks output of all previously run tests (skipping ones already done).
2. append to each `.R` file a call to a function that flushes stdout (`flush(stdout())`) and checks tests output.
3. define a `.Last()` function (will be called by `q()`) that checks test output.

Hangup with the above approach: it's not possible to have output from the runs go to stdout – because all the output from running a .R file is redirected to the corresponding .Rout file.

Alternate approach: a setup file calls my own version of .runPackageTests(), which then executes an R session for each .R file. Still have the problem of not getting any output to the console. Can solve this by by-passing R CMD BATCH, and just invoking R more directly.

Original scriptests design based on Makefiles

An initial design changed the default goal by assigning .DEFAULT_GOAL in tests/Makefile. This works with GNU make version 3.81, which is standard in Ubuntu Linux. However, the Rtools set of programs for Windows includes GNU make version 3.79, which does not appear to recognize the .DEFAULT_GOAL special variable. Additionally, version 2.6.2 (2008-02-08) of "R Installation and Administration" specifically says that GNU make version 3.81 does not work to compile under Windows. Furthermore, Mac OS X version 10.4 (Tiger) includes GNU make version 3.80, which also does not appear to recognize the .DEFAULT_GOAL special variable.

Consequently, a different approach to getting R CMD check to run additional tests is needed.

The approach implemented as of version 0.1-6 (March 2009) is to use a "%" target in the makefile (which is always called), with an action in the body of the rule that invokes a 'make' recursively with the desired target (here `all-Rt`).

http://www.gnu.org/software/automake/manual/make/Force-Targets.html describes 'force targets'.

Here's the relevant section from scriptests/inst/scripts/Makefile.sub:

```
# Use 'force' to effectively create another target, by calling
# make recursively with the target 'all-Rt'.
# Based on code at
# http://www.gnu.org/software/automake/manual/make/Overriding-Makefiles.html
# but with more levels of protection to avoid calling make with
# the target 'all-Rt' more than once, because this makefile is
# read many times.  Condition on DONEFORCE being not defined
# to avoid infinite recursion.
ifeq ($(strip $(DONEFORCE)),)

        @(if [ ! -f forceonce ] ; then \
      $(MAKE) -f $(R_SHARE_DIR)/make/$(RSHAREMAKEFILE) $(makevars) -f $(MAINTESTMAKE) DONEFORCE=TRUE
        fi )
        @touch forceonce

force: ;
endif
```

The various 'make' variables used here are defined in ...

Not used: A Makefile framework that modifies Makevars to create a goal, and also uses a dummy .R file

This makefile does two things to try to create a goal: (1) it edits Makevars to insert an extra goal; and (2) it uses a specific rule for a allrt.Rin to allrt.R to run a sub-make (the rule is written to Makevars).

Both of these approaches should work, but the whole thing does not seem to work reliably. In any case, the sub-make can be called directly from the "force" block instead of using the force block to create a rule that calls the sub-make. This code is preserved here because it uses a number of techniques that might come in handy somewhere else.

This framework uses three files in the tests directory: `Makefile`, `Makefile.win` and `allrt.Rin`.

Makefile:

```
# Redefine the default goal (a GNU make feature) so that we
# have the Rt tests as subgoals.
# Note that the default goal can contain only one target.
# .DEFAULT_GOAL := all+Rt

ifeq ($(strip $(RSHAREMAKEFILE)),)
  RSHAREMAKEFILE=tests.mk
endif

# Need MAINTESTMAKE so that when we edit Makevars we can set $makevars
# to include the same makefile as R CMD check uses (which, under Windows,
# is Makefile.win if it exists)
ifeq ($(strip $(MAINTESTMAKE)),)
  MAINTESTMAKE=Makefile
endif

# Define ScripTestsErrorAction to be 'continue' or 'stop'.
# This controls what happens running under R CMD check
# when there are errors in one of the Rt tests.
ScripTestsErrorAction=stop

# Under Unix-likes we want to run R like this:
#    @R_LIBS=$(R_LIBS) $(R) ... arguments ...
# while under Windows we want to run R like this:
#    $(R) @R_LIBS=$(R_LIBS) ... arguments ...
# So, always run R like this: $(R_PRE) @R_LIBS=$(R_LIBS) $(R_POST)
# with appropriate definitions for R_PRE and R_POST.
# If we are running under Windows, we will have already executed
# Makefile.win, which will have defined R_PRE=$(R) .
ifeq ($(strip $(R_PRE)),)
  # Unix-like
  R_PRE=
  R_POST=$(R)
endif

# If we do want to change the RDIFF command, should include
# 'changeRdiff' as the second dependency for all+Rt
all+Rt: createScripts all rt-tests ScripTests.summary
all-Rt: createScripts rt-tests ScripTests.summary
        @echo doing all-Rt

allrt := $(wildcard *.Rt)
# allrtwant := $(allrt:.Rt=.Rout.want)
# allrtout := $(allrt:.Rt=.Rout)
allrtres := $(allrt:.Rt=.Rtres)
# allrtin := $(allrt:.Rt=.R)
```

```
	@(if [ ! -f forceonce ] ; then \
	echo Modifying Makevars for rttests ; \
	sed -i 's/test-src-1 =/test-src-1 = all-Rt/' Makevars ; \
	sed -i "s/makevars =/makevars = -f $(MAINTESTMAKE)/" Makevars ; \
	fi )
	@(if [ ! -f forceonce ] ; then echo 'allrt.R:
	echo '  @echo Using special rule $$@ to run all tests ... ' ; \
       echo '  $$(MAKE) -f $$(R_SHARE_DIR)/make/tests.mk $$(makevars) all-Rt' ; \
	echo '  @echo 1+1 > $$@' ; fi ) >> Makevars
	@touch forceonce

force: ;

# Don't need this unless we want standard .R/.Rout.save tests
# to use the modified RDIFF (not the ones run here - they use
# the commands in the .Rout.Rtres rule)

changeRdiff:
	@echo Changing Rdiff in Makevars
	echo RDIFF = echo hehehe >> Makevars

# Create scripts we will need (do this here to minimize the
# number of files needed to support scriptests in a package.)
createScripts:
      @( echo '## Run a script in the scriptests/scripts (from scriptests/inst/scripts)' ; \
      echo '## This file does not have .R suffix because if it did, R CMD check would want to run it as
	echo 'library(package="scriptests", char=TRUE)' ; \
	echo 'args <- commandArgs(TRUE)' ; \
	echo 'debug <- is.element("--debug", args)' ; \
	echo 'if (length(args)>0) {' ; \
       echo '   script.path <- system.file(package="scriptests", "scripts")' ; \
	echo '    if (debug) {' ; \
      echo '      cat("RtTestScript called with ", length(args), " args: ", paste("\"", args, "\"", c
      echo '      cat("Looking for scripts in \"", script.path, "\"\n", sep="")' ; \
	echo '    }' ; \
	echo '    source(file.path(script.path, args[1]))' ; \
	echo '}' ) > RunScripTestsScript

# Don't need the following because we can run and check the tests
# with out needing a sub-make.  If we do use a sub-make, then we
# need to put rules used in 'Makevars'.
#       echo .SUFFIXES: .R .Rin .Rout .Rt >> Makevars
#       echo .Rt.R: >> Makevars
#       echo '  echo Creating $$@ from $$<' >> Makevars
#       echo '  sed -n "s/^[>+] //p" $$< > $$@' >> Makevars
#       echo '  cat $$< > $$@out.save' >> Makevars

#  This suffix list and rule for .Rt.R needs to go in Makevars
```

```
.SUFFIXES: .R .Rin .Rout .Rt .Rtres

# Files:
#   .Rt: a transcript, with both commands and desired output, & possibly directives
#    .R: R commands generated from .Rt
#   .Rout.want: desired transcript output generated from .Rt (maybe don't need this)
#    .Rt.save: the processed Rt script as an R object
#    .Rout: the output from R when given the commands in .R
#    .Rtres: summary of results from comparing .Rout and .Rout.want

.Rt.R:
        @echo '**' Creating $@ and $@out.want from $<
      $(R_PRE) @R_LIBS=$(R_LIBS) $(R_POST) $(R_OPTS) --vanilla --slave --args prepin.R $< $@ $@out.wa

# Preserve intermediate .R and .Rout files - the user might want to inspect them
# (the .R file to see exactly what the R commands were, and the .Rout file to
#  see what output was actually produced)
.PRECIOUS:

# Diff the output
# Need to have R_LIBS & R round the other way for Windows
.Rout.Rtres:
        @echo '**' Creating $@ from $< and $<.want
      $(R_PRE) @R_LIBS=$(R_LIBS) $(R_POST) $(R_OPTS) --vanilla --slave --args diffout.R $< $(@:Rtres=

rt-tests: $(allrtres)

# Method for using a 'make' sub process to run tests (will use a chained rule
# for X.Rout: X.Rt -> X.R -> X.Rout if we put the rule .Rt.R in Makevars
# (needs to go there because this Makefile is not read by the sub-make).
rt-tests2:
        echo Making target rt-tests
        (out=`echo *.Rt | sed 's/\.Rt\( \|$\)/.Rout /g'`; \
          if test -n "$${out}"; then \
          $(MAKE) -f $(R_SHARE_DIR)/make/$(RSHAREMAKEFILE) $(makevars) $${out}; \
          fi)

ScripTests.summary: $(allrtres)
        @echo '**' Creating summary of tests in $@
      $(R_PRE) @R_LIBS=$(R_LIBS) $(R_POST) $(R_OPTS) --vanilla --slave --args summary.R $@ $(allrtres
      @(if [ "$(ScripTestsErrorAction)" = stop -a -f ScripTests.haserrors ] ; then \
         echo "Stopping because tests had errors and ScripTestsErrorAction=stop in Makefile" ; \
           exit 1 ; \
        fi)
```

Makefile.win:

```
RSHAREMAKEFILE=wintests.mk
```

```
# R_SHARE_DIR is defined when running under Linux, but not Windows...
R_SHARE_DIR=$(R_HOME)/share
MAINTESTMAKE=Makefile.win
# Under Windows we want to run R like this: $(R) R_LIBS=$(R_LIBS)
# Seems that don't need this anymore, at least as of R 2.6.1
# R_PRE=$(R)
# R_POST=
include Makefile
```

allrt.Rin:

```
cat("1\n", file="allrt.R")
```

A Makefile framework using .DEFAULT_GOAL

This framework requires two files in the tests directory: `Makefile` and `Makefile.win`. Note that some of the code in here is unnecessary (e.g., the R_PRE and R_POST stuff – this was used to put VAR=VALUE before or after the name of the R program on a command line depending on whether Windows or Unix was being used.

Makefile:

```
# Redefine the default goal (a GNU make feature) so that we
# have the Rt tests as subgoals.
# Note that the default goal can contain only one target.
.DEFAULT_GOAL := all+Rt

ifeq ($(strip $(RSHAREMAKEFILE)),)
  RSHAREMAKEFILE=tests.mk
endif

# Define ScripTestsErrorAction to be 'continue' or 'stop'.
# This controls what happens running under R CMD check
# when there are errors in one of the Rt tests.
ScripTestsErrorAction=stop

# Under Unix-likes we want to run R like this:
#    @R_LIBS=$(R_LIBS) $(R) ... arguments ...
# while under Windows we want to run R like this:
#    $(R) @R_LIBS=$(R_LIBS) ... arguments ...
# So, always run R like this: $(R_PRE) @R_LIBS=$(R_LIBS) $(R_POST)
# with appropriate definitions for R_PRE and R_POST.
# If we are running under Windows, we will have already executed
# Makefile.win, which will have defined R_PRE=$(R) .
ifeq ($(strip $(R_PRE)),)
  # Unix-like
  R_PRE=
  R_POST=$(R)
endif
```

```
# If we do want to change the RDIFF command, should include
# 'changeRdiff' as the second dependency for all+Rt
all+Rt: createScripts all rt-tests ScripTests.summary

allrt := $(wildcard *.Rt)
allrtwant := $(allrt:.Rt=.Rout.want)
allrtout := $(allrt:.Rt=.Rout)
allrtres := $(allrt:.Rt=.Rtres)
# allrtin := $(allrt:.Rt=.R)

# Don't need this unless we want standard .R/.Rout.save tests
# to use the modified RDIFF (not the ones run here - they use
# the commands in the .Rout.Rtres rule)

changeRdiff:
        @echo Changing Rdiff in Makevars
        echo RDIFF = echo hehehe >> Makevars

# Create scripts we will need (do this here to minimize the
# number of files needed to support scriptests in a package.)
createScripts:
      @( echo '## Run a script in the scriptests/scripts (from scriptests/inst/scripts)' ; \
      echo '## This file does not have .R suffix because if it did, R CMD check would want to run it as
        echo 'library(package="scriptests", char=TRUE)' ; \
        echo 'args <- commandArgs(TRUE)' ; \
        echo 'debug <- is.element("--debug", args)' ; \
        echo 'if (length(args)>0) {' ; \
       echo '   script.path <- system.file(package="scriptests", "scripts")' ; \
        echo '    if (debug) {' ; \
      echo '      cat("RtTestScript called with ", length(args), " args: ", paste("\"", args, "\"", c
      echo '      cat("Looking for scripts in \"", script.path, "\"\n", sep="")' ; \
        echo '    }' ; \
        echo '    source(file.path(script.path, args[1]))' ; \
        echo '}' ) > RunScripTestsScript


# Don't need the following because we can run and check the tests
# with out needing a sub-make.  If we do use a sub-make, then we
# need to put rules used in 'Makevars'.
#       echo .SUFFIXES: .R .Rin .Rout .Rt >> Makevars
#       echo .Rt.R: >> Makevars
#       echo '  echo Creating $$@ from $$<' >> Makevars
#       echo '  sed -n "s/^[>+] //p" $$< > $$@' >> Makevars
#       echo '  cat $$< > $$@out.save' >> Makevars

#  This suffix list and rule for .Rt.R needs to go in Makevars

.SUFFIXES: .R .Rin .Rout .Rt .Rtres
```

```
# Files:
#   .Rt: a transcript, with both commands and desired output, & possibly directives
#    .R: R commands generated from .Rt
#   .Rout.want: desired transcript output generated from .Rt (maybe don't need this)
#    .Rt.save: the processed Rt script as an R object
#    .Rout: the output from R when given the commands in .R
#    .Rtres: summary of results from comparing .Rout and .Rout.want

.Rt.R:
        @echo '**' Creating $@ and $@out.want from $<
      $(R_PRE) @R_LIBS=$(R_LIBS) $(R_POST) $(R_OPTS) --vanilla --slave --args prepin.R $< $@ $@out.wa

# Preserve intermediate .R and .Rout files - the user might want to inspect them
# (the .R file to see exactly what the R commands were, and the .Rout file to
#  see what output was actually produced)
.PRECIOUS:

# Diff the output
# Need to have R_LIBS & R round the other way for Windows
.Rout.Rtres:
        @echo '**' Creating $@ from $< and $<.want
      $(R_PRE) @R_LIBS=$(R_LIBS) $(R_POST) $(R_OPTS) --vanilla --slave --args diffout.R $< $(@:Rtres=

rt-tests: $(allrtres)

# Method for using a 'make' sub process to run tests (will use a chained rule
# for X.Rout: X.Rt -> X.R -> X.Rout if we put the rule .Rt.R in Makevars
# (needs to go there because this Makefile is not read by the sub-make).
rt-tests2:
        echo Making target rt-tests
        (out=`echo *.Rt | sed 's/\.Rt\( \|$\)/.Rout /g'`; \
          if test -n "$${out}"; then \
          $(MAKE) -f $(R_SHARE_DIR)/make/$(RSHAREMAKEFILE) $(makevars) $${out}; \
          fi)

ScripTests.summary: $(allrtres)
        @echo '**' Creating summary of tests in $@
      $(R_PRE) @R_LIBS=$(R_LIBS) $(R_POST) $(R_OPTS) --vanilla --slave --args summary.R $@ $(allrtres
      @(if [ "$(ScripTestsErrorAction)" = stop -a -f ScripTests.haserrors ] ; then \
          echo "Stopping because tests had errors and ScripTests ErrorAction=stop in Makefile" ; \
            exit 1 ; \
        fi)
```

Makefile.win:

```
RSHAREMAKEFILE=wintests.mk
# Under Windows we want to run R like this: $(R) @R_LIBS=$(R_LIBS)
R_PRE=$(R)
```

```
R_POST=
include Makefile
```

---

source.pkg *Quickly load code and data of a package*

Description

source() all of the source files and data files in a package into a specially created environment. It also loads DLLs if requested, and if those DLLs can be found. This function is intended for use in code development, in situations where code from a package can be tested without worrying about imports, namespaces, etc.

Usage

```
source.pkg(pkg.dir = getOption("scriptests.pkg.dir", "pkg"),
           pattern = ".*",
           suffix = "\\.R$",
           dlls = c("no", "check", "build", "src"),
           pos = NA, all = FALSE,
           reset.function.envirs=TRUE,
           path=getOption("scriptests.pkg.path", default=getwd()))
```

Arguments

pkg.dir	The directory where the package code lives. This is remembered and the same value used as the default in subsequent invocations. This can be different from the package name, which is read from <pkg.dir>/DESCRIPTION.
pattern	A regular expression specifying the R files to source
suffix	The suffix for the R code files. Files are sourced only if they match both pattern and suffix.
dlls	Indicates where to look for DLLs or shared-objects: • no: don't load any DLLs • check: look in <pkg.dir>.Rcheck/<pkg.name>/libs and <pkg.name>.Rcheck/<pkg.name>/libs • build: look in the directory build (can be created with R CMD INSTALL -l build <package-tarba
pos	Which environment on the search path to source R code into
all	If TRUE, all files matching the pattern and suffix are sourced, if FALSE, only files that have changed since last sourced are sourced again.
reset.function.envirs	If TRUE the environments on all functions sourced are set to the global environment. See NOTE below for explanation.
path	The file system path to the directory in which the package is located. The initial default for path is the current directory. By default, R source files are looked for in the directory found by appending pkg.dir to path, i.e. <path>/<pkg.dir>/R/*.R. If path contains the string $PKG, pkg.dir is substituted for $PKG instead of being appended to path. path is remembered and the same value used as the default in subsequent invocations.

Details

The package directory (pkg.dir) does not need to be the same as the package name. The package name (refered to here as pkg.name) is found by reading the DESCRIPTION file.

All the objects created by the .R files are created in a special environment named pkgcode:<pkg.name>. If this environment does not already exist, it will be created and attached to the search path in position pos. If it does exist, no attempt is made to clean it before sourcing the .R files. All functions and objects defined in the .R files will be visible – namespaces are not implemented.

The easiest way to use this function is when the working directory of the R session is the directory where each package lives, i.e., the R code for package mypackage will be in the directory ./mypackage/R (relative to the working directory of the R session.) However, if the package directory is located elsewhere, supply it as path=, and this will be remembered for future invocations of source.pkg().

This function does not attempt to replicate all the actions involved in creating and attaching a package. It does the following:

- creates a environment named pkgcode:<pkg.name> where <pkg.name> is the name of the package (if it doesn't already exist)
- looks for a Depends line in the DESCRIPTION file and loads specified packages
- looks for .R files in the R subdirectory of the package, and, as appropriate, the R/windows or R/unix subdirectories, and uses sys.source() to read these into the pkgcode:<pkg.name> environment. If there is a Collate field in the DESCRIPTION files, this is used to sort the files before sourcing them.
- looks for .Rdata and .rda files in the data subdirectory, and uses load() to read these into the pkgcode:<pkg.name> environment
- if dlls="check" (not the default), source.pkg() looks for DLLs (SO files under Unix) in the directory left by R CMD check <pkg.dir>, i.e., in <pkg.dir>.Rcheck/<pkg.name>/libs or <pkg.name>.Rcheck/<pkg.name>/libs, and uses dyn.load() to load these DLLs. If the DLL was already loaded (as indicated by getLoadedDLLs()), dyn.unload() is called first. Be aware that unloading and reloading a DLL is not a reliable operation under many OS's, and even when the call completes without apparent error, the R session can be corrupted.

Value

A list of the problems encountered when sourcing the files.

Side effects

An environment is created and attached to the search path (or if it already exists, it is modified.) A variable named scriptests.pkg.dir is set in the global environment to the directory of the package whose code was sourced.

Note

If reset.function.envirs=FALSE is supplied, the environments on the functions sourced will be the environment where they live on the search path. Functions with their environment set like this will not be able to see objects in the global environment or in any environment earlier on the search path. The default behavior of setting their environment to be the global environment

produces behavior that is more similar to when the functions are sourced directly into the global environment.

Author(s)

Tony Plate <tplate@acm.org>

References

- Similar ideas in an R-devel post by Barry Rowlingson: http://n4.nabble.com/Lightweight-package-idea-td924 html#a924000
- Hadley Wickham source_package() function http://gist.github.com/180883, reads DESCRIPTION, loads dependencies, respects collation order

See Also

runtests() shares the options() variables scriptests.pkg.dir and scriptests.pkg.path that provide defaults for the pkg.dir and path arguments.

scriptests-package gives and overview of the package.

Examples

```
## Not run:
# sourcing the code in a package stored in <mypackage>/{DESCRIPTION,R,man,tests}:
source.pkg("<mypackage>")
# sourcing the code in a package stored in path/to/dir/<mypackage>/{DESCRIPTION,R,man,tests}:
source.pkg("<mypackage>", path="path/to/dir")
# sourcing the code in a package stored in pkg/{DESCRIPTION,R,man,tests}:
# where "pkg" is unrelated to the name of the pacakage
source.pkg("pkg", path="path/to/dir")
# sourcing the code in a package stored in <mypackage>/pkg/{DESCRIPTION,R,man,tests}:
source.pkg("<mypackage>", path="$PKG/pkg")

## End(Not run)
```

Index