testCoverage Package Vignette

Abstract

testCoverage is an R package for determining how much source code is tests by a given testing suite. The package supports tests written in both RUnit and testthat and outputs a report in HTML. Code coverage is measured by the coverage of executable statements by the testing suite.

Revision History

Version	Prepared By	Date	Details
0.1	Suchen Jin	2014-08-26	Draft

1 Introduction

1.1 Code Coverage

In order to measure the effectiveness of a set of unit tests validating R code this package computes the code coverage of those tests.

1.2 History

Originally written by Tom Taverner in 2013 as a tool to ensure R packages are thourghly validated and to score existing test suites. Since then the output have been improved and integrated with build systems.

Although intended to be released to the community from the onset a number of techniques used to intercept calls to particular base functions make this package unlikely to ever find its way onto CRAN.

1.3 Overview

```
y <- x + 10
```

Symbols in the source code are replaced by unique identifiers with the format `` `_x_y` `` where `x` is a number that uniquely identifies each file and `y` is a number that uniquely identifies a symbol within a file.

```
`_1_1` <- `_1_2` + 10
```

At every assignment a `_trace()` call is inserted.

```
`_1_1` <- {_trace(); `_1_2` + 10}
```

Where it is meaningful to do so unique identifiers within a trace point are setup to allow the `_trace` function to perform the coverage accounting.

```
y <- {_trace(1, 2); x + 10}
```

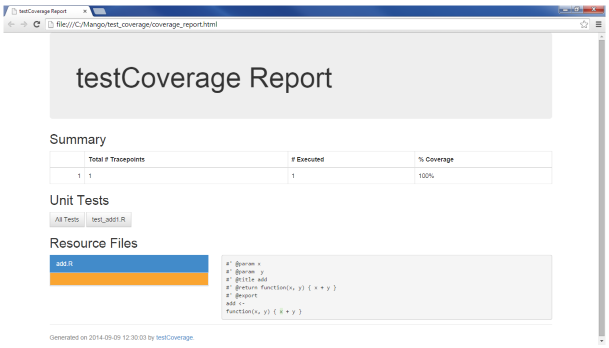
2 Usage

2.1 Individual Files

To use testCoverage with a disorganised set of R source files and test files is simply call reportCoverage with a character vector of your source files and another character vector with your unit test files. The parameter `isrunit` should be set to true if the tests have not been written in testthat as is assumed.

For example in the simplest where someone is testing a hypothetical file named `add.R` with a file of testthat tests named `test_add1.R` then the following will generate a basic report.

```
testCoverage("add.R", "test_add1.R")
```


testCoverage Report

file:///C:/Mango/test_coverage/coverage_report.html

testCoverage Report

Summary

	Total # Tracepoints	# Executed	% Coverage
1	1	1	100%

Unit Tests

All Tests | test_add1.R

Resource Files

add.R

```
#' @param x
#' @param  y
#' @title add
#' @return function(x, y) { x + y }
#' @export
add <-
function(x, y) { x + y }
```

Generated on 2014-09-09 12:30:03 by testCoverage.

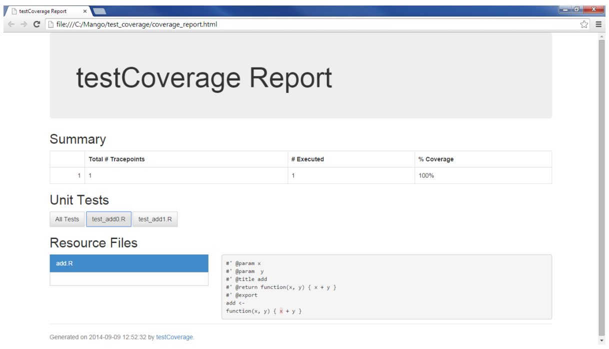
Figure 1: Report for add.R.

For runnable examples consult the package's compiled help files.

In this example the unit tests in `test_add0.R` has managed to test all of `add.R` by hitting its only trace point highlighted in green. (You may need to zoom in.)

Replacing the parameters with character vectors will produce a more complex report. In order to see with unit test file covered which files they are individually selectable along with which source file to display. The orange progress bar underneath a source file name indicates the test coverage achieved for that file.

```
testCoverage("add.R", c("test_add0.R",
  "test_add1.R"))
```



testCoverage Report ×
file:///C:/Mango/test_coverage/coverage_report.html

testCoverage Report

Summary

	Total # Tracepoints	# Executed	% Coverage
1	1	1	100%

Unit Tests

All Tests | test_add0.R | test_add1.R

Resource Files

add.R

```
#' @param x
#' @param  y
#' @title add
#' @return function(x, y) { x + y }
#' @export
add <-
function(x, y) { x + y }
```

Generated on 2014-09-09 12:52:32 by testCoverage.

Figure 2: Rreport for add.R with multiple tests.

In this example the selected unit test file named `test_add0.R` has failed to test any of `add.R`'s code. The trace point is highlighted in red whilst the file `test_add0.R` is selected to indicate this.

For documentation on the remaining parameters and runnable versions of the examples here access the functions help page from R using `?testCoverage` .

2.2 Packages

The `pkgCoverage` function from the last subsection can be used to test the code coverage of packages by manually specifying the code and tests directories, like before or simply calling, for example:

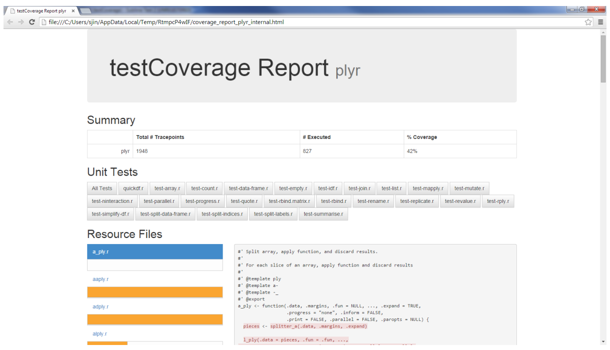
```
pkgCoverage("~/myPackage", "inst/tests")
```

And letting the report coverage assume the default folders. Although it is optional to specify the test folder it is recomended.

2.3 CRAN

As this is perceived to be a common use case the `cranCoverage` function can simplify the testing of packages by downloading packages not available locally from CRAN and allowing vectorisation over a vector of package paths or names. This will only work with packages which the maintainer has written compatible unit tests for.

If the package maintainer has used a non standard location for the unit tests it may be necessary to manually specify the folder path.



testCoverage Report plyr ×
file:///C:/Users/sjin/AppData/Local/Temp/RtmpcP4wIF/coverage_report_plyr_internal.html

testCoverage Report plyr

Summary

	Total # Tracepoints	# Executed	% Coverage
plyr	1948	827	42%

Unit Tests

All Tests | quickdf.r | test-array.r | test-count.r | test-data-frame.r | test-empty.r | test-idf.r | test-join.r | test-list.r | test-mapply.r | test-mutate.r | test-ninteraction.r | test-parallel.r | test-progress.r | test-quote.r | test-rbind.matrix.r | test-rbind.r | test-rename.r | test-replicate.r | test-revalue.r | test-rply.r | test-simplify-df.r | test-split-data-frame.r | test-split-indices.r | test-split-labels.r | test-summarise.r

Resource Files

a_ply.r
aaply.r
adply.r
alply.r

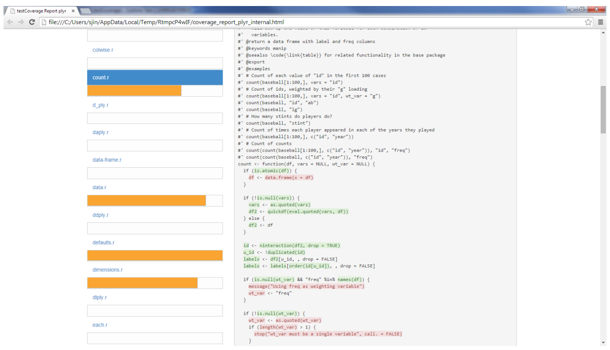
```
#' Split array, apply function, and discard results.
#'
#' For each slice of an array, apply function and discard results
#'
#' @template ply
#' @template a-
#' @template -_
#' @export
a_ply <- function(.data, .margins, .fun = NULL, ..., .expand = TRUE,
                  .progress = "none", .inform = FALSE,
                  .print = FALSE, .parallel = FALSE, .paropts = NULL) {
  pieces <- splitter_a(.data, .margins, .expand)

  l_ply(.data = pieces, .fun = .fun, ...,
```

Figure 3: Report for plyr.

In this package there is a large number of source files and test files that necessitate the ability to view results specific to a single unit test file.

The next figure is an example of a file with only partial test coverage. Untested execution paths are clearly highlighted in red.



testCoverage Report plyr ×
file:///C:/Users/sjin/AppData/Local/Temp/RtmpcP4wIF/coverage_report_plyr_internal.html

colwise.r
count.r
d_ply.r
daply.r
data-frame.r
data.r
ddply.r
defaults.r
dimensions.r
dlply.r
each.r

```
#'   variables.
#' @return a data frame with label and freq columns
#' @keywords manip
#' @seealso \code{\link{table}} for related functionality in the base package
#' @export
#' @examples
#' # Count of each value of "id" in the first 100 cases
#' count(baseball[1:100,], vars = "id")
#' # Count of ids, weighted by their "g" loading
#' count(baseball[1:100,], vars = "id", wt_var = "g")
#' count(baseball, "id", "ab")
#' count(baseball, "lg")
#' # How many stints do players do?
#' count(baseball, "stint")
#' # Count of times each player appeared in each of the years they played
#' count(baseball[1:100,], c("id", "year"))
#' # Count of counts
#' count(count(baseball[1:100,], c("id", "year")), "id", "freq")
#' count(count(baseball, c("id", "year")), "freq")
count <- function(df, vars = NULL, wt_var = NULL) {
  if (is.atomic(df)) {
    df <- data.frame(x = df)
  }

  if (!is.null(vars)) {
    vars <- as.quoted(vars)
    df2 <- quickdf(eval.quoted(vars, df))
  } else {
    df2 <- df
  }

  id <- ninteraction(df2, drop = TRUE)
  u_id <- !duplicated(id)
  labels <- df2[u_id, , drop = FALSE]
  labels <- labels[order(id[u_id]), , drop = FALSE]

  if (is.null(wt_var) && "freq" %in% names(df)) {
    message("Using freq as weighting variable")
    wt_var <- "freq"
  }

  if (!is.null(wt_var)) {
    wt_var <- as.quoted(wt_var)
    if (length(wt_var) > 1) {
      stop("wt_var must be a single variable", call. = FALSE)
    }
```

Figure 4: Report showing partial coverage in count.R.

2.4 Self Test

testCoverage includes a number of demo packages that include tests with a known level of testing coverage in both testthat and RUnit format.

Warning: Before running these test be aware that if allowed to generate and opens HTML reports a large number of browser tabs to open. To run these tests call:

```
selfCoverage(testtype = "testthat")
```

3 Known Issues

S4 classes are currently unsupported by this package. Work is being made to add this.

Both testCoverage and one of its dependencies devtools will mask a number of core functions when loaded. This will cause some instability when accessing some features like searching the help documentation. It is recommended that users restart their R session after using testCoverage to be certain no functions remain masked. The possibility of running testCoverage without first loading the package is being considered.

Both RUnit and testthat are suggested packages. One or both will have to be loaded for testCoverage to work however neither is added as a dependency to avoid forcing users to install a package unnecessarily.

4 Future Enhancements

The original author envisaged extending test coverage to include additional metrics code coverage metrics.

Rather than recording whether or not a trace point has been hit, it is expected that this will be replaced with the number of hits occurred during testing. By providing additional granularity to the reporting it becomes possible to determine rarely tested code paths as well as untested ones.

This package currently output reports in HTML and as an R object with much less detail. By finalising the separation of testing and reporting modular report generators could be developed, in additional to the default HTML, to integrate with other services such as continuous integration platforms.

Finally there is interest in expanding the scope of testing to include C code called by R. This would involve inserting trace points into the C code and using a modified `.C` or `.Call` functions to hand back profiling data to R as traditional tools such as gcov does not integrate with the way R loads compiled C functions.