Multi-File Code Coverage in DrRacket

Jonathan Walsh

Computer Science Department
College of Engineering
California Polytechnic State University
2011

Submitted: June 10, 2011
Advisor: John Clements

Contents

Abstract

DrRacket, the integrated development environment for Racket, includes basic code coverage limited to a single file. The main goal of this senior project was to extend DrRacket's code coverage to multiple files. Two additional goals were also included: provide concrete information on the amount of covered code versus uncovered, and implementing the extension with minimal modification to DrRacket's source code. To fulfill these goals the Multi-file Code Coverage Tool was created after first researching DrRacket's code coverage implementation and user interface. The Multi-file Code Coverage Tool is written as a PLaneT package which extends DrRacket without requiring any modifications to DrRacket's source code. The tool adds a new button to DrRacket and new dialogs that display code coverage information.

I Introduction

There are many software development tools and practices that help programmers produce more reliable code, faster and efficiently. One of these tools is code coverage. Code coverage measures the fraction of the program that has been tested [2]. It is useful in helping programmers write tests that execute most, or all, of the source code. Code coverage can be seen as an indirect measure of test quality [2].

DrRacket, the Racket integrated development environment (IDE), includes limited code coverage. It only displays code coverage highlighting on a single file. That is, if a program's test cases are in a separate file, only that file will have coverage highlighting. This is not helpful when trying to determine the code coverage of the entire program, since only the test case file has code coverage. This gives rise to the main goal of this senior project, extending DrRacket's code coverage to multiple files.

DrRacket's code coverage highlighting works by coloring covered code green and uncovered code red. While this does make uncovered code stand out, it only gives a general feeling of the amount of covered versus uncovered code. No details about the code coverage, such as the percent of a file that is covered, is provided to the user. This is the first additional goal: provide the user with concrete code coverage information.

The final additional goal was to implement the extension in such a way that minimal modifications to DrRacket's source code were required. By limiting the modifications to DrRacket's source code distribution would be easier and faster. Additionally, limited modification would reduce the amount of new bugs introduced to DrRacket.

II Background

Racket is a programming language [3]. Included with every download of Racket is an integrated development environment (IDE), DrRacket. DrRacket is the "official" IDE for Racket and is maintained by the same core group of contributors.

DrRacket's user interface is divided into two main areas: the definitions window, by default on top, and the interactions window. The definitions window contains the source file while the interactions window displays the program's output. Above the definitions window are a horizontal list of buttons. Included in this list is the "run" button, which, when pressed, executes the program in the definitions window. See figure 1 for a sample DrRacket environment.

DrRacket's default code coverage can be enabled by selecting "Syntactic Test Suite Coverage" in the "Choose Language" dialog. Once code coverage has been enabled, it is collected every time the program is executed by clicking the "run" button. The code coverage information is then displayed by coloring covered code green and uncovered code red; as seen in figure 1. However, if the entire program is covered no code coloring is done.

In order to determine which sections of code to color, DrRacket must keep track of which expressions of have been executed. DrRacket does this through instrumenting. Instrumenting adds

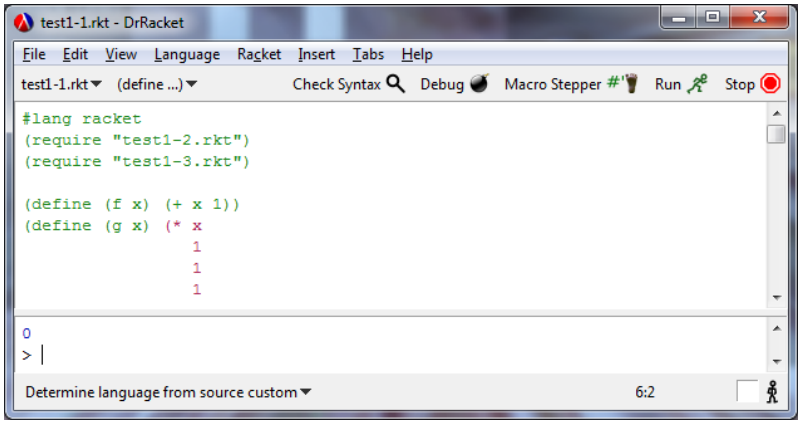


Figure 1: DrRacket

additional code, hidden from the user, before the program is run. This added code surrounds every expression with an integer variable that represents the number of times the expression has been executed. So, if the variable is 0 the expression has not been executed. Then, by examining these variables, it can be determined which sections of code have not been executed. Additionally, these variables can be used to profile which sections of code are heavily used, but is not the focus of this senior project and will not be covered.

DrRacket can be extended in two ways, either through a new collection or a PLaneT package. Collections are tightly coupled with DrRacket and are often distributed with the default installation. PLaneT packages, on the other hand, are more modular. They can be downloaded and automatically installed form a central package repository. Both methods can be used to create new tools for DrRacket. At start up DrRacket looks for tools by reading *info.rkt* files found in collections and installed PLanT packages [1]. These tools can then extend DrRacket through the use of mixins. Every major component of DrRacket is a mixin, from the "Run" button to tabs.

III DrRacket's Code Coverage Framework

Three important facts of DrRacket's framework were heavily used: one, the organization of DrRacket's user interface follows the hierarchy seen in figure 2; two, DrRacket generates a test coverage info hash table that contains information for all uncompiled required files; and three, the test info hash table persists after code coloring has been applied.

DrRacket's user interface is composed of four main classes, seen as boxes in figure 2. The highest class is the *frame*. The *frame* is an entire DrRacket environment as seen in figure 1. Each frame has at least one *tab*. A *tab* always contains a definitions window, where the source code is visible and editable. A *tab* may also contain an interactions window. Additionally, the frame group contains

a list of all currently open *frames*.

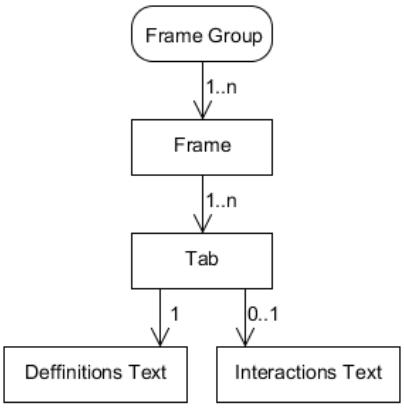


Figure 2: DrRacket User Interface Classes

When a program is run DrRacket generates a hash table, called *test-coverage-info* internally. Included in the hash table are expressions, with their respective source code location, and whether or not they have been evaluated. By examining the contents of the test coverage info hash table, for a program that includes multiple files, we discovered that coverage information is collected for all uncompiled files. So while DrRacket's default code coverage only applies code coloring to the active file, the test coverage info hash table has the information needed to color additional files. Code coverage is not collected for compiled files because the compiled version can not be instrumented to track executed expressions. DrRacket, by default, will load compiled versions of code if possible. However, if it were possible to select between compiled and uncompiled versions, then instrumentation could theoretically be applied to every required file. Figure 3 shows the simplified program flow for code coverage and code highlighting in DrRacket before the solution implemented by this senior project. Starting at **active source file**, when the user presses the "Run" button DrRacket will **require** all the **source files** and **compiled files** needed. Then, if code coverage is enabled, DrRacket **instruments** the code and then **evaluates** it. This produces both the program's **output** and the **test coverage info**. The **test coverage info** is then sent back to the **active source file** and **code coverage coloring** is done.

The test coverage info hash table can be sent to a method of DrRacket's *tab* class, called *show-test-coverage-annotations*, which does code coloring for that tab based on the hash table. This hash remains available for use after code coloring has been completed. In order to clear the test coverage info hash table an additional method, *clear-test-coverage*, must be called.

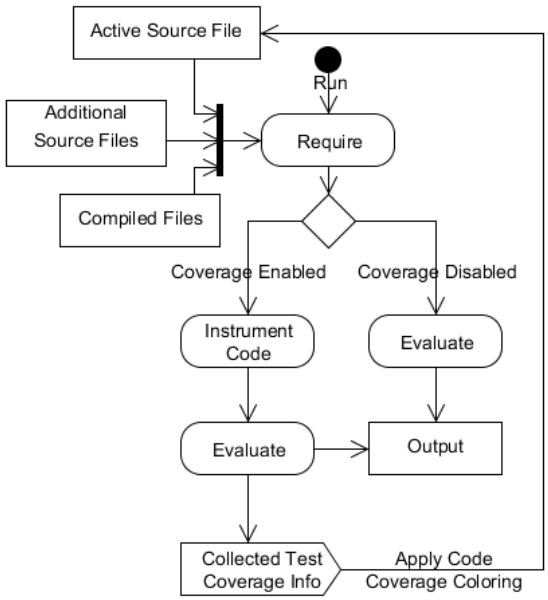


Figure 3: DrRacket Code Coverage Flow

IV Implementation

The Multi-file Code Coverage Tool is implemented as a PLaneT package. This requires no modification to any of DrRacket's source files and satisfies one of the additional goals for the project. PlaneT also brings the benefits of easy distribution, installation, updating, and feedback for the tool. The Multi-file Code Coverage Tool works in the following way: first, searches for a test info hash table; second, colors the code of currently open files; and third, displaces a series of dialogs that give the user code coverage information. This process can be seen in figure 5 on page 5.

The Multi-file Code Coverage Tool adds a new button to DrRacket, labeled "Multi-file Coverage" (Figure 4). Clicking this button will color code in all open tabs using the currently in focus tab's test coverage info hash table. To explain this in another way, if the user opens their test file and runs it, and then clicks the button, all open tabs will be colored relative to that test files. However, if the user switches to another file, before clicking the "Multi-file Coverage" button, the new file's test coverage will be applied. This behavior, while perhaps not immediately obvious, allows for code coverage information and highlighting to be quickly switched between.

After the "Multi-file Coverage" button is pressed, the the first thing done is loading the test coverage info hash table. This hash table contains all the information needed to properly display code coverage. The test coverage info hash table may be found either in the current *tab's* memory or as a saved coverage info file. If it is found in memory, then the program was recently run and

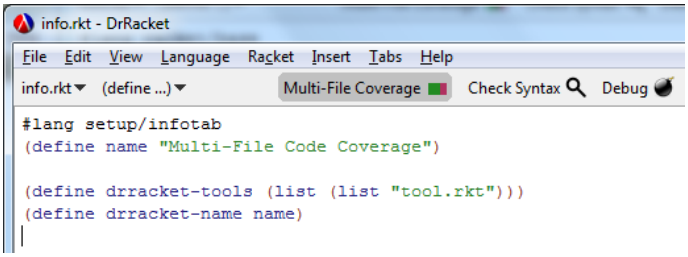


Figure 4: Multi-File Coverage Button in DrRacket

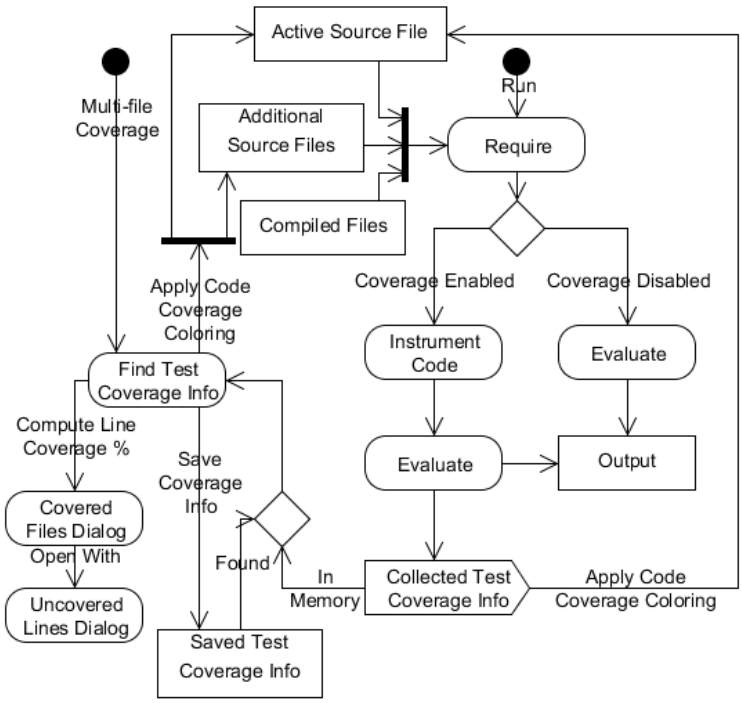


Figure 5: Extended DrRacket Code Coverage Flow

the hash table is as up to date. For this reason, loading the hash table from memory is preferred. However, if the test coverage info could not be found in memory, then it is looked for on the disk. A previously saved hash table can be found in the "compiled" directory, next to the source file. This "compiled" directory is also where DrRacket would place compiled versions of the source program. Every program only has one saved coverage file with a file name that is the same as the source file,

but with a special coverage extension. When the test coverage info is loaded from a saved file, it may be out of date. So, the save file's last modification date is compared to that of the source file's. If the source file was modified more recently then the test coverage info is out of date. A warning, as seen in figure 6, is displayed to inform the user of this. While ignoring the warning means that code coverage will use outdated information, it allows the user to load code coverage information without running the source file again. This could be useful if the source file is large and takes a long time to run. If the test coverage information could not be found, either in memory or in a save file, an error message is displayed, as seen in figure 7. Finally, if the test coverage info was loaded from memory, then the data is written to the coverage save file.

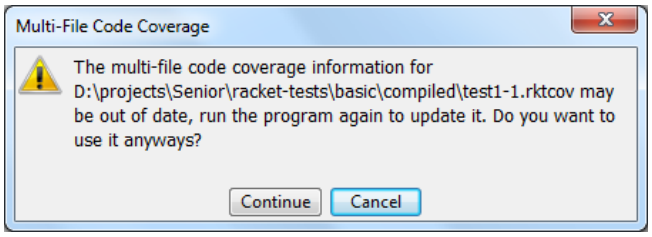


Figure 6: Out Of Date Dialog

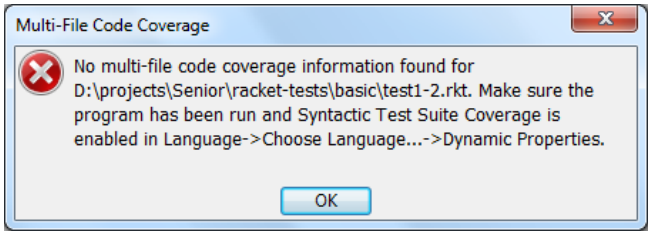


Figure 7: No Coverage Found Dialog

Next the loaded code coverage information is sent to open files that were covered by the source file to do code coloring. Each code expression in the test info coverage hash table has a file name attached to it. By searching through the coverage information, a list of covered files can be computed. The coverage info is then applied to every open file. A list of open files are found by looking through every *tab*, in every *frame* found in the *frame group*. Then, for each open file that is also covered by the source file, the test coverage info is sent to it. No reduction of the test coverage hash table is needed before applying it to a file, even though it will contain irrelevant coverage information. The *show-test-coverage-annotations* method in *tab* will only use the relevant information. During the process of sending coverage information to tabs, it is also computed if the

coverage is valid. Coverage information can become invalid when a file is modified after the coverage information was collected. The test coverage info hash table has no internal way of determining it's validity. So, before it is sent, a check is preformed by comparing the last modification date of the saved coverage file and the source file. If the source file has been more recently than the coverage file its test coverage is considered invalid and no code coloring will be applied to it. This information will also appear as an asterisk in the Covered Files Dialog (Figure 8). As mentioned in section III code coverage is only collected for uncompiled files. Modifying DrRacket to choose between compiled and uncompiled versions of a file would have required modification to DrRacket's source code. This would be in opposition to the goal of requiring minimal, or no, modification to DrRacket's source. Additionally, this modification would have to distinguish between user compiled code and DrRacket's compiled libraries. We decided that the added complexity of this modification would not produce enough utility to warrant it.

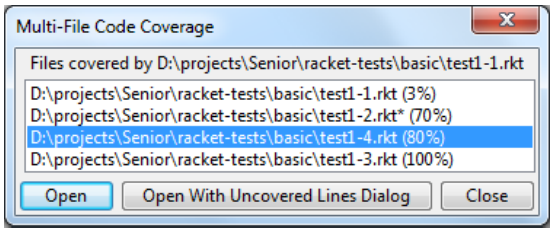


Figure 8: Covered Files Dialog

The final step of the Multi-file Code Coverage Tool is displaying a series of dialogs to inform the user on the collected coverage information. The first one displayed is the Covered Files Dialog (Figure 8). This displays a list of files that have been covered by the source file. Next to each file is its covered line percent; 100% indicating that the entire file is covered. The files are sorted in acending order by covered percent. This is done because it is resonable that the user is more interested in files with uncovered code. The user may then select one, or more, files to switch to, or open if needed, by clicking "Open". If the selected files were not already open, the code coveverage hash table is sent to them so that they may apply code coloring. Additionally, the user may selected the "Open With Uncovered Lines Dialog" indstead. This button will behave like the "Open" button with the addition of displaying the Uncovered Lines Dialog (Figure 9). This displays a list of lines containing uncovered code in the file that was just opened. The list of uncovered lines allows the user to more quickly find lines of interest. Without it, the user must visually scan the entire file for sectons of code that have been colored red.

V Evaluation

The evaluation of this senior project was done in two parts: first, an evaluation of the Multi-file Code Coverage Tool; and second, an evaluation of developing it for DrRacket.

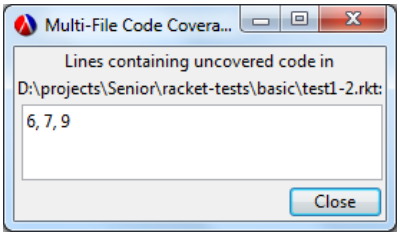


Figure 9: Uncovered Lines Dialog

The Multi-file Code Coverage tool successfully extends DrRacket's code coverage to multiple source files. It does so without requiring any modifications to DrRacket's source code. Additionally, it provides concrete information to the user on the amount of covered code versus uncovered code. These were the main foci of this senior project. The Multi-file Code Coverage tool has been available on PLaneT for, at the time of this writing, approximately two weeks. Thus far, no bugs have been reported.

While the tool meets all of the requirements, it does have a few areas that could be improved upon. First, the detection of valid files does not work in some cases. Specifically, files that have been modified, but not saved, and are not in the focused *tab* of a *frame*, will report that the coverage is valid. Fixing this would require learning more details of the DrRacket user interface and its management. Second, the Multi-file Code Coverage Tool could be slightly better integrated with DrRacket. One possible way of doing this would be extending the "Run" button to automatically save coverage information. Currently, if the user runs the program, modifies it, and then attempts to load multi-file coverage, it will load outdated data. By also integrating the tool with the "Run" button the most recent test coverage information would always be saved. Finally, the parsing of the test coverage info hash table is not particularly fast, especially on larger projects. Further research could be done to improve the efficiency of the algorithm, cache data, or some other method to speed it up.

Overall, developing the Multi-file Code Coverage Tool for DrRacket went smoothly. DrRacket is well documented and there are many tutorials available on docs.racket-lang.org. However, there were a few issues that were encountered during the development of the tool: one, lower level methods and variables were not as well documented; two, finding new information was sometimes difficult; and three, there were limited external Racket resources. All of the issues presented next may actually have solutions that were not discovered. However, if this is the case then another issue is present: understanding DrRacket is difficult and takes a long time. Each issue presented was exhaustively researched by myself, and if I failed to find the correct answer then better documentation and explanations are needed.

While most high level, and often used, functions are well documented, lower level ones are not as well. This presented challenges when trying to determine what the purpose of the function was.

One specific example is the variable called *test-coverage-enabled*. There are actually two separate variables in DrRacket's source, both named *test-coverage-enabled*. One used by DrRacket's user interface and one used by the code coverage generation. Additional documentation could have made this clearer. Overall DrRacket's low level documentation was much better than what I have seen in other projects.

Finding related functions and documentations was not always obvious. One example of this was seen when attempting to find all of the active DrRacket *frames*. I thought I had searched everywhere for a function to find them. It was not until I was pointed to the *group:get-the-frame-group* function that I was able to find the needed functions. No where else had I seen mentions of groups and had no reason to search for them. However, this is not a problem inherent with DrRacket, but one that is present in projects with a large code base.

Finally, since Racket and DrRacket are not as widely used as Java or C++, there were limited external resources on the subject. This presented problems when I would attempt to research, what I felt would be common error messages. Often I would only find one or two results, both on the Racket mailing list. The answers provided there would not always fix my problem. This is in comparison to searching a Java error, where pages of results, that include sites such as Stack Overflow, are displayed. This forced me to ask my advising professor, John Clements, many questions that had simple answers I could have figured out independently had it been a more common language.

VI Conclusion

The main goal of this senior project, extending DrRacket's code coverage to multiple files, was successfully implemented. The two additional goals, providing concrete code coverage information and requiring minimal or no DrRacket source modification, were also completed. Concrete code coverage information was provided through a series of additional dialogs. Since the Multi-file Code Coverage Tool was implemented as a PLaneT package, no modifications to DrRacket's source were required. An additional feature of saving code coverage information was also implemented. While there is certainly areas where the Multi-file Code Coverage Tool could be improved, doing so would require an even greater understanding of the underpinnings of DrRacket.

List of Figures

References

[1] *Implementing DrRacket Plugins*. URL: `http://docs.racket-lang.org/tools/implementing-tools.html`.

[2] Lasse Koskela. *Introduction to Code Coverage*. 2004. URL: `http://www.javaranch.com/journal/2004/01/IntroToCodeCoverage.html`.

[3] *Racket*. URL: `http://racket-lang.org/`.

Appendixes

A Source Code

info.rkt

```
#lang setup/infotab
(define name "Multi-File Code Coverage")

(define drracket-tools (list (list "tool.rkt")))
(define drracket-name name)

(define multi-file-code-coverage-info-file #t) ;used by info-helper to find this file

;Names, labels, and other items that the docs and source code will have in common
(define tool-name name)
(define button-label "Multi-File Coverage")
(define open-with-label "Open With Uncovered Lines Dialog")
(define coverage-suffix ".rktcov")


;Stuff for PLanet
(define blurb
  `("Extends code coverage highlighting to multiple files"))
(define categories `(devtools))
(define primary-file '("main.rkt"))
(define release-notes
  `("Updated Documentation"))
(define version "0.4")
(define repositories '("4.x"))
(define scribblings '(("code-coverage.scrbl" ())))
```

info-helper.rkt

```
#lang racket
;Provides the names of labels and other items that are defined in info.rkt and used across
; tool.rkt and code-coverage.scrbl so that to change the value in all 3 locations only
; info.rkt needs to be updated
(require setup/getinfo)
(require syntax/location)
(provide info-look-up
         coverage-suffix
         tool-name
         button-label
         open-with-label)

(define package-dir
  (let* ([rel-dirs (find-relevant-directories '(multi-file-code-coverage-info-file))])
    (if (> (length rel-dirs) 0)
        (first rel-dirs)
        (current-directory)))
  )

(define info-proc (get-info/full package-dir))

(define (info-look-up name) (if info-proc
                                (info-proc name ( () (symbol->string name)))
                                 ;return something if we fail to find the info file so we dont crash
                                (symbol->string name)))

(define coverage-suffix (info-look-up 'coverage-suffix))
(define tool-name (info-look-up 'tool-name))
(define button-label (info-look-up 'button-label))
(define open-with-label (info-look-up 'open-with-label))
```

tool.rkt

```
#lang racket/base
(require "info-helper.rkt"
         drracket/tool
         drracket/tool-lib
         drracket/private/debug
```

```
         drracket/private/rep
         drracket/private/get-extend
         racket/class
         racket/gui/base
         racket/unit
         racket/serialize
         racket/list
         racket/path
         mrlib/switchable-button
         framework)
(provide tool@)

(define tool@
  (unit
    (import drracket:tool^ )
    (export drracket:tool-exports^)


    (define coverage-button-mixin
      (mixin (drracket:unit:frame<%>) ()
        (super-new)
        (inherit get-button-panel
                 get-definitions-text
                 register-toolbar-button
                 get-tabs
                 get-current-tab
                 get-interactions-text)

        ;Applies code coverage highlighting to all open files. Displays
        ; a dialog with a list of files covered by the currently in focus file
        ; and allows the user to select a file to open and jump to. Coverage
        ; information is first looked for in DrRacket (the program has just been run)
        ; If the program has not been run, or has been modified to invalidate it's
        ; coverage information, look in the compiled directory for coverage info
        ; and display that.
        (define (load-coverage)
          (let* ([current-tab (get-current-tab)]
                 [source-file (send (send current-tab get-defs) get-filename)]
                 [coverage-file (get-temp-coverage-file source-file)]
                 [test-coverage-info-ht (get-test-coverage-info-ht current-tab coverage-file)])
            (when test-coverage-info-ht
              (begin
                (define coverage-report-list (make-coverage-report test-coverage-info-ht coverage-file))

                ;send the coverage info to all files found in the coverage-report
                (map ( (report-item)
                       (let* ([coverage-report-file (string->path (first report-item))]
                              [located-file-tab (locate-file-tab (group:get-the-frame-group) coverage-report-file)])
                         (when (and located-file-tab (is-file-still-valid? coverage-report-file coverage-file))
                           (send located-file-tab show-test-coverage-annotations test-coverage-info-ht #f #f #f))
                         ))
                     coverage-report-list)

                (let* ([frame-group (group:get-the-frame-group)]
                       [choice-pair (get-covered-files-from-user
                                     (format "Files covered by ~a" source-file)
                                     (map ( (item)
                                            (format "~a~a (~a%)"
                                                    (first item)
                                                    (if (second item) "" "*")
                                                    (third item)))
                                          coverage-report-list))]
                       [choice-open-with (first choice-pair)]
                       [choice-index-list (last choice-pair)]
                       )
                  ;switch to or open a new frame with the selected file and display the uncoverd lines in a new dialog
                  (when choice-index-list
                    (map ( (choice-index)
                           (let* ([coverage-report-item (list-ref coverage-report-list choice-index)]
                                  [coverage-report-file (string->path (first coverage-report-item))]
                                  [coverage-report-lines (last coverage-report-item)]
                                  [edit-frame (handler:edit-file coverage-report-file)])
                             (when (and choice-open-with (> (length coverage-report-lines) 0))
                               (send (uncovered-lines-dialog coverage-report-file coverage-report-lines) show #t))

                             ;send coverage info to the newly opened file
```

```
                    (define located-file-tab (locate-file-tab frame-group coverage-report-file))
                    (when (and located-file-tab (is-file-still-valid? coverage-report-file coverage-file))
                      (send located-file-tab show-test-coverage-annotations test-coverage-info-ht #f #f #f))
                    ))
                choice-index-list))


          )))))

    ;Get the given tab's coverage info, either from drrackets current test-coverage-info (if it has been run) or from
    ; a saved one. If the test coverage does not need to be loaded (the program has recently been run) save it to
    ; coverage-file. If the test coverage needs to be loaded then check if the given tab has
    ; been modified since the coverage was saved and display a warning if it has. If no coverage information can
    ; be found display a warning with suggestions to fix it.
    ;drracket:unit:tab<%> path? -> (or #f test-coverage-info)
    (define (get-test-coverage-info-ht current-tab coverage-file)
      (let* ([source-file (send (send current-tab get-defs) get-filename)]
             [interactions-text (get-interactions-text)]
             [test-coverage-info-drracket (send interactions-text get-test-coverage-info)])
        (if test-coverage-info-drracket ;if DrRacket has some test coverage info
            (begin
              (when coverage-file
                  (save-test-coverage-info test-coverage-info-drracket coverage-file))
              (send interactions-text set-test-coverage-info #f) ;clear out drrackets test-coverage-info-ht so we can use
                                                                 ;our coverage file modified time as a reference for when
                                                                 ;coverage was last run
              test-coverage-info-drracket)

            (if (and coverage-file (file-exists? coverage-file)) ;Maybe we have some saved test coverage info?
                (if (and (not (is-file-still-valid? source-file coverage-file)) ;check if the saved info is up to date
                         (not (out-of-date-coverage-message coverage-file))) ;if its not up to date, ask the user if
                                                                             ;they want to use it anyways
                    #f
                    (load-test-coverage-info coverage-file))
                (begin ; no test coverage info, tell the user and how they might collect some
                  (no-coverage-information-found-message source-file)
                  #f)
                ))))



    ;Creates the load coverage button and add it to the menu bar in DrRacket
    (define load-button (new switchable-button%
                             (label button-label)
                             (callback ( (button)
                                         (load-coverage)))
                             (parent (get-button-panel))
                             (bitmap code-coverage-bitmap)
                             ))

    (register-toolbar-button load-button)
    (send (get-button-panel) change-children
          ( (l)
            (cons load-button (remq load-button l))))
    ))

;Takes a test-coverage-info-ht and returns a sorted (alphibetical, but with uncovered
;files first) list of file-name coverage-is-valid? %covered uncovered-lines.
;hasheq path -> (list string?  boolean? integer? (list integer?))
(define (make-coverage-report test-coverage-info-ht coverage-file)
  (let* ([file->lines-ht (make-hash)])
    (hash-for-each test-coverage-info-ht
                   ( (key value)
                     (let* ([line (syntax-line key)]
                            [source (format "~a" (syntax-source key))]
                            [covered? (mcar value)]
                            [file->lines-value (hash-ref file->lines-ht
                                                         source
                                                         (list
                                                          (is-file-still-valid? (string->path source) coverage-file)
                                                          0 ;number of lines in the file
                                                          (list) ;list of uncovered lines
                                                          ))])
                       (hash-set! file->lines-ht source
                                  (list (first file->lines-value)
                                        (max line (second file->lines-value))
```

```
                                    (if (or covered? (member line (last file->lines-value)))
                                        (last file->lines-value)
                                        (sort (append (last file->lines-value) (list line)) <)))
                                ))))
    (let* ([test-coverage-info-list (sort (map ( (item) (list (first item)
                                                              (second item)
                                                              (get-percent (length (last item)) (third item))
                                                              (last item)))
                                               (hash->list file->lines-ht))
                                          ( (a b) (< (third a) (third b))))])
      test-coverage-info-list)))

;Determine if coverage-file's coverage info will still be valid for file. If file has been updated/modified
;more recently than coverage-file this indicates that coverage-file's coverage info may not be valid for file
;
(define (is-file-still-valid? file coverage-file)
  (if coverage-file
      (let* ([file-modify-valid? (> (file-or-directory-modify-seconds coverage-file)
                                    (file-or-directory-modify-seconds file))]
             [located-file-frame (send (group:get-the-frame-group) locate-file file)]
             [file-untouched-valid? (if located-file-frame
                                        ;dosnt work for individual tabs inside of a single frame
                                        (not (send (send located-file-frame get-editor) is-modified?))
                                        #t)])
        (and file-modify-valid? file-untouched-valid?)
        )
      #t ;no coverage-file indicating that the source file has not been saved, so they only way we could have coverage info
      ;is if we got drracket's which means it is up to date.
      ))



;Get the name and location of a code coverage file based on the name of a source file or
;#f if the source file has not been saved. Also creates coverage dir if it does not exisit
;path? -> (or path? #f)
(define (get-temp-coverage-file source-file)
  (if source-file
      (let* ([file-base (file-dir-from-path source-file)]
             [file-name (file-name-from-path source-file)]
             [temp-coverage-file-name (path-replace-suffix file-name coverage-suffix)]
             [temp-coverage-dir (build-path file-base "compiled")]
             [temp-coverage-file (build-path temp-coverage-dir temp-coverage-file-name)])
        (when (not (directory-exists? temp-coverage-dir))
          (make-directory temp-coverage-dir))
        temp-coverage-file
        )
      #f))

;Get a the directory from path
(define (file-dir-from-path path)
  (define-values (file-dir file-name must-be-dir) (split-path path))
  file-dir)

;writes the test-coiverage-info hash table to the given file so that "load-test-coverage-info"
;can reconstruct the hash table
;hasheq path -> void
(define (save-test-coverage-info test-coverage-info coverage-file)
  (with-output-to-file coverage-file
    (lambda () (begin
                 (write (hash-map test-coverage-info
                                  ( (key value)
                                    (cons (list (serialize (syntax-source key))
                                                (syntax-position key)
                                                (syntax-span key)
                                                (syntax-line key))
                                          value))

                                  ))))
      #:mode 'text
      #:exists 'replace
    ))

;Convert the coverage file to a test-coverage-info hash table
;path -> hasheq
(define (load-test-coverage-info coverage-file)
  (make-hasheq (map (lambda (element)
```

```
                    (let* ([key (car element)]
                           [value (cdr element)])
                      (cons (datum->syntax #f (void) (list
                                                      (deserialize (car key))
                                                      (cadddr key)
                                                      1
                                                      (cadr key)
                                                      (caddr key)))
                            (mcons (car value) (cdr value)))))
                  (read (open-input-file coverage-file)))))


;Locates the tab with the given file name from a group of frames
(define (locate-file-tab frame-group file)
  (let*  ([located-file-frame (send frame-group locate-file file)]
          [located-file-tab (if located-file-frame
                                (findf ( (t)
                                         (equal?
                                          (send (send t get-defs) get-filename)
                                          file))
                                       (send located-file-frame get-tabs))
                                #f)])
    located-file-tab))

;Do some random math to try and get a decent hight for a listbox based on the number of items in it
(define (get-listbox-min-height num-items)
  (inexact->exact (min 500 (round (sqrt (* 600 num-items))))))

;compute the percent of uncovered lines rounded to a whole percent. Only return 100 the number of
; uncovered lines is exactly 0
(define (get-percent num-uncovered total)
  (if (= num-uncovered 0)
      100
      (min (round (* (- 1 (/ num-uncovered total)) 100)) 99)))

;Display a message warning the user that the code coverage may be out of date
;returns #t if the user clicks "Continue", #f otherwise
(define (out-of-date-coverage-message file)
  (equal?
   (message-box/custom tool-name
                       (string-append (format "The multi-file code coverage information for ~a" file)
                                      " may be out of date, run the program again to update it."
                                      " Do you want to use it anyways?")
                       "Continue" ;button 1
                       "Cancel" ;button 2
                       #f
                       #f
                       (list 'caution 'default=2))
   1))

;Display a message informing the user that no multi-file code coverage info was found and how they might collect some
(define (no-coverage-information-found-message source-file)
  (message-box tool-name
               (format (string-append "No multi-file code coverage information found for ~a. "
                                      "Make sure the program has been run and Syntactic Test Suite Coverage "
                                      "is enabled in Language->Choose Language...->Dynamic Properties.")
                       (if source-file source-file "Untitled"))
               #f
               (list 'ok 'stop))
  )

;Similar to get-choices-from user, but has two open buttons. One with uncovered lines dialog and one without
;string? (list string?) -> (list boolean? (list integer?))
(define (get-covered-files-from-user message choices)
  (define button-pressed (box 'close))
  (define (button-callback button)
    ( (b e)
      (set-box! button-pressed button)
      (send dialog show #f)))

  (define (enable-open-buttons enable?)
    (send open-button enable enable?)
    (send open-with-button enable enable?)
    (if enable?

        (when (send close-button-border is-shown?)
```

```
        (begin
          (send panel delete-child close-button-border)
          (send panel add-child close-button)))

      (when (send close-button is-shown?)
        (begin
          (send panel delete-child close-button)
          (send panel add-child close-button-border))))
  )

  (define dialog (instantiate dialog% (tool-name)))
  (new message% [parent dialog] [label message])
  (define list-box (new list-box%
                        [label ""]
                        [choices choices]
                        [parent dialog]
                        [style '(multiple)]
                        ;[min-height (get-listbox-min-height (length choices))]
                        [callback ( (c e)
                                    (if (> (length (send list-box get-selections)) 0)
                                        (if (eq? (send e get-event-type) 'list-box-dclick)
                                            ((button-callback 'open) null null)
                                            (enable-open-buttons #t))
                                        (enable-open-buttons #f)))
                                  ]))

  (define panel (new horizontal-panel% [parent dialog]
                     [alignment '(right bottom)]
                     [stretchable-height #f]))


  (define open-button (new button% [parent panel]
                           [label "Open"]
                           [callback (button-callback 'open)]
                           [enabled #f]
                           [style '(border)]))
  (define open-with-button (new button% [parent panel]
                                [label open-with-label]
                                [callback (button-callback 'open-with)]
                                [enabled #f]))
  (define close-button-border (new button% [parent panel]
                                   [label "Close"]
                                   [style '(border)]
                                   [callback (button-callback 'close)]))
  (define close-button (new button% [parent panel]
                            [label "Close"]
                            [style '(deleted)]
                            [callback (button-callback 'close)]))

  (send dialog show #t)
  (case (unbox button-pressed)
    ['open (list #f (send list-box get-selections))]
    ['open-with (list #t (send list-box get-selections))]
    [else (list #f #f)])
  )

; Dialog that displays the uncovered lines. Not a message box so the user can still interact
; with DrRacket without having to close it.
(define (uncovered-lines-dialog file lines)
  (let* ([dialog (instantiate frame% (tool-name))])
    (new message%
         [parent dialog]
         [label "Lines containing uncovered code in"])
    (new message%
         [parent dialog]
         [label (format "~a:" file)])
    (define text-field (new text-field%
         [label ""]
         [parent dialog]
         [style (list 'multiple)]))
    (send text-field set-value (foldl ( (item text)
                              (string-append text
                                             (if (equal? text "") "" ", ")
                                             (number->string item)))
                            "" lines))
    (send (send text-field get-editor) lock #t)
```

```
    (define panel (new horizontal-panel%
                       [parent dialog]
                       [stretchable-height #f]
                       [alignment '(right bottom)]))

    (new button%
         [parent panel]
         [label "Close"]
         [style '(border)]
         [callback ( (b e) (send dialog show #f))])
    dialog))

; Graphic for the code coverage button
(define code-coverage-bitmap
  (let* ((bmp (make-bitmap 16 16))
         (bdc (make-object bitmap-dc% bmp)))
    (send bdc erase)
    (send bdc set-smoothing 'smoothed)
    (send bdc set-pen "black" 1 'transparent)
    (send bdc set-brush "forest green" 'solid)
    (send bdc draw-rectangle 2 5 12 9)
    (send bdc set-brush "maroon" 'solid)
    (send bdc draw-rectangle 11 5 14 9)
    (send bdc set-bitmap #f)
    bmp))

(define (phase1) (void))
(define (phase2) (void))

(drracket:get/extend:extend-unit-frame coverage-button-mixin)))
```

B PLaneT Documentation

Multi-File Code Coverage Tool

June 8, 2011

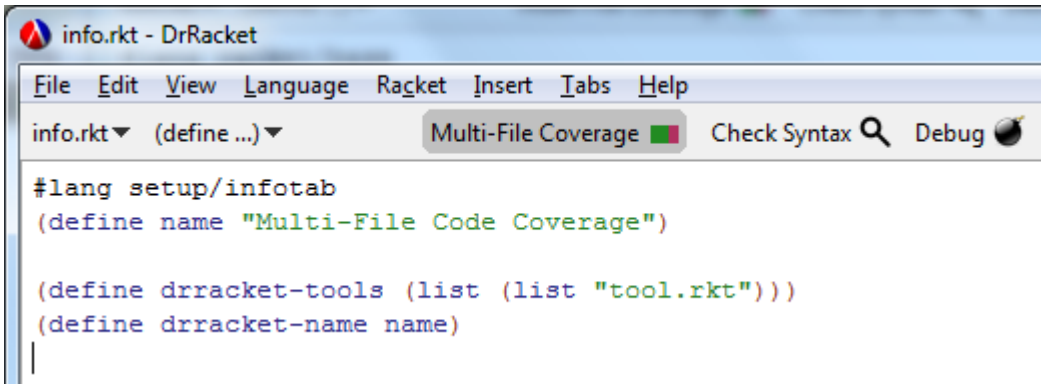
The Multi-File Code Coverage Tool allows coverage information gathered from a single program evaluation to be displayed on multiple source files in multiple DrRacket windows.

1 Installing the Tool

Multi-File Code Coverage is a Planet package, however it only adds a tool rather than providing functions. To install evaluate the following program:

```
#lang racket
(require (planet jowalsh/code-coverage))
```

... and then restart DrRacket. The "Multi-File Coverage" button should now be visible as seen below.



2 Using the Tool

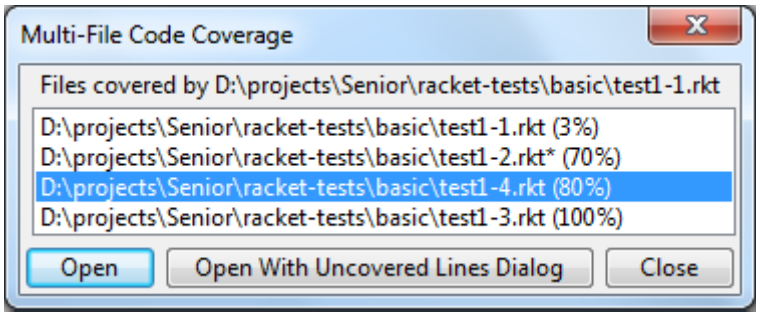
First ensure that you have "Syntactic Test Suite Coverage" enabled in the "Language->Choose Language..." dialog. You may need to click the "Show Details" button to see the expanded options. Then run the program you wish to collect multi-file coverage

information for. Finally, click the "Multi-File Coverage" button. This will send the coverage information to other open files and apply code coverage highlighting to them. The section 3 will also appear containing the list of files covered by the program you just ran.

You may then select one, or more, of the covered files to open and switch focus to. Additionally, by clicking the "Open With Uncovered Lines Dialog", instead of just "Open", each selected file will spawn the section 4 with a list of lines containing unevaluated expressions.

3 Covered Files Dialog

The section 3 appears after clicking the "Multi-File Coverage" button. It displays a list of files covered by the currently in-focus program and allows the selection of a file to open and switch focus to.



Covered Files Dialog

3.1 Line Coverage Percent

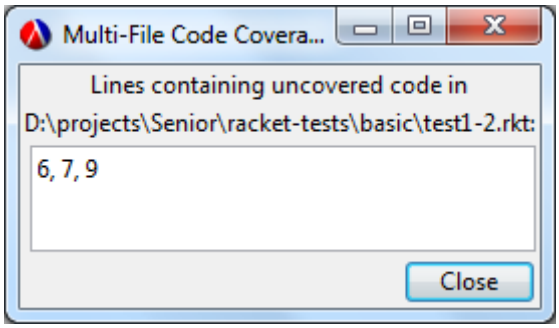
The percent in parentheses, next to each file in the section 3, is the percent of covered lines in that file. So 100% indicates that every line in that file is covered. Blank lines and comments are also counted as covered lines.

3.2 Invalid Coverage Information

An asterisk may appear next to a file in the section 3. This indicates that the file has been modified since multi-file coverage information was last collected, which may have invalidated its coverage info. Multi-file coverage information is not applied to these files. To ensure that all coverage information is valid run the program you are collecting multi-file code coverage for again.

4 Uncovered Lines Dialog

The Uncovered Lines Dialog contains a list of line numbers, for an individual file, that are not fully covered.



Uncovered Lines Dialog

5 Common Questions

5.1 Saving Coverage Information

Whenever the "Multi-File Coverage" button is clicked the multi-file code coverage information is saved to a file. This file is named <file name>.rktcov and saved in the "compiled" directory next to the source file.

5.2 My source file doesn't show up in the covered files dialog

Only un-compiled files will appear in the section 3. To ensure that all your covered files appear delete any "compiled" directories next to your source files. Then run your program again and click the "Multi-File Coverage" button.