Coccinelle Usage (version 1.0.0-rc6)

August 21, 2011

1 Introduction

This document describes the options provided by Coccinelle. The options have an impact on various phases of the semantic patch application process. These are:

1. Selecting and parsing the semantic patch.
2. Selecting and parsing the C code.
3. Application of the semantic patch to the C code.
4. Transformation.
5. Generation of the result.

One can either initiate the complete process from step 1, or to perform step 1 or step 2 individually.

Coccinelle has quite a lot of options. The most common usages are as follows, for a semantic match `foo.cocci`, a C file `foo.c`, and a directory `foodir`:

- `spatch -parse_cocci foo.cocci`: Check that the semantic patch is syntactically correct.
- `spatch -parse_c foo.c`: Check that the C file is syntactically correct. The Coccinelle C parser tries to recover during the parsing process, so if one function does not parse, it will start up again with the next one. Thus, a parse error is often not a cause for concern, unless it occurs in a function that is relevant to the semantic patch.
- `spatch -sp_file foo.cocci foo.c`: Apply the semantic patch `foo.cocci` to the file `foo.c` and print out any transformations as a diff.
- `spatch -sp_file foo.cocci foo.c -debug`: The same as the previous case, but print out some information about the matching process.
- `spatch -sp_file foo.cocci -dir foodir`: Apply the semantic patch `foo.cocci` to all of the C files in the directory `foodir`.
- `spatch -sp_file foo.cocci -dir foodir -include_headers`: Apply the semantic patch `foo.cocci` to all of the C files and header files in the directory `foodir`.

In the rest of this document, the options are annotated as follows:

- ◆: a basic option, that is most likely of interest to all users.
- ⬙: an option that is frequently used, often for better understanding the effect of a semantic patch.
- ◇: an option that is likely to be rarely used, but whose effect is still comprehensible to a user.
- An option with no annotation is likely of interest only to developers.

2 Selecting and parsing the semantic patch

2.1 Standalone options

◆ **-parse_cocci** ⟨**file**⟩ Parse a semantic patch file and print out some information about it.

2.2 The semantic patch

◆ **-sp_file** ⟨**file**⟩, **-c** ⟨**file**⟩, **-cocci_file** ⟨**file**⟩ Specify the name of the file containing the semantic patch. The file name should end in `.cocci`. All three options do the same thing. These options are optional. If they are not used, the single file whose name ends in `.cocci` is assoumed to be the name of the file containing the semantic patch.

◇ **-sp "semantic patch string"** Specify a semantic match as a command-line argument. See the section "Command-line semantic match" in the manual.

2.3 Isomorphisms

◇ **-iso, -iso_file** Specify a file containing isomorphisms to be used in place of the standard one. Normally one should use the `using` construct within a semantic patch to specify isomorphisms to be used *in addition to* the standard ones.

◇ **-iso_limit** ⟨**int**⟩ Limit the depth of application of isomorphisms to the specified integer.

◇ **-no_iso_limit** Put no limit on the number of times that isomorphisms can be applied. This is the default.

◇ **-disable_iso** Disable a specific isomorphism from the command line. This option can be specified multiple times.

-track_iso Gather information about isomorphism usage.

-profile_iso Gather information about the time required for isomorphism expansion.

2.4 Display options

◇ **-show_cocci** Show the semantic patch that is being processed before expanding isomorphisms.

◇ **-show_SP** Show the semantic patch that is being processed after expanding isomorphisms.

◇ **-show_ctl_text** Show the representation of the semantic patch in CTL.

◇ **-ctl_inline_let** Sometimes `let` is used to name intermediate terms CTL representation. This option causes the let-bound terms to be inlined at the point of their reference. This option implicitly sets **-show_ctl_text**.

◇ **-ctl_show_mcodekind** Show transformation information within the CTL representation of the semantic patch. This option implicitly sets **-show_ctl_text**.

◇ **-show_ctl_tex** Create a LaTeX files showing the representation of the semantic patch in CTL.

3 Selecting and parsing the C files

3.1 Standalone options

◆ **-parse_c** ⟨**file**/**dir**⟩ Parse a `.c` file or all of the `.c` files in a directory. This generates information about any parse errors encountered.

◆ **-parse_h** ⟨**file**/**dir**⟩ Parse a `.h` file or all of the `.h` files in a directory. This generates information about any parse errors encountered.

◆ **-parse_ch** ⟨**file**/**dir**⟩ Parse a `.c` or `.h` file or all of the `.c` or `.h` files in a directory. This generates information about any parse errors encountered.

◆ **-control_flow** ⟨**file**⟩, **-control_flow** ⟨**file**⟩:⟨**function**⟩ Print a control-flow graph for all of the functions in a file or for a specific function in a file. This requires `dot` (http://www.graphviz.org/) and `gv`.

◇ **-control_flow_to_file** ⟨**file**⟩, **-control_flow_to_file** ⟨**file**⟩:⟨**function**⟩ Like -control_flow but just puts the dot output in a file in the *current* directory. For PATH/file.c, this produces file:xxx.dot for each (selected) function xxx in PATH/file.c.

◇ **-type_c** ⟨**file**⟩ Parse a C file and pretty-print a version including type information.

-tokens_c ⟨**file**⟩ Prints the tokens in a C file.

-parse_unparse ⟨**file**⟩ Parse and then reconstruct a C file.

-compare_c ⟨**file**⟩ ⟨**file**⟩, **-compare_c_hardcoded** Compares one C file to another, or compare the file tests/compare1.c to the file tests/compare2.c.

-test_cfg_ifdef ⟨**file**⟩ Do some special processing of #ifdef and display the resulting control-flow graph. This requires `dot` and `gv`.

-test_attributes ⟨**file**⟩, **-test_cpp** ⟨**file**⟩ Test the parsing of cpp code and attributes, respectively.

3.2 Selecting C files

An argument that ends in `.c` is assumed to be a C file to process. Normally, only one C file or one directory is specified. If multiple C files are specified, they are treated in parallel, *i.e.*, the first semantic patch rule is applied to all functions in all files, then the second semantic patch rule is applied to all functions in all files, etc. If a directory is specified then no files may be specified and only the rightmost directory specified is used.

◆ **-include_headers** This option causes header files to be processed independently. This option only makes sense if a directory is specified using **-dir**.

◆ **-use_glimpse** Use a glimpse index to select the files to which a semantic patch may be relevant. This option requires that a directory is specified. The index may be created using the script `coccinelle/scripts/glimpseindex_cocci.sh`. Glimpse is available at http://webglimpse.net/. In conjunction with the option **-patch_cocci** this option prints the regular expression that will be passed to glimpse.

◆ **-use_idutils** Use an id-utils index created using lid to select the files to which a semantic patch may be relevant. This option requires that a directory is specified. The index may be created using the script `coccinelle/scripts/ idindex_cocci.sh`. In conjunction with the option **-patch_cocci** this option prints the regular expression that will be passed to glimpse.

◇ **-dir** Specify a directory containing C files to process. A trailing / is permitted on the directory name and has no impact on the result. By default, the include path will be set to the "include" subdirectory of this directory. A different include path can be specified using the option **-I**. **-dir** only considers the rightmost directory in the argument list. This behavior is convenient for creating a script that always works on a single directory, but allows the user of the script to override the provided directory with another one. Spatch collects the files in the directory using `find` and does not follow symbolic links.

-kbuild_info ⟨**file**⟩ The specified file contains information about which sets of files should be considered in parallel.

-disable_worth_trying_opt Normally, a C file is only processed if it contains some keywords that have been determined to be essential for the semantic patch to match somewhere in the file. This option disables this optimization and tries the semantic patch on all files.

-test ⟨**file**⟩ A shortcut for running Coccinelle on the semantic patch "file`.cocci`" and the C file "file`.c`".

-testall A shortcut for running Coccinelle on all files in a subdirectory `tests` such that there are all of a `.cocci` file, a `.c` file, and a `.res` file, where the `.res` contains the expected result.

-test_okfailed, -test_regression_okfailed Other options for keeping track of tests that have succeeded and failed.

-compare_with_expected Compare the result of applying Coccinelle to file`.c` to the file file`.res` representing the expected result.

-expected_score_file ⟨**file**⟩ which score file to compare with in the testall run

3.3 Parsing C files

◇ **-show_c** Show the C code that is being processed.

◇ **-parse_error_msg** Show parsing errors in the C file.

◇ **-verbose_parsing** Show parsing errors in the C file, as well as information about attempts to accomodate such errors. This implicitly sets -parse_error_msg.

◇ **-type_error_msg** Show information about where the C type checker was not able to determine the type of an expression.

◇ **-int_bits ⟨n⟩, -long_bits ⟨n⟩** Provide integer size information. n is the number of bits in an unsigned integer or unsigned long, respectively. If only the option **-int_bits** is used, unsigned longs will be assumed to have twice as many bits as unsigned integers. If only the option **-long_bits** is used, unsigned ints will be assumed to have half as many bits as unsigned integers. This information is only used in determining the types of integer constants, according to the ANSI C standard (C89). If neither is provided, the type of an integer constant is determined by the sequence of "u" and "l" annotations following the constant. If there is none, the constant is assumed to be a signed integer. If there is only "u", the constant is assumed to be an unsigned integer, etc.

◇ **-no_loops** Drop back edges for loops. This may make a semantic patch/match run faster, at the cost of not finding matches that wrap around loops.

-use_cache Use preparsed versions of the C files that are stored in a cache.

-cache_prefix Specify the directory in which to store preparsed versions of the C files. This sets -use_cache

-cache_limit Specify the maximum number of preparsed C files to store. The cache is cleared of all files with names ending in .ast_raw and .depend_raw on reaching this limit. Only effective if -cache_prefix is used as well. This is most useful when iteration is used to process a file multiple times within a single run of Coccinelle.

-debug_cpp, -debug_lexer, -debug_etdt, -debug_typedef Various options for debugging the C parser.

-filter_msg, -filter_define_error, -filter_passed_level Various options for debugging the C parser.

-only_return_is_error_exit In matching "`...`" in a semantic patch or when forall is specified, a rule must match all control-flow paths starting from a node matching the beginning of the rule. This is relaxed, however, for error handling code. Normally, error handling code is considered to be a conditional with only a then branch that ends in goto, break, continue, or return. If this option is set, then only a then branch ending in a return is considered to be error handling code. Usually a better strategy is to use `when strict` in the semantic patch, and then match explicitly the case where there is a conditional whose then branch ends in a return.

Macros and other preprocessor code

◆ **-macro_file ⟨file⟩** Extra macro definitions to be taken into account when parsing the C files. This uses the provided macro definitions in addition to those in the default macro file.

◆ **-macro_file_builtins ⟨file⟩** Builtin macro definitions to be taken into account when parsing the C files. This causes the macro definitions provided in the default macro file to be ignored and the ones in the specified file to be used instead.

◇ **-ifdef_to_if,-no_ifdef_to_if** The option **-ifdef_to_if** represents an `#ifdef` in the source code as a conditional in the control-flow graph when doing so represents valid code. **-no_ifdef_to_if** disables this feature. **-ifdef_to_if** is the default.

◇ **-use_if0_code** Normally code under #if 0 is ignored. If this option is set then the code is considered, just like the code under any other #ifdef.

-noadd_typedef_root This seems to reduce the scope of a typedef declaration found in the C code.

Include files

- **-recursive_includes, -all_includes, -local_includes, -no_includes** These options control which include files mentioned in a C file are taken into account. **-recursive_includes** indicates that all included files mentioned in the .c file(s) or any included files will be processed. **-all_includes** indicates that all included files mentioned in the .c file(s) will be processed. **-local_includes** indicates that only included files in the current directory will be processed. **-no_includes** indicates that no included files will be processed. If the semantic patch contains type specifications on expression metavariables, then the default is **-local_includes**. Otherwise the default is **-no_includes**. At most one of these options can be specified.

- **-I** ⟨**path**⟩ This option specifies a directory in which to find non-local include files. This option can be used several times.

- **-relax_include_path** This option when combined with -all_includes causes the search for local include files to consider the current directory, even if the include patch specifies a subdirectory. This is really only useful for testing, eg with the option **-testall**

- **-c++** Make an extremely minimal effort to parse C++ code. Currently, this is limited to allowing identifiers to contain "::", tilde, and template invocations. Consider testing your code first with spatch -type_c to see if there are any type annotations in the code you are interested in processing. If not, then it was probably not parsed.

4 Application of the semantic patch to the C code

4.1 Feedback at the rule level during the application of the semantic patch

- **-show_bindings** Show the environments with respect to which each rule is applied and the bindings that result from each such application.

- **-show_dependencies** Show the status (matched or unmatched) of the rules on which a given rule depends. **-show_dependencies** implicitly sets **-show_bindings**, as the values of the dependencies are environment-specific.

- **-show_trying** Show the name of each program element to which each rule is applied.

- **-show_transinfo** Show information about each transformation that is performed. The node numbers that are referenced are the number of the nodes in the control-flow graph, which can be seen using the option **-control_flow** (the initial control-flow graph only) or the option **-show_flow** (the control-flow graph before and after each rule application).

- **-show_misc** Show some miscellaneous information.

- **-show_flow** ⟨**file**⟩, **-show_flow** ⟨**file**⟩:⟨**function**⟩ Show the control-flow graph before and after the application of each rule.

-show_before_fixed_flow This is similar to **-show_flow**, but shows a preliminary version of the control-flow graph.

4.2 Feedback at the CTL level during the application of the semantic patch

◆ **-verbose_engine** Show a trace of the matching of atomic terms to C code.

◇ **-verbose_ctl_engine** Show a trace of the CTL matching process. This is unfortunately rather voluminous and not so helpful for someone who is not familiar with CTL in general and the translation of SmPL into CTL specifically. This option implicitly sets the option **-show_ctl_text**.

◇ **-graphical_trace** Create a pdf file containing the control flow graph annotated with the various nodes matched during the CTL matching process. Unfortunately, except for the most simple examples, the output is voluminous, and so the option is not really practical for most examples. This requires `dot` (http://www.graphviz.org/) and `pdftk`.

◇ **-gt_without_label** The same as **-graphical_trace**, but the PDF file does not contain the CTL code.

◇ **-partial_match** Report partial matches of the semantic patch on the C file. This can be substantially slower than normal matching.

◇ **-verbose_match** Report on when CTL matching is not applied to a function or other program unit because it does not contain some required atomic pattern. This can be viewed as a simpler, more efficient, but less informative version of **-partial_match**.

4.3 Actions during the application of the semantic patch

◆ **-D rulename** Run the patch considering that the virtual rule "rulename" is satisfied. Virtual rules should be declared at the beginning of the semantic patch in a comma separated list following the keyword virtual. Other rules can depend on the satisfaction or non satifaction of these rules using the keyword `depends on` in the usual way.

◆ **-D variable=value** Run the patch considering that the virtual identifier metavariable "variable" is bound to "value". Any identifier metavariable can be designated as being virtual by giving it the rule name `virtual`. An example is in demos/vm.coci

◇ **-allow_inconsistent_paths** Normally, a term that is transformed should only be accessible from other terms that are matched by the semantic patch. This option removes this constraint. Doing so, is unsafe, however, because the properties that hold along the matched path might not hold at all along the unmatched path.

◇ **-disallow_nested_exps** In an expression that contains repeated nested subterms, *e.g.* of the form `f(f(x))`, a pattern can match a single expression in multiple ways, some nested inside others. This option causes the matching process to stop immediately at the outermost match. Thus, in the example `f(f(x))`, the possibility that the pattern `f(E)`, with metavariable `E`, matches with `E` as `x` will not be considered.

◇ **-no_safe_expressions** normally, we check that an expression does not match something earlier in the disjunction. But for large disjunctions, this can result in a very big CTL formula. So this option give the user the option to say he doesn't want this feature, if that is the case.

◇ **-pyoutput coccilib.output.Gtk, -pyoutput coccilib.output.Console** This controls whether Python output is sent to Gtk or to the console. **-pyoutput coccilib.output.Console** is the default. The Gtk option is currently not well supported.

-loop When there is "..." in the semantic patch, the CTL operator AU is used if the current function does not contain a loop, and AW may be used if it does. This option causes AW always to be used.

◇ **-ocaml_regexps** Use the regular expressions provided by the OCaml `Str` library. This is the default if the PCRE library is not available. Otherwise PCRE regular expressions are used by default.

-steps ⟨**int**⟩ This limits the number of steps performed by the CTL engine to the specified number. This option is unsafe as it might cause a rule to fail due to running out of steps rather than due to not matching.

-bench ⟨**int**⟩ This collects various information about the operations performed during the CTL matching process.

-popl, -popl_mark_all, -popl_keep_all_wits These options use a simplified version of the SmPL language. **-popl_mark_all** and **-popl_keep_all_wits** implicitly set **-popl**.

5 Generation of the result

Normally, the only output is a diff printed to standard output, containing the differences between the original code and the transformed code. If stars are used in column 0 rather than - and +, then the - lines in the diff are the lines that matched the stars.

◆ **-keep_comments** Don't remove comments adjacent to removed code.

◆ **-linux_spacing, -smpl_spacing** Control the spacing within the code added by the semantic patch. The option **-linux_spacing** causes spatch to follow the conventions of Linux, regardless of the spacing in the semantic patch. This is the default. The option **-smpl_spacing** causes spatch to follow the spacing given in the semantic patch, within individual lines.

◇ **-o** ⟨**file**⟩ This causes the transformed code to be placed in the file `file`. A diff is still printed to the standard output. This option only makes sense when - and + are used.

◇ **-in_place** Modify the input file to contain the transformed code. A diff is still printed to the standard output. By default, the input file is overwritten when using this option, with no backup. This option only makes sense when - and + are used.

◇ **-backup_suffix** The suffix of the file to use in making a backup of the original file(s). This suffix should include the leading ".", if one is desired. This option only has an effect when the option `-in_place` is also used.

◇ **-out_place** Store the result of modifying the code in a .cocci_res file. A diff is still printed to the standard output. This option only makes sense when - and + are used.

◇ **-no_show_diff** Normally, a diff between the original and transformed code is printed on the standard output. This option causes this not to be done.

◇ **-U** Set number of diff context lines.

◇ **-patch** ⟨**path**⟩ The prefix of the pathname of the directory or file name that should dropped from the diff line in the generated patch. This is useful if you want to apply a patch only to a subdirectory of a source code tree but want to create a patch that can be applied at the root of the source code tree. An example could be `spatch -sp_file foo.cocci -dir /var/linuxes/linux-next/drivers -patch /var/linuxes/linux-next`. A trailing / is permitted on the directory name and has no impact on the result.

◇ **-save_tmp_files** Coccinelle creates some temporary files in `/tmp` that it deletes after use. This option causes these files to be saved.

-debug_unparsing Show some debugging information about the generation of the transformed code. This has the side-effect of deleting the transformed code.

6 Other options

6.1 Version information

◆ **-version** The version of Coccinelle. No other options are allowed.

◆ **-date** The date of the current version of Coccinelle. No other options are allowed.

6.2 Help

◆ **-h, -shorthelp** The most useful commands.

◆ **-help, –help, -longhelp** A complete listing of the available commands.

6.3 Controlling the execution of Coccinelle

◆ **-timeout** ⟨**int**⟩ The maximum time in seconds for processing a single file.

◇ **-max** ⟨**int**⟩ This option informs Coccinelle of the number of instances of Coccinelle that will be run concurrently. This option requires **-index**. It is usually used with **-dir**.

◇ **-index** ⟨**int**⟩ This option informs Coccinelle of which of the concurrent instances is the current one. This option requires **-max**.

◇ **-mod_distrib** When multiple instances of Coccinelle are run in parallel, normally the first instance processes the first n files, the second instance the second n files, etc. With this option, the files are distributed among the instances in a round-robin fashion.

-debugger Option for running Coccinelle from within the OCaml debugger.

-profile Gather timing information about the main Coccinelle functions.

-disable_once Print various warning messages every time some condition occurs, rather than only once.

6.4 Miscellaneous

◇ **-quiet** Suppress most output. This is the default.

-pad, -hrule ⟨dir⟩, -xxx, -l1