Documentation for `spgen`

Chi Pham

October 26, 2015

Contents

1 About

We first provide information about the tool, its motivation, and its uses.

1.1 What is spgen?

spgen is a Coccinelle[1] metaprogramming tool that can generate hardened semantic patches for use in e.g. the Linux kernel.
Or, in less fancy words, spgen can take your simple Coccinelle script containing one or more rules with *, +, or -, and then output the same script with more options.

In particular, spgen generates the patch, context, org, and report virtual rules. These options are prevalent in the Coccinelle scripts included in the Linux kernel and are used in the following cases:

- patch: Used for +/- rules that transform the matched C code and output the changes in Unix diff format.
- context: Used for * rules that find the matched C code and output it in diff-like format.
- org: Used for script rules that output matches in emacs org mode format[2] with an error message and line numbers.
- report: Used for script rules that output matches with an error message and line numbers.

1.2 Features

spgen includes (but is not limited to) support for

- Generating a context (aka *) version of a patch (aka +/-) rule.
- Generating org and report (aka script) versions of both patch and context rules.
- Adding patch, context, org, and report dependencies to rule headers.
- Allowing the user to specify preface information for the generated rule, such as keywords, options, etc. as well as error messages for rules and names for nameless rules.
- Automatic rulename and error message generation when none specified by the user.
- Rule splitting to ensure correct context mode output for rules containing pattern matching disjunctions.
- And more ...

[1]http://coccinelle.lip6.fr/
[2]http://orgmode.org/

2 Usage

This section contains information about how the tool is installed and used.

2.1 Installation and uninstallation

To install: You need to have `Coccinelle` and all its dependencies installed. You also need to have the `Coccinelle` source code available. By default, the `spgen` directory lies within the `Coccinelle` directory in `coccinelle/tools/spgen`.[3] From the `spgen` directory, do:

1. Run `make` to compile the program.
2. Run `make install`[4] to install the program.
3. (Optional) Try out the program by running `spgen examples/addvoid.cocci` or `spgen <path_to_your_script>.cocci`.

To uninstall: From the `spgen` directory, do:

1. Run `make uninstall`[4].

2.2 Running the program

The most common usages are as follows, for a semantic patch file `foo.cocci` and an spgen config file `foo.config`:

- `spgen foo.cocci`: Generate the file with the information found d in `foo.config` if it exists. If not, the program is run in interactive mode.
- `spgen --config foo.config foo.cocci`: Generate the file with `foo.config` as the configuration file. The shorthand `-c` can be used instead of `--config`.
- `spgen foo.cocci --interactive`: Run the program in interactive mode. The shorthand `-i` can be used instead of `--interactive`.

Additional options:

- `--default`: Generates the file, entirely using default values, such as generic error messages, instead of user input. Can, e.g., be used to quickly check the generated context rule(s).
- `-o <filename>`: Saves the generated file to `<filename>` instead of printing it to standard output.
- `--no-output`: Generates the file, but doesn't output the result.
- `-help, --help`: Displays the list of options.

[3]If you change this, you need to modify the `COCCIDIR` path in `source/Makefile` to point to the `Coccinelle` source code directory.

[4]If permission is denied, run it in superuser mode e.g. `sudo make install`, depending on your system.

2.3 User input

When generating a script, `spgen` might need some extra information from the user. There are two kinds of information:

- **Preface**: information to go at the beginning of the script. Contains metainformation about the script such as description, author, etc. See the full list in Section 2.3.2.
- **Rule information**: rule-specific info, such as error messages that are output in `org` and `report` mode.

There are two ways of passing this information, interactive mode and configuration mode.

2.3.1 Interactive mode

In interactive mode, the program prompts the user for the information through the commandline. The user can then choose to save the information into an `spgen` config file, which can be further modified and reused in configuration mode.

2.3.2 Configuration mode

In configuration mode, the program looks for an `spgen` config file to provide the needed information to generate the file. For `<file>`.cocci, the config file should be called `<file>`.config.

The user-specified attributes in the **preface** are, in no specific order ((r) means required):

Attribute name	**Shorthand**	**Value**	**Description**
`description` (r)	`d`	Single-value text	Describes what the Coccinelle script does.
`confidence` (r)	`c`	Low, Moderate, High	Confidence level for the script.
`authors`	`a`	Multi-value text	Authors of the script, including affiliation and license.
`url`	`u`	Single-value text	URL for the script.
`limitations`	`l`	Multi-value text	Limitations for the script.
`keywords`	`k`	Single-value text	Keywords for the script.
`options`	`o`	Single-value text	`spatch` options with which to run the script.
`comments`	`m`	Single-value text	Additional comments.

The **rule information** contains error messages for `org` and `report`, and possibly rule names for rules that are unnamed in the original `Coccinelle script`.

The syntax for the `spgen` config files is rather simple, and the easiest way to learn it is to run `spgen` in interactive mode and study the resulting config. But for completeness ...:

- The syntax for attributes is

  ```
   <attribute_name> = <value>
  ```

  where `<attribute_name>` can be either the attribute name or its shorthand. The end of `<value>` is marked by a newline. It is therefore not possible to insert newlines in any of the values.

- For multi-valued attributes, values are delimited by pipes, `|`, ie.

  ```
  <attribute_name> = <value_1>|<value_2>|...|<value_n>
  ```

- Error messages for rules follow the syntax

  ```
  <rule_name> =
    org:<message>
    report:<message>
  ```

  for `org` and `report` error messages, respectively. Here, `<rule_name>` is either the actual rule name, or, if it is a nameless rule, `<line_that_rule_starts_on>:<new_name>`. Meanwhile, `<message>` follow the syntax of `python` format strings, e.g.

  ```
  "This is a message that references two metavariables %s and %s." % (x,y)
  ```

  where `x` and `y` are metavariables in the rule. If using metavariables from another rule, write `<other_rule_name>.<metavar_name>`. If using no metavariables, just write the error message surrounded by quotes.

- Comments can be written in `C`-style, ie. `//` and `/**/`.

2.4 Examples

Example files can be found in the `examples` directory. For each example, there should be four files. The file extensions denote the following:

- `<name>.c`: `C` source file that returns matches/patches for the corresponding cocci file. Can be tested with `spatch -sp-file <name>.cocci <name>.c`.

- `<name>.cocci`: simple, unhardened Coccinelle script.

- `<name>.config`: `spgen` configuration file for specifying preface and rule information.

- `<name>_.cocci`: expected output when running `spgen` on the unhardened Coccinelle script with the config file. Should be a valid, hardened Coccinelle script. Can be tested with e.g. `spatch -sp-file <name>_.cocci <name>.c -D report -no-show-diff`.

3 Implementation

This section contains information about the implementation details of the tool. Much of the documentation of the implementation is contained in the interface files of the source code.

TODO: make a graph illustrating the workflow.

Some notes:

- See `spgen/source/README.md` for a broad overview of the workflow and dependencies, read `spgen/source/spgen.ml` for the driver of the program.
- Several of the modules make extensive use of the AST0 visitor[5]. It is used because it abstracts away a lot of the boilerplace code needed for accessing the components of the abstract syntax tree.
- An easy way to debug in `spgen/source/rule_body.ml`: add

  ```
  >> Snapshot.add "debug message" >>
  ```

  in some function sequence. Then "debug message" will appear in the exact same place it was called in the generated script.
- Absolutely not optimised for performance (in particular, memory).
  - Snapshot need not be purely functional; rule map can be hashtable in mutable record field instead of map (needs to be sorted in get_result however). However, a map might be beneficial later on if we want to keep various copies for e.g. rule splitting.
  - Rule map is converted to string list before printing; no need to do so, could just print directly from rule map. However, this makes for a better separation interface-wise.
  - Most importantly, the generally small size of Coccinelle scripts means that performance is not actually a problem in practice.
- spgen needs its own flag in the Coccinelle parser: `Flag_parsing_cocci.generating_mode`. This ensures that dependencies are not optimised away in the parser, as we need that information for printing the rules properly. It cannot be substituted for the `ignore_patch_or_match` option, because that option also affects other parts of the parser.
- Regression tests: run spgen with flag `--test <path_to_test_dir` to run the tests. They should be run once before changing anything in the code. The tests test diff equality with expected files (ie. a bit too strict) and that generated files are parsable.

[5] `coccinelle/parsing_cocci/visitor_ast0.ml`

4 Known issues

This section lists the known issues that might either cause the tool to fail or to generate an erroneous script.

- **Missing information in rule headers**: Rule headers will not be generated correctly if the original rules contain `extends` or `expression`. Those qualifiers will be missing in the generated rule since they are not included in the output of the parser.
  Fix: Change the parser to include this information.
- **Typedefs in rule headers**: If there are meta typedefs in the original rule headers, they will be included in every generated rule that uses the type. This causes an error when using the generated script since meta typedefs can only be declared once.
  Fix: Remove the error check in the parser since this should not cause an error.
- Disjunction generation has a number of issues:
  - **Selecting wrong position in statement dots cases**: There must be the same number of positions in each disjunction case, otherwise `org` and `report` will only match when all positions can be found. In statement dots disjunctions, this is currently solved by putting the position at the first possible statement. The issue here is that, if the case contains many statements, only the first surrounding statement will be highlighted instead of the important part.
    Fix: Implement support for finding a single best statement in a statement dots (list of statements).
  - **Nested disjunctions**: The position counter is frozen within a disjunction. But if there is a nested disjunction inside it, the same position will be used in both disjunction levels, causing a nonsensical script.
    Fix: Keep track of the current nest and name the position accordingly.
- **No format string metavariable check**: The user can specify metavariables to be used in the messages for `org` and `report` mode. The program currently does not check if the declared metavariables actually exist in the original rule.
  Fix: Implement check of metavariables.
- **Dependencies between patch rules**: It is possible to make patch rules that depend on other patch rules modifying the code. E.g. if one patch rule transforms f(0) and one transforms f(e), then f(0) will only match the first, since it is transformed to something else when it reaches the f(e) rule. But in `context` mode, both rules will print the f(0) occurrence.
  Fix: ??? Somehow detect that two rules will match the same case and insert constraints such that any match in subsequent rules does not match the first one.
- **Dependencies in context rules**: In a `context` script, if there are dependencies between rules, they might be mixed up. This happens if there are rules dependent on the generated rules since they will then be printed before the rules on which they are

dependent!
Fix: ???

- **Type and switch case disjunctions**: Currently, the program fails if attempting to generate a `Coccinelle` script with type or switch case disjunctions. The failure happens in the position generator. The reason it is not implemented is that it requires quite a lot of code for a case that rarely appears.
  Fix: Implement full position generation for types and switch cases.

- **File transformation not context-dependent**: When printing the transformed script, a somewhat primitive string replacement strategy is used to rename rules and inject dependencies (e.g. change `@rulename@` to `@rulename depends on patch@`). The transformation is not context-dependent which means that the script will get mangled if there is e.g. a `@@` *inside comments* on a new line, since this will be mistaken for a SmPL-syntax `@@`.
  Fix: Use simple parsing in the file transformation. Another solution is to not transform the original file at all, but to completely reconstruct it from the AST (this requires a pretty printer for AST that retains comments and plus slices!). In practice, however, this is only a problem in very specific cases that are unlikely to occur.

5 Future work

This section lists the work to be done on the tool aside from fixing the known issues.

- **Global configuration file**: Some of the preface attributes remain more or less constant across generated scripts, such as author and url. Furthermore, there are some hardcoded values in the program that should be configurable, such as the default values for rulenames, position names, and error messages.
  It could be useful to have some kind of configuration file that collects these values and reuses them each time a script is generated.

- **Generic rule splitting**: The disjunction rule generation implementation currently doesn't lend itself very well to possible expansion. For instance, if there was another case where we would like to expand a rule into several rules, how could we do this in a generalised way? How should this work with several split rules?
  Example: any construct where the patch rule changes something other than itself, e.g. function declarations; if the declared function has a prototype, the prototype is changed as well in `patch` mode, but this has to be done explicitly with two rules in `context` mode.
  Discussion: consider splitting generation of extra rules out into its own module, one for each type of split rule. Essentially using multiple passes over the AST0 to get the rules, one pass per rule type. This is however slightly complicated by the fact that context rule generation should be modified if there is a disjunction.
  Note: could have following design (prolly not though):

  - Given ast0 rule:
  - Generate positions using rebuilder (rebuilder works like combiner except its functions are 'a -> 'a and all functions must be like "take an ast0 component and return an ast0 component of the same type". This is different from the combiner in that the combiner says "take an ast0 component and return a 'a", where ALL ast0 components are converted into a 'a, whereas for the rebuilder, the statement is turned into a statement, the expression is turned into an expression etc.). Only problem is that it modifies state: we need to know the name of the added metaposition ...
  - turn to pretty-print strings (at this stage, stars and context mode + whencodes are handled as well)
  - things to watch out for: no_gen mode (in whencodes), disjunction handling, inc_star, whencode handling, context_mode star generation,

- **Stars and braces**: If a braced statement is starred, it would be nicer if the braces were not on the same lines as the stars.

- **Character limit**: Ensuring character limit in the generated rule. This is currently implemented for the preface, but not for rule headers and script rules.

- Perhaps rethink position generation at some point. If the script already contains minuses, we would rather put the positions there compared to the heuristic version.